Book

The Entity Modeler

New patterns and transformations for structural modeling

V7

SORRY NO UML YET, BUT THIS DRAFT IS REVIEWABLE

Contents

1. Introduction

This is not the first book on entity modelling for enterprise applications. This one begins by reviewing well-established principles. But it goes on to say many new things, and it ends up challenging the prevailing orthodoxies.

Is this a book on logical data modeling for analysts? Yes. It is certainly about ways to define a logical data model that specifies business rules.
Is this a book on physical data modeling for database designers? Not really. It does have some relevance to defining a physical data model that becomes an efficient database structure, but this is not the main aim.
Is this a book on class diagramming for OO programmers? Yes. It is meant to be about ways to define an OOP "class diagram" that can be implemented as Java on an app server.
This book is perhaps the first to take a view of entity modelling that spans conventional relational data analysis and object-oriented software design for enterprise applications. It presents principles that apply to drawing database schemas and object-oriented class diagrams. It suggests object-oriented and relational paradigms are narrow views of a broader field.

The book presents a new knowledgebase of patterns for drawing an entity model. Entity 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 structure that meets performance requirements and can be readily maintained. The book presents a new knowledgebase of transformations between closely related patterns that help you to meet these goals.

Readers will range from analysts and designers working on enterprise applications, to students of computer science. For readers familiar with UML, an entity model is a kind of a class diagram; you may choose to stereotype the classes with <<entity>> under the class name.

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.

At the back end of an enterprise application are the layers of software that manage the stored data. Software that manages data about the real world is crucial. The enterprise has to invest a lot of time and money in specifying and coding business rules to ensure its stored data tells a true and consistent story about the business domain the enterprise operates in. Data integrity is *the* challenge for enterprise architects.

> "If users don't trust your data, your database is garbage" IT director of a major telecommunications company.

> "Problems with data integrity at one company turned a 2 month exercise into an 18 month exercise." Information Week May 19th 1997.

Almost every enterprise application is built on top of an entity model. Yet knowledge of how to build an entity model to capture business rules and ensure data integrity is thinly distributed. This book is immediately help to anybody working on an entity model.

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. This book also sheds light on various debates about object-oriented software design. E.g. it illuminates the difference between type-based and state-based theories of what a class it.

2. Preface

2.1 Entity models

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 behavioral things/features in a companion volume, “The Event Modeler”, and only very briefly in this one. This book focuses on modelling structural things/features in the form of an entity model; see the What column 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** / **Entity model**	Behavioral Model	
		Event models	State machine models
	What	How	When
Enterprise model	David Hay’s interest		
Enterprise application model	**This book**	“The Event Modeler”	“The Event Modeler”
Technology model			

David Hay (ref 99), who draws entity models for the purpose of enterprise modeling, has given me permission to copy substantial chunks of my email dialogues with him.

In ordinary conversation, entity can mean an entity instance (an object) or an entity type (a class of objects). 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 entity and model according to their context, as you do in conversation.

An entity model lies at the heart any enterprise application. An entity model shows a structure of entities, attributes and relationships, annotated with business rules. Entity models are used by software engineers working on enterprise applications to:

- refine a higher-level enterprise model for a specific project
- facilitate discussions with business people and clarify requirements
- specify the business rules and processing constraints for developers
- specify an object-oriented class structure from which business services are coded
- specify a database structure against which data services are coded

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 Modeller. The Modeller 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 Modeller keeps an entity model simple, is aware of different modeling options, understands trade offs between them, and introduces complexity only when and where it is needed. The Agile Modeller 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 the entity model "right" is less straightforward than a course in object-oriented design or relational data analysis might lead you to believe. Entity modelers face awkward questions to be explored later. When should I include a class hierarchy in an entity model? When should I build constraints into an entity model and/or database structure? 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) and other kinds of model. 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 components, 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. PART ONE: BUSINESS RULES IN ENTITY MODELS

4. Business rules in entity models

This chapter introduces some the basic ideas about business rules, because the use of entity models to specify business rules is a theme that runs through many later chapters. It discusses the specification of structural terms and facts; these appear as the names of entities, attributes and relationships in an entity model. It also discusses the specification of structural constraints and derivation rules which have to be coded into the enterprise application one way or another.

4.1 Structural terms

An entity model is built around entity types or classes. Model builders often start by naming entities, then go on to name the structural features of those entities, that is their attributes and relationships.

4.1.1 Entities

How to begin? You may start by listing the terms used in the business, and consider creating an entity class for each term. Some people say to list the nouns written down in requirements statements. But this is rather trite as an analysis technique. More helpful techniques are suggested below.

4.1.1.1 Ask about process control requirements

Consider the system as a process control system and name the things the users want to monitor, if not control. E.g. In an order processing system, the users want to stimulate their customers to place orders and to pay for them, to control the creation and completion of orders, and to monitor the stock level of each product type.

Business process modeling can help here, though the data analysis is more to do with considering what it is in the real world that the business seeks to influence, rather than how this influence will be exercised.

4.1.1.2 Look for business keys

Look for the things that the business already assigns keys to. Business people only give identifying numbers and codes to things they care about and want to keep track of over time. The key of an entity state record is not just a database concept; it is a necessary business concept.

> "This point sometimes seems to be lost on object-oriented folks." Michael Zimmer

A key enables users to:

- distinguish an object from another of the same class and
- map an entity state record onto a real-world object in the business environment.

So, you may reasonably start by looking for identifying numbers and codes that are important in the

business - customer numbers, product codes, order/invoice numbers and so on.

4.1.1.3 Specialise generalised abstractions

Analysts can prompt business people to think about their business by starting with some generalised super classes and asking about the specialisations that the business people are interested in.

> "Abstractions are useful in discussions with users. Abstractions force users to think more broadly about their business, and can be an aid to reengineering the business." Michael Zimmer

David Hay tells me his enterprise models feature entity types that are general enough to be recognizable by people in every business he models.

David's entities	aka
Party	
Product Type	or Item Type, or Asset Type
Product	or Item or Asset
Activity Type	or Procedure
Activity	usually along with Work Order
Contract	or Order or Agreement

Most people recognise broad generalisations such as those. So you can use these to uncover the subtypes that are specific to the business domain you are working in. I like the alliteration of the P words in my scheme below.

Graham's "P" entities	David's entities	David Hay's comments
Party	Party	Party subtypes into Person and Organization.
Person	A subtype of Party	
Partnership	Contract	
Product	Product Type	And Product
Process and Event	Activity Type and Activity	I have to model these for some clients, but not others.
Place / Point In Space	Real Spatial Element	I show subtypes of this in a later chapter.
Point In Time / Date		Never an entity in my models. But all my intersect entity types have beginning and ending dates.

4.1.1.4 Trial and error

In practice, experts normally start by guessing a few major entities and naming them in boxes. They then use their expertise to ask the right analysis questions - the questions that will help them to refine their initial guess. I am interested in cataloguing the analysis questions that experts ask.

One of the messages of pattern-based modeling is: Don't worry too much about getting the entities right to begin with. Looking for patterns will help you to assure quality, and to correct whatever picture you draw to start with.

> "How does this relate to the sorts of patterns that others have published?" Michael Zimmer

I am talking about a different kind of pattern, useful for asking questions and making model transformations. Wait and see.

4.1.2 Predicates

Predicates are characteristics that define what an entity is. Close on the heels of naming an entity, you may extend the range of business terms by naming at least some of its predicates. E.g.

Predicates of the customer entity
Customer Number
Name
Country
Telephone Numbers

Predicates are an excellent, if rather data-oriented, way of looking at entities.

> "Bob Schmid, in his book *Entity modeling for Information Professionals* has an original way of describing entity modeling that I think is quite brilliant. Among other things, he discusses "Predicates" early as characteristics of a Class." David Hay

Predicates are structural features. Some object-oriented designers prefer to define an entity by its behavioral features, its services or operations, but I don't take that view until I get to discuss rules in behavioural models later.

It is usual to divide the predicates of an entity between attributes and relationships. An attribute becomes a relationship when the attribute is specified separately as an entity in its own right (e.g. country).

Entity	**Predicates**	**Attribute or Relationship**	**Related entity**
Customer	Customer Number	Attribute	
	Name	Attribute	
	Country	Relationship	Country
	Telephone Numbers	Relationship	Telephone Number

The distinction between attributes and relationships is a subtle one. A question that helps to make the distinction clear is the question of uniqueness. Do users care if two objects have the same value for a given attribute? Can they change one value without changing the other?

Users don't care if two customers have the same name; they can change the spelling of one name

without changing the other. So, “name” is only an attribute, not a relationship to a uniquely identifiable entity.

Users care rather more if two customers have the same country; they would not want to change the spelling of a country name without changing it everywhere it appears. So, “country” is more than an attribute, it is a relationship to a uniquely identifiable entity.

> “What about the situation where for one class there are multiple values of an attribute? The old first normal form issue.” David Hay

I treat first normal form as a policy rather than a rule. There are some cases where a repeating attribute (e.g. telephone numbers above) is reasonably regarded as contained within an entity, rather than turned into a child entity related to the parent entity. I will return to this in the paper “Aggregate entities”.

> “Terry Halpin’s ORM is clever in that it treats both entity types and data types as ‘objects’. An attribute in relational-speak becomes a relationship to a data type. Among other things, this means that you can relate an entity type to a data type, and if the datavalue type later turns into an entity type, the changes required to the model are minimal.” David Hay

I have explore this pattern and transformation in later chapters, but I usually stick to a more conventional view, closer what people expect a database schema to look like.

4.1.3 Typical attribute terms

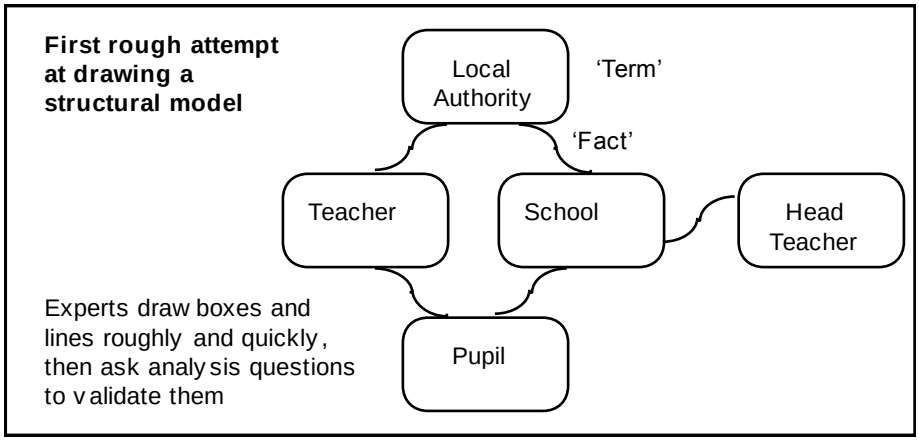
Attribute word	Notes
Description	text description
Memo	narrative, large text blocks
Name	well nigh an identifier, but no uniqueness constraint
Short Name	abbreviation
Number	what people call numbers, may include characters
Locator	map co-ordinates, postal address, email, phone number
Amount	usually currency, could be a Balance
Value	usually currency
Measure	quantity, size, length (not currency)
Sequence	
Date and time	
Date	
Time	
Indicator	short range of values: Boolean (true, false) and longer (yes, no, undecided)
Code	medium range designator: countries, colours, tastes…
Identifier	long range key: name, national insurance number
Image	picture
Video	moving image

Sound	voice, audio
Document	
Executable	

4.2 Structural facts

Every attribute or relationship is not just a term; it is also a structural fact. It specifies a relationship between one structural term (an attribute or relationship name) and another structural term (an entity name). Each attribute and relationship is potentially an entity in its own right, and you can view the name of an attribute or relationship as the name of a connection to this other potential entity.

Consider a system to record the pupils and teachers in schools run by local authorities. The boxes in the figure below represent terms used in the business. The lines reflect facts of life - reasons why terms are related in this business. Local Authorities employ Teachers; Pupils attend Schools, and so on.



You might initially represent the facts by connecting entities with wavy lines that don't specify constraints, but multiplicity constraints will press themselves upon your attention very quickly, and you should capture them as soon as you can.

4.2.1 Discovering terms and facts

Though it may be declaimed as heresy by some object-oriented purists, a very good way to discover entities, attributes and relationships is to use relational data analysis to divide relevant data structures (especially legacy databases and required outputs) into normalised relations.

> "Craig Larman, in his book on OO, has a half page discussing the idea of normalisation, without once using the term as far as I can see." Michael Zimmer

I'll say more about relational data analysis later.

4.3 Structural constraints

Terms and facts are fundamental. They come first. But you can't do much without the constraints; this is where all the useful stuff is.

A constraint is a business rule that limits the way entities are born, change state or die, or limits the values of attributes that are stored. A model with just six or seven entities and relationships might require the specification of 300 constraints.

> "I don't question that you have observed this, but I am surprised." Michael Zimmer

A member of the Business Rules Working Group reported those numbers to me. Don't forget that constraints include the data type of every data item, and every other precondition for valid processing.

Some constraints can be captured as rules governing the multiplicity of attributes and relationships. On discovering a fact that connects two terms you may immediately ask mathematical questions about the constraints that govern an object at one end of the relationship:

> Ask of each end of a relationship about its optionality - can the object exist without the relationship? And its multiplicity - how many objects at the other end can the object at this end be related to?

It is a pity that UML hides the optionality of a relationship inside the definition of multiplicity. So you have to look the far end of the relationship (right across the page sometimes) to see whether it is optional for that entity or not.

Remember technology-independence: the relationship lines in the model show facts about a business, they do not necessarily define how pointers are stored in objects or database records; this is a lower level of specification.

When drawing an entity model, you make no technology or implementation-level decision. You don't choose between different database management systems, decide which objects store pointers to other objects, choose between pointer chains or indexes, or choose between object-oriented and procedural programming languages.

Getting the semantic constraints on a relationship right is important. Relationships control how the system behaves and performs. The relationships not only constrain the behavior of the system. They also act as message passing or navigation routes between objects of different entities.

> "This seems to be the fundamental difference between a class diagram and an entity relationship diagram." Michael Zimmer

I have to disagree with you. The relationships in an entity model show the possible navigation routes between entities. These turn into message passing routes if you encapsulate the processing of each entity in the form of object-oriented style classes.

> "You weren't forceful enough in your reply to Michael. An entity/relationship diagram is technology independent. A relationship is structural. We don't care what kind of database structures or processes will be required to implement it.

A relational database implements relationships with foreign keys. (This is why IDEF1X is fundamentally a *design* technique.) Foreign keys are fundamentally structural. You "navigate" them with joins.

An object-oriented designer implements relationships with program code. A "behavior" for a class may include navigating an association to get information from another class. The navigation may be both directions, or always in the same direction." David Hay

I am not so anti-relational as that. For me, the foreign key is merely an alternative representation of a relationship, and I see no harm in that. I find the presence of foreign keys in an entity-attribute definition can help me to define some business rules in a concise way.

4.3.1 Primitive data types

Among the most basic constraints are primitive data types. Somebody, somewhere has to specify the data type of every data item in an enterprise application. How to do this in a technology-independent specification?

In the absence of an internationally-accepted standard, can we give analysts an instantly understandable shared language? Ignoring mathematical and complex number formats, people tend divide data items into five broad categories.

Data type	Perhaps applicable to attributes of this kind
String	Description, Memo, Name, Short Name, Alpha Number, Locator
Number	Amount, Value, Measure, Sequence
Date and/or time	
Label	Indicator, Code, Identifier, anything defined with a uniqueness constraint:
Complex object	Image, Video, Sound, Document, Executable

I don't mention primitive data types in the examples that follow. But they are important. And before I leave them, which level of model do we declare the primitive data types in?

Technology level? Yes. Primitive data types must appear at the technology level. Each implementation technology provides its own range of data types, or requires that you define them.

Enterprise application level? They do belong in the enterprise application model. You certainly have to define any user-defined data type (say country codes) and derivation rules. And you have to define any non-trivial display formats on outputs (e.g. a date might appear in several formats). For each model you build, you should have a list of the primitive data types. But in practice there are three reasons to exclude *primitive* data types from a model at this level.

- First, there is no internationally agreed standard.
- Second, they will in any case have to be translated into different data types for the given technology.
- Third, they are relatively trivial; you can surely trust intelligent educated developers to define them, and they have to be involved in system analysis at least to this extent.

Enterprise level? If the enterprise level truly is a model of real-world objects, then it might be argued

that data types do not belong there. Data types apply to entity state records rather than to real-world objects.

4.4 Structural derivation rules

A derivation rule derives data by some kind of calculation from other data. E.g.

Attribute of Invoice	Derivation rule
InvoiceNumber	= LastInvoiceNumber + 1
AmountDue	= AmountBanked + AmountRemaining
AmountBanked	= AmountDue - AmountRemaining
AmountRemaining	= AmountDue - AmountBanked

Which attributes are stored and which are derived? You can derive any one of the three 'amount' attributes from the other two. But you don't need to specify the derivation rule against all three attributes. Any one of them will do. By convention, specifying a derivation rule against one attribute (say AmountRemaining) implies the other two are stored attributes, not derived.

The illustration above might be part of an entity model, or it might be part of a database model. These different models need to be considered separately.

4.4.1 Storing derived data in a database

Some people insist that since derived data is redundant, it should never be stored in a database. But refusing to store derived data has led many systems designers into a wasteful excess of redundant processing. There is always a trade-off between update and enquiry efficiency.

In fact, financial institutions (banks, insurance companies and the like) do maintain a good deal of derived data in their databases. Or do you think your bank calculates your account balance every time that you request it, by working through all your transactions since you opened the account, adding all the credits and subtracting all the debits?

So, database designers may decide to store the AmountRemaining, or not. Some technologies allow them to specify this by declaring a derived data item to be 'actual' or 'virtual'. This is one way that behavioral processing operations have crept into the database paradigm.

> "Sign me up as in favor of representing derived data on the diagram. It is essential in explaining what's going on. I usually use typography (parentheses, or a leading /) to describe a derived attribute, and the derivation logic itself, of course, has to be documented behind the scenes. You correctly point out that to assert that an attribute is derived in the model says nothing about how that derivation should be implemented. Way back in the early 80's I used a wonderful dbms that had the concept of a derived field. This was the first time I had encountered this idea and it was wonderful. It made coding whole chunks of the business logic trivial. The only problem was, when we ran a query, the lights would dim. It turns out that in many cases, derived data should be derived when the data are input, not when the query is run. Oh, well." David Hay

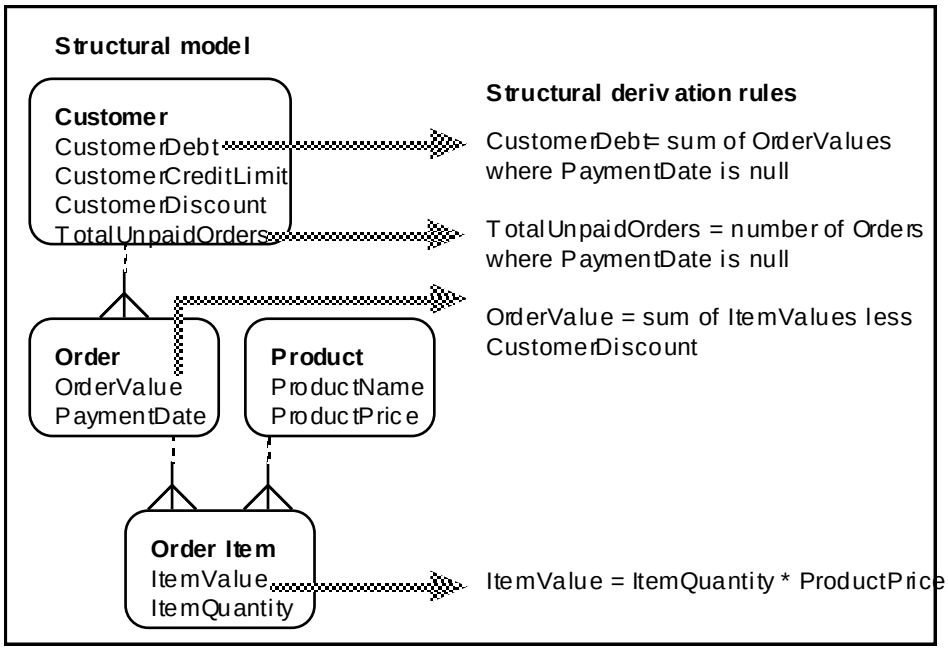
4.4.2 Separating specification from implementation

Derived data is important to users. Ideally, a business rules model will define all data that is important to users, whether it appears in the form of information displayed on a user's screen, or is used in testing constraints on input events.

The analysts job is to define the universe of discourse of system purchasers and users. At least some derived data and derivation rules are part of this universe of discourse, and naturally belong in a business rules model. Analysts should be able to name a derived data item as an attribute, and specify this attribute in the form a derivation rule, without deciding whether the attribute will be stored and updated in a database, or derived when needed for an enquiry.

E.g. A Business Rules specification should include the AmountRemaining, but need not say whether it will be stored as a data item, or derived when needed by calculation.

The figure below shows a fragment of an order processing system's specification. It shows that four of the attributes are derived by calculation from other attributes in the model.



This diagram presents four calculations as structural derivation rules, defining them as properties of attributes in the entities of the entity model.

In fact, the specification is incorrect. The four supposedly structural rules are not applied on the Order Registration event, nor or on the Order Item Addition event that adds an Order Item to an Order. The business would regard it as a mistake if the rules were applied at these times. The four derivation rules in our example are only fired by an Order Closure event, and only guaranteed to be true just after that event has been completely processed. So these derivation rules belong in a behavior model. See the companion book <The event modeler>.

4.5 Some questions and answers

4.5.1 Do we build our entity model for software engineers to use?

Yes. I don't really mind what conceptual models people draw during analysis, and/or for communication with users. It is the hand over to design I worry about. I see too many analysts drawing models for users that have to be completely rebuilt by the designers - and some analysts never realise that.

> "I recognize this problem. It's a two-way street. Yes, we modelers should work with designers to make sure that they understand the intentions and implications of the models. But the designers could but more energy into understanding the models as well.
>
> I rarely have problems [with users]. It seems to be the developers who have the most trouble understanding the kind of abstraction that is a model." David Hay

I am all in favour of discussing models with users. I am wholly against expecting users to validate a model. They are not equipped and don¹t attempt to understand an entity model in the way that designers and developers do. Developers worry about the implications for design and code, and that makes them (reasonably in my view) a tougher audience.

> "Ah, but the point is that a conceptual model is not supposed to be concerned with whether it can be implemented. It is supposed simply to describe the nature of what is." David Hay

That's OK if you model an enterprise per se. I want my models to be used by software engineers coding enterprise applications. For me, a conceptual model is supposed to be logical (technology independent) but it is also supposed to capture specific requirements and, in the end, be codable as the basis of a system that meets those requirements.

4.5.2 Do we define attributes, or hide them behind operations?

A few object-oriented purists do not define attributes, define only operations. They say 'encapsulation' means the attributes should be hidden behind the operations. This is plain silly for most enterprise applications. Do define the attributes.

When you name a boring attribute like CustomerAddress and define it as freely updatable, this is a short-hand way of saying that a CustomerAddress value can be retrieved by an enquiry operation from a Customer object, and overwritten with a new value by an update operation on that object. Spelling out such trivial operations in a specification would be tedious and unhelpful.

4.5.3 Do we store derived attributes as persistent data?

That is a physical design decision. When you name an attribute like AccountBalance and define its derivation rule (say, AccountBalance = Credits - Debits) you are not saying whether AccountBalance will be stored in a database or derived by an enquiry operation. Whether the business fact is implemented in the form of data or process is a design decision.

4.5.4 Can we build an entity model without a behavioral model?

You can, but it really is better to consider both data and processes in parallel. Even if you don't care to document behavior, an entity model does imply behavior. The attributes in the entity model represent the state data (local variables) of long-running business processes.

> "I understand this from your volume "The Event Modeler", but it is probably not commonly known." Michael Zimmer

These long-running processes can be represented in the form of state machines. And the relationships in the entity model declare which of these state machines are able to locate and talk to each other.

4.5.5 Do we build the entity model before the behavioral model?

Not necessarily. You might start with some behavioral analysis that identifies the use cases and the events or transactions to be processed. You might even specify the behavioral operations of an entity before its structural attributes. I have done this on process control system examples.

But for enterprise applications, defining the entity model first is natural. A typical enterprise application maintains a large data structure, and most of the operations merely store or retrieve the values of variables in that data structure.

4.5.6 Does an entity model imply a database?

Not necessarily. I have drawn entity models (using patterns in this book) for process control system specification, where the state is merely a few variables stored in memory. But for enterprise applications, the entity model does usually imply a database.

- A database becomes necessary where there are so many parallel-running state machines you cannot hold all their state data in main store.
- And a database is practical where almost all state machines are 'asleep' almost all of the time, so most of the data is inactive.

These two conditions pretty much define an enterprise application.

> "I see this relates to other papers where you discuss the essential differences between enterprise applications and the embedded systems often used as case studies in the object-oriented world." Michael Zimmer

4.5.7 Is an entity model exactly the same as a data model?

No. Sometimes the 'right' entity model corresponds to a relational database structure; other times it differs. The entity model of a business services layer is not 'merely' a data model. It is the structure against which behavior is specified, and operations are coded, just as any object-oriented designer would expect.

4.5.8 Does an entity model allow denormalization?

Yes, in two ways. An entity model allows division of one entity into smaller parallel aspect or role entities.

> "This would surely make relational purists have a fit." Michael Zimmer

Probably. But this kind of denormalization can be useful where an entity has parallel state machines or life histories. And it is essential where stored data is distributed, perhaps as a result of component-based development.

And an entity model also allows aggregation of child entities with a parent entity into an aggregate entity. However, this kind of denormalization is perhaps not as common as you might expect from reading books on object-oriented design, for reasons explored in the later chapter called <Aggregate entities>.

5. From data model to entity model

An enterprise application is a software system that records the state of entities in the business. The entities in an enterprise application tend to differ from entities in other kinds of application in two ways:

- there are thousands or millions of them, and
- they persist while the computer is switched off.

For these reasons, the state of enterprise application entities is usually stored in a database. And these reasons do influence they way you draw entity models. I start here with a traditional database model.

Some object-oriented designers are uncomfortable with the database, look on it as “the crazy aunt in the attic”. But it is fundamental and they neglect it at their peril. You can always reverse engineer an entity model from a database schema. This chapter starts with a physical database structure and explores ways to represent the business rules in the more conceptual model of classes and relationships that I call an entity model.

Terms and facts appear in software specifications as the names of entities, attributes and relationships. You can document terms and facts on their own, outside the context of a model in which constraints and derivation rules are also documented.

> “I would prepare a glossary of business language, even if some of the terms and facts were not part of the model, just because the client will use them in discussions.” Michael Zimmer

But facts tend to disappear, because as soon as you start to draw an entity model, you merge facts with constraints into the form of relationships. **Constraints and derivations** appear in software specifications as invariant conditions and procedures attached to entities, attributes and relationships.

> “Ah ha! As I generalize my models, I remove some business rules. While that sounds dramatic it in fact is not.
>
> As you have observed, three categories of business rules are **terms, facts, and derivation rules**. Those stay very nicely in my models.
>
> I have specifically excluded the **constraints** (except for multiplicity, of course). First of all, these are entity models we are creating that describe what can be done. It is not appropriate for them to also try to describe what may or may not be done. That is a different kind of model. Ron Ross tried to lay constraints on top of entity models with his notation, which demonstrated this point, even though it's a terrible notation.” David Hay

Here lie some differences between us. My concern is application-specific entity models rather than generalized models. I want my entity models to specify as many business rules as they can bear. I leave it up to the designers whether these rules are coded in programs or built into a database structure. Second, I don’t believe you! You don’t remove every constraint. Every one-to-many relationship constrains entities at the many end to be related to no more than one entity at the one end.

I believe it is true that the lack of a distinct behavioral model has forced Ron Ross into notation overload. I'm not sure Terry Halpin's Object Role Modeling (ORM) escapes this criticism entirely.

> “Object Role Modeling does model many constraints very well. It is a completely different approach to modeling, however, and I haven't had enough experience with it to know how to relate my patterns to it. I don't mind leaving constraints out. These are the things that do change a lot, so I am happy to model them as a separate exercise. I also look forward to the day when there are tools that let us model them (and change the models) separately from the database design effort.” David Hay

I am concerned to model constraints. I propose people should capture invariant constraints in the entity model and capture transient constraints in the behavioral model.

> You make the point that it is shaky linking business rules to data, since data change. That is the basis for your argument that they should be linked to behavior. But if you follow my philosophy, the data structures won't change. While rules may change, the kinds of things they refer to won't.

> Most businesses I enter are in an environment of total chaos. When I can show them the relatively simple structures that underlie their business, it gives us all an opportunity to examine what is truly unique about the organization and address it systematically. Every model I create is unique, just for that client. It is only that every model starts with the same underlying structure. By giving them an understanding of what is fundamental, I give them the ability to be truly creative with what is unique.” David Hay

Again, and it is worth repeating to be sure all readers know what I am talking about. My concern is application-specific entity models, where you have to define the business rules somehow. I am using an entity model to specify constraints. I am happy to leave the database designer to decide whether these constraints will be implemented in the database structure, or coded by programmers. One way or another, constraints do have to be specified and implemented.

5.1 A data storage structure

There are various ways to specify constraints by annotations on a data storage structure. I can use a case study to illustrate some ideas in an informal way. I will revisit the concepts and notations in more detail in later chapters.

Figure 4a shows how tables are connected in a relational database structure. A common convention is to underline the **primary key** for a table, that is, the unique identifier used to distinguish one row in the table (one object of the class) from any other.

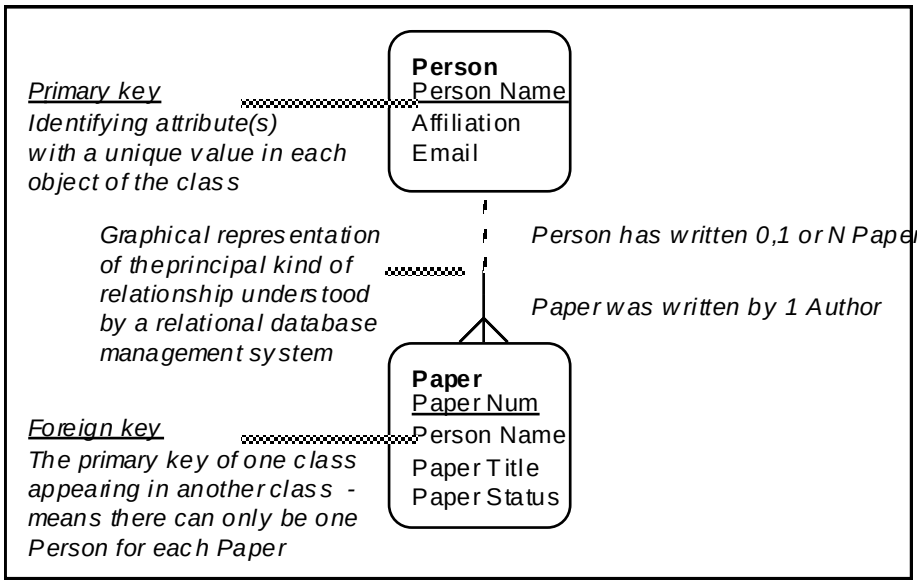


Fig. 4a

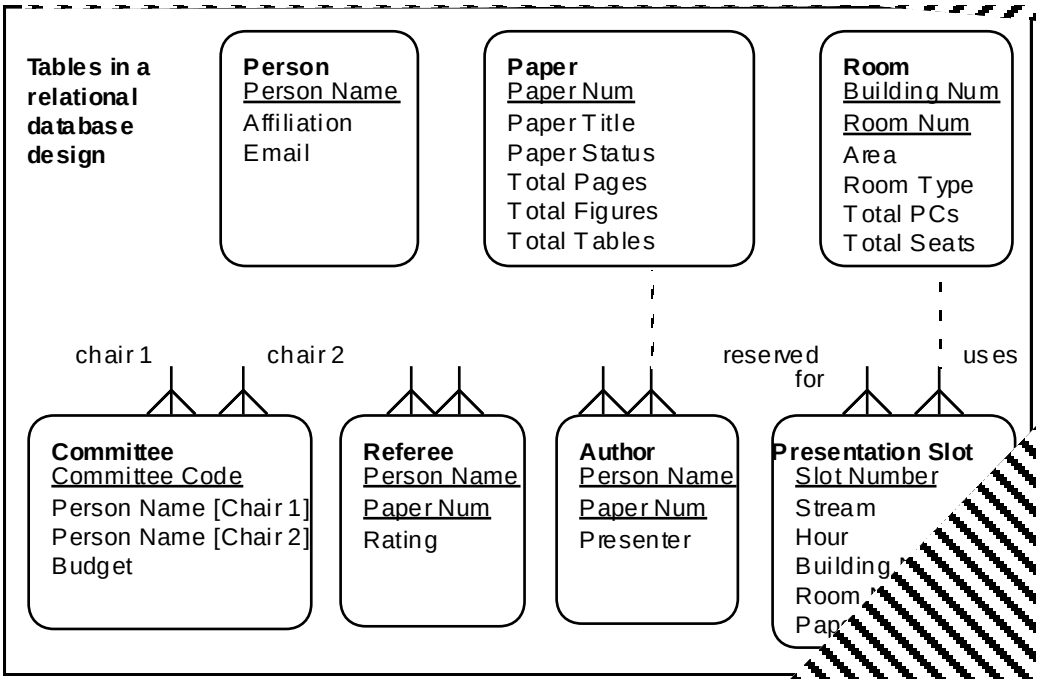
Where the primary key appears in another table, it is called a **foreign key**. A foreign key imposes a **uniqueness constraint** on a table; it says an entry in that table cannot be related to more than one entry in another table (where the foreign key appears as a primary key).

You can think of a foreign key, or any attribute that is not a primary key, as a rolled-up relationship. When you list attributes inside a table, you are saying that each attribute has a 1:1 relationship to the table.

In small examples, you can list the keys and other attributes inside boxes on the diagram. This is impractical where there are hundreds of tables. So most CASE tools provide a documentation scheme to back up the model and help you retrieve the attributes behind a box when you want them.

5.1.1 Relational database tables

I have stolen a case study from Halpin [1995]. Figure 4b shows the tables in a relational database and the relationships between them that are implied by foreign keys.



It is reasonably clear that a single entry in one of these tables (one object of one of these entities) will map onto a thing in the real world that users of the system will understand: a Person, a Paper, a Referee and so on.

Similarly, the value for an attribute in a table will represent a fact about a thing in the real world: a Person's email number for example.

Some people view each database table as a business entity class, and each row or entry in a table as the state of a business entity object. But a relational database design is not a full conceptual model. There are many business rules missing from the structure of tables shown above.

5.2 Constraints in attribute specification

5.2.1 Constraints on attribute values

The Referee and Author tables both contain an attribute that has only a short, fixed range of values. Figure 4c illustrates a convention that shows the range of values within {} brackets.

Author
Person Name
Paper Num
Presenter {Y,N}
Referee
Person Name

Paper Num
Rating {1…10}

Fig. 4c

Where the range of values for an attributes may include 'null' because the fact is missing or unknown, some say the attribute is optional (rather than mandatory), some say it is partial (rather than total). Figure 4d brackets the optional attributes in the Person and Referee tables.

Person	
Person Name	
Affiliation	
o--	Email
Referee	
Person Name	
Paper Num	
o--	Rating {1…10}

Fig. 4d

In practice, I am lazy about distinguishing mandatory and optional attributes; so you won't see the '**o**' symbol where it would be appropriate in every one of our examples.

How to show in the Paper table that three statistical attributes (the total number of pages, figures and tables) are only recorded for papers that are accepted? Figure 4e places an IF condition against the optional list of attributes.

Paper	
Paper Num	
Paper Title	
Paper Status {approved, undecided, accepted}	
Selection	*Optional attribute group*
o-- If paper selected	Total pages
	Total figures
	Total tables

Fig. 4e

The table brackets the group to show that if one total exists, then they all exist.

5.2.2 Constraints between different attributes' existence

Figure 4j introduces a three-way optional structure into the list of attributes in the Room table, and employs an IF, IF, ELSE structure.

Paper
Building Num
Room Num

Area	
Room type [lab, lec, office}	
Selection	*Mutually exclusive attributes*
o-- If Room type = lab	Total PCs
o-- If Room type = lec	Total seats
o-- Else	

Fig. 4f

The table brackets the mutually exclusive options, reflecting the three subtypes recorded as values in the Room Type attribute.

5.2.3 Attribute roles

The Committee table contains an attribute which appears twice playing different roles. Figure 4g shows the two role names in square brackets. It also shows the second role is optional.

Committee	
Committee Code {Org, Prog}	
Person Name [Chair 1]	
o--	Person Name [Chair 2]
o--	Budget

Fig. 4g

Since the primary key has only two values (Organisation and Programme), there can be only two entries in this table, two objects of this class.

5.2.4 Constraints between attribute values

How to show the rule that the same person cannot be both chair 1 and chair 2 of the same Committee? Figure 4h specifies the constraint as a statement against the second of the two attributes.

Committee		
Committee Code {Org, Prog}		
Person Name [Chair 1]		
o--	Person Name [Chair 2]	Not = Person Name [Chair 1]
o--	Budget	

Fig. 4h

5.3 Constraints in relationship specification

So far, all the relationships have been drawn as one-to-many. How to show that a Paper can have no more than one Presentation Slot? Figure 4i shows this by taking the fork off the relationship line.

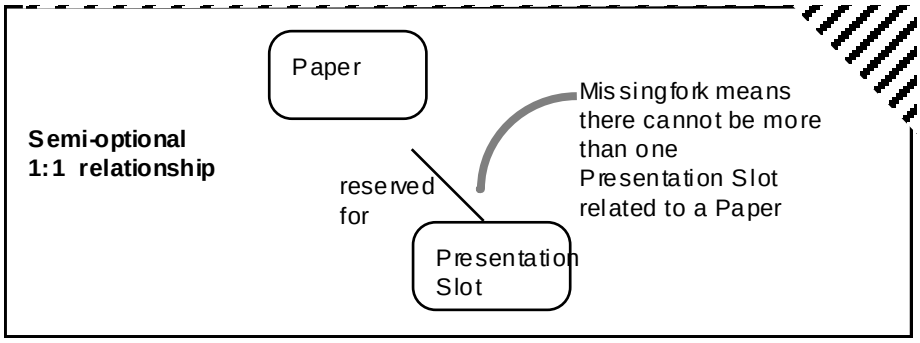


Fig. 4i

Figure 4j shows the same kind of semi-optional 1:1 relationship can be used to model the situation where the optional component is a subclass (rather than an different kind of thing connected in an aggregate).

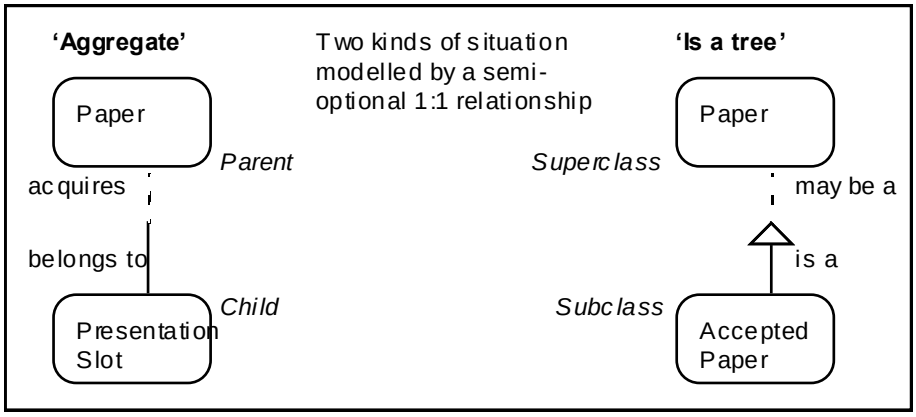


Fig. 4j

The difference between an aggregate of different things and a hierarchy of subclasses is not always obvious. Later chapters explore the difference further.

The triangle symbols here are only for the human reader. The relational database designer might implement these is-a relationships using 'foreign keys' in the normal way.

5.3.1 Introducing subclasses to impose constraints

The yes-no attribute in Author masks the fact that there are really three reasons why a Person may be related to a Paper. Figure 4k shows these many-to-many relationships as distinct boxes.

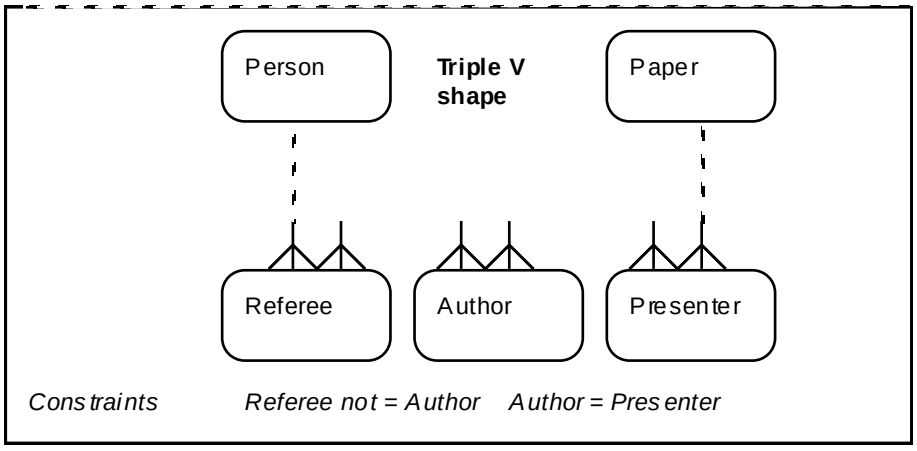


Fig. 4k

A multiple V shape (one of the patterns in the volume 'Patterns in entity modelling') prompts us to ask about constraints on the relationships between the child or link entities.

Figure 4l reshapes the model to show the constraint that a Presenter must be an Author, but an Author may not be a Presenter.

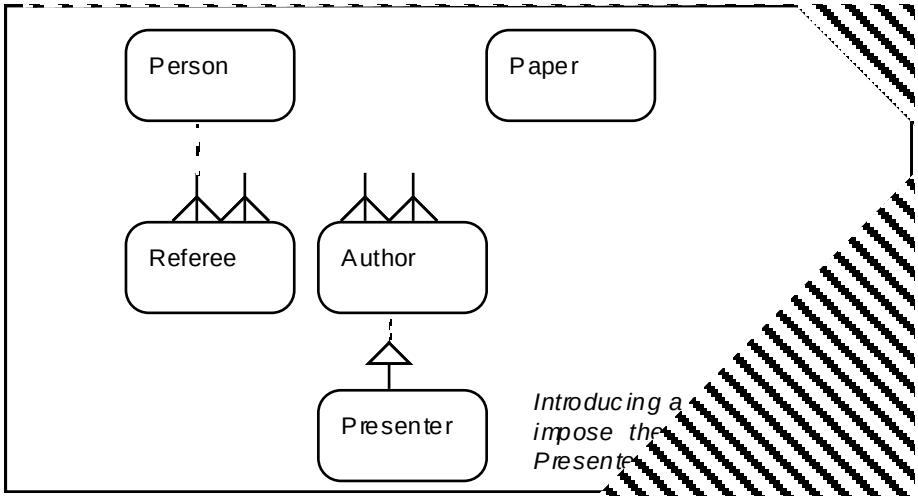


Fig. 4l

Figure 4m reshapes the model to show the constraint that a Presenter can only be related to a Paper that has been accepted.

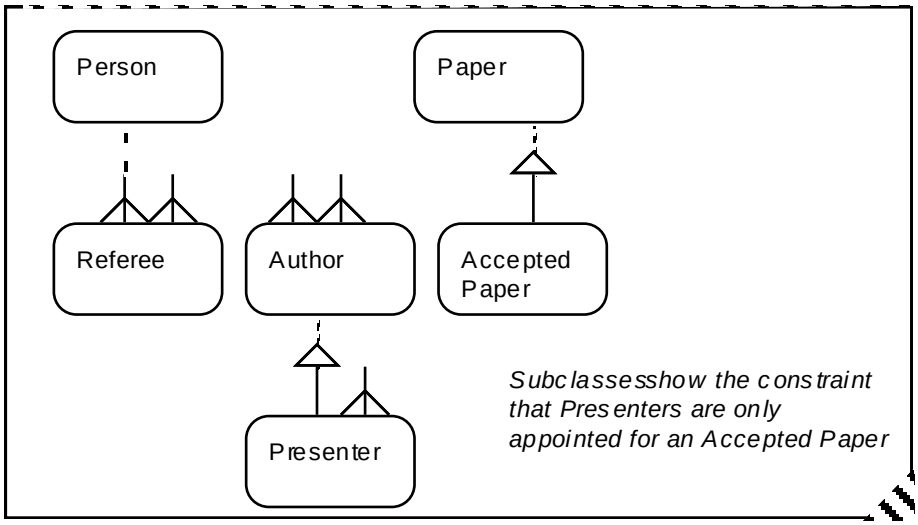


Fig. 4m

Figure 4n reshapes the model to replace the two subclasses by an optional relationship from Author to Paper.

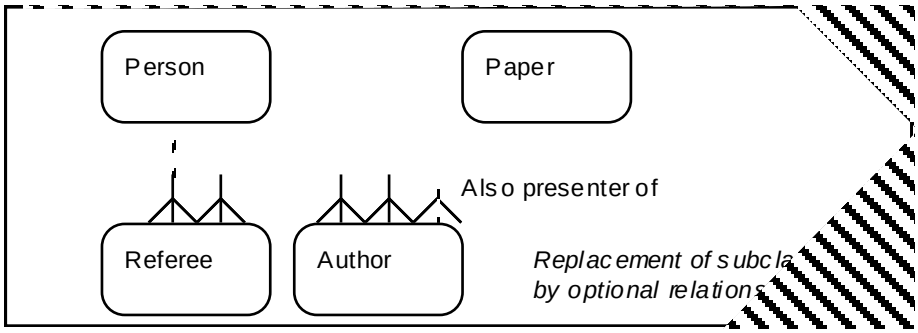


Fig. 4n

Figure 4n implies the primary key of Paper, Paper Num, will appear twice in Author, playing two different roles; the second role is optional since an author may not be selected to present their paper.

Author	
Person Name	
Paper Num [author]	
Paper Num [presenter]	Not = Paper Num [author]

Figure 4o shows the second Paper Num can be transformed into a yes-no attribute without loss of meaning.

Author
Person Name
Paper Num
Presenter {Y,N}

Fig. 4o

The yes-no attribute in the table I started with implied a second relationship from Author to Paper. The relationship may be obscure, but it does belong in a full conceptual model.

5.3.2 Conditional relationships

How to show that a Room cannot be used to present a Paper if it is an Office? How to show that a Paper cannot have any presenters until it has been accepted? Figure 4p shows both these constraints by writing IF statements on relationship lines in the diagram.

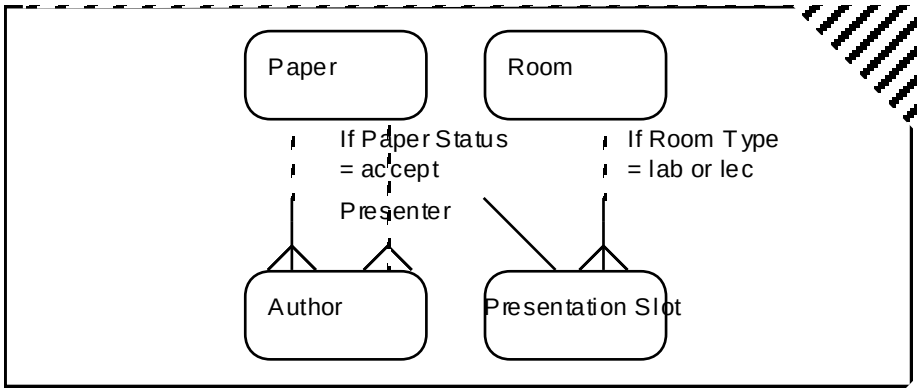


Fig. 4p

Rather than write IF statements in the boxes and on the relationships, you can introduce is a relationships into the data structure, then attach attributes or relationships that are specific to the subclasses to them rather than to superclasses where they are optional or mutually exclusive.

Figure 4q shows the is-a tree more graphically. So you can see more readily that only an Accepted paper can have a Presentation Slot or Presenters selected for it, and only a Presentation room (not an office) can be used for presentations.

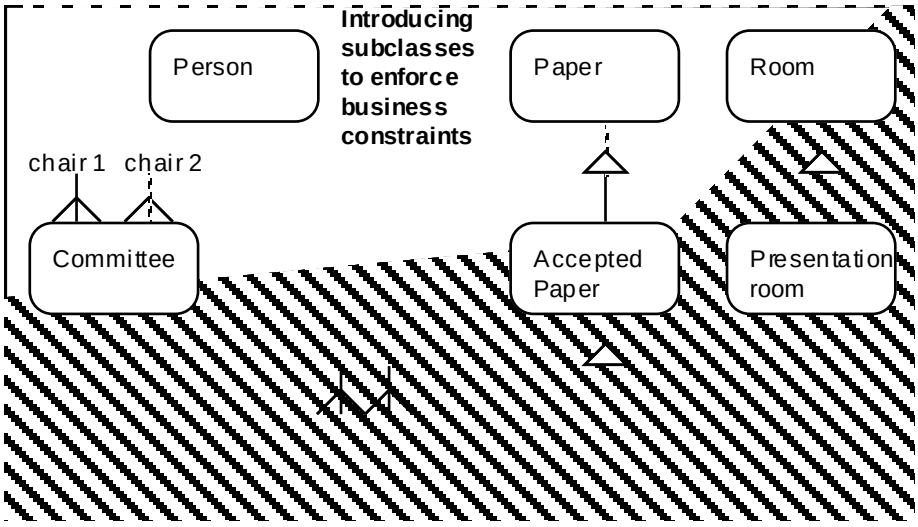


Fig. 4q

I do not recommend you introduce subclasses like this to define constraints on optional data, I am only

showing it is possible.

Remember also, for the moment, the triangle symbols are only for the human reader. The relational database designer might implement these is-a relationships using ‘foreign keys’ in the normal way.

Figure 4r shows how you might extend the original relational database structure to show enforce at least some of the business rules.

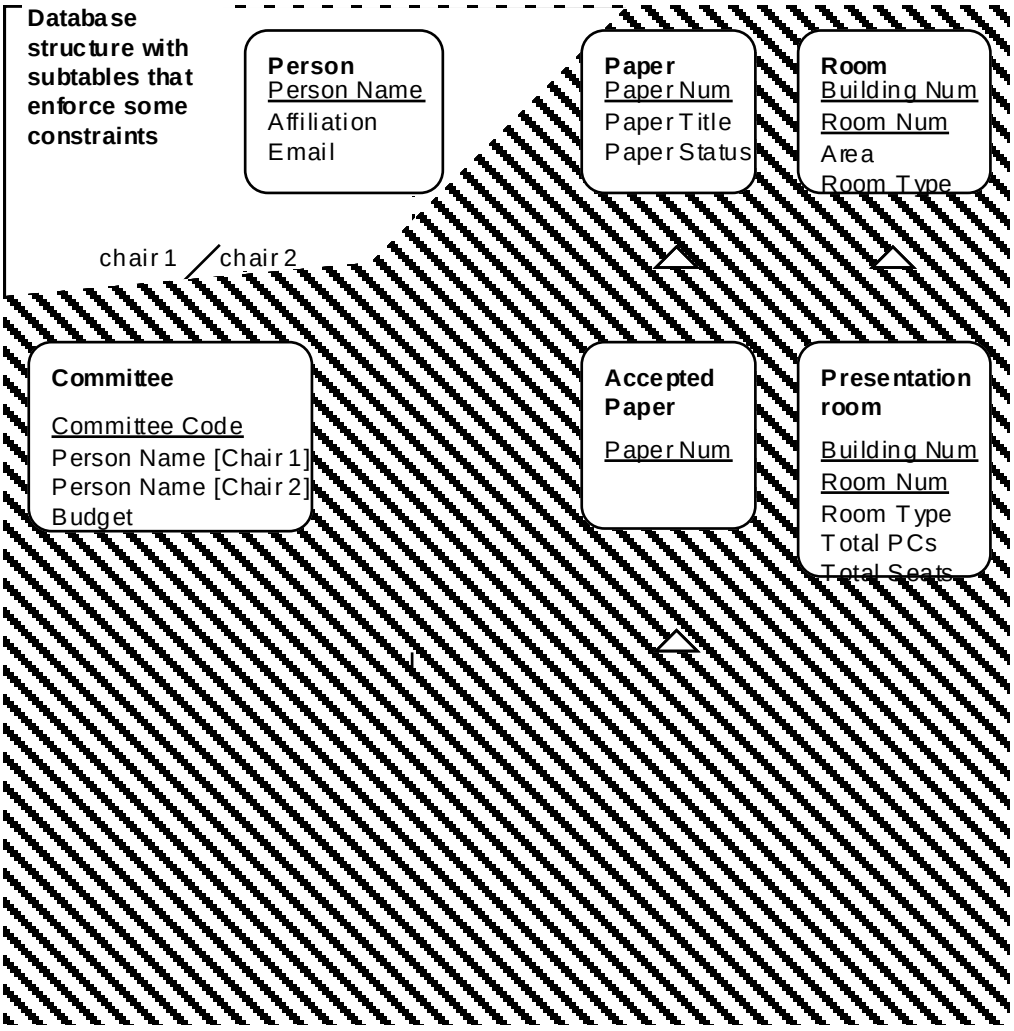


Fig. 4r

Figure 4r is not meant to be a definitive conceptual model for the case study. There is one more transformation worth illustrating before I leave this example.

5.4 Turning attributes into relationships

Starting from an entity model where entities have lots of attributes but a few relationships, you can turn all the attributes into parent entities.

You can relate each non-key attribute (other than primary keys) to a key-only parent of the original

entity. The key-only parent entity (otherwise known as an operational master entity, a collection entity, or a categorising entity, or perhaps a domain entity) stores the valid or actual range of values for the attribute.

Figure 4s shows the kind of diagram that might result from turning attributes into relationships. You may compare it with the diagram on page 381 of Halpin's book.

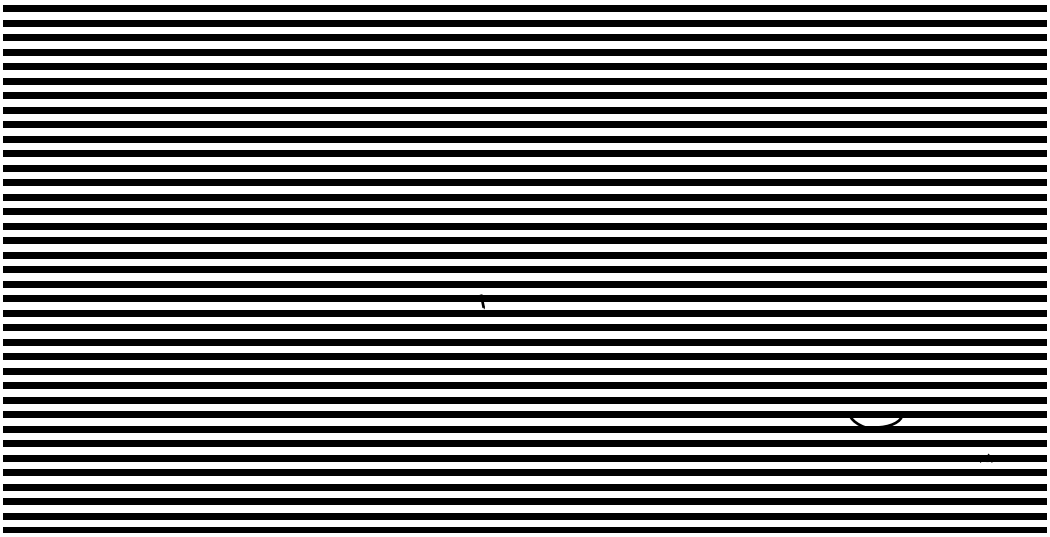


Fig. 4s

You need not show all non-key attributes as entities. However, you will want to raise the status of some attributes in this way.

Why and when should you do this? Chapter 5 provides some analysis questions. Briefly, you should consider whether the range of attribute values is controlled by users or by designers, and whether the attribute has properties of its own.

Some teach that a logical model must be simpler than a physical one. So it is worth noting is that the conceptual model above is a good deal more complex than the set of physical database tables I started with.

So where does the complexity get coded if it is not in the database structure? Probably in processing rules. You do need to understand how to capture business rules in process specification as well as in database specification.

I will revisit some of the ideas illustrated by Halpin's case study, and add many more.

6. PART TWO: ENTITY MODEL PATTERNS AND TRANSFORMATIONS

This part catalogues patterns in the relationships between entities. It reveals the rules of thumb and business analysis questions triggered by pattern recognition. It continues from where Part one finishes, starting on ground that is relatively firm and finishing with some more speculative suggestions.

7. Patterns in entity models

Using patterns and model transformations to get the constraints right.

This paper highlights the importance of the getting the constraints right. It introduces a catalogue of standard structural shapes and the notion of entity model transformations. These ideas are developed in later chapters.

7.1 Better design through pattern recognition

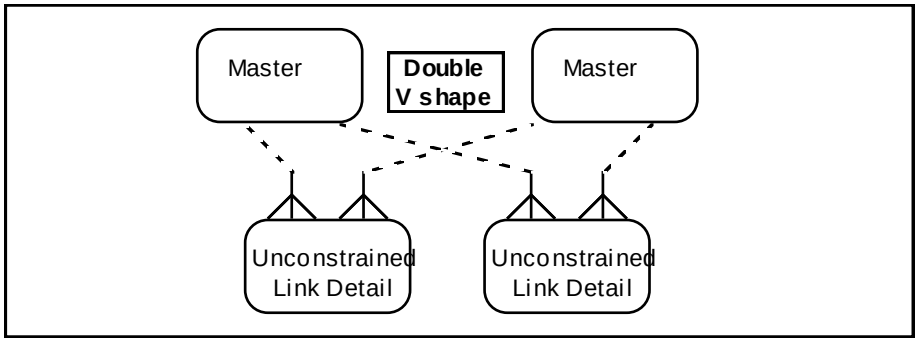
The success of a system depends on the relationships between entities being specified correctly. If they are not, then it becomes:

- easier for useless data to get into the system
- harder for programmers to locate the information they need to find.

To prevent the above difficulties from occurring, to sharpen up the act of analysts, and to save time and effort, you can apply some simple quality assurance techniques. One technique is enquiry access path analysis. This means defining the route by which required information is extracted from the model.

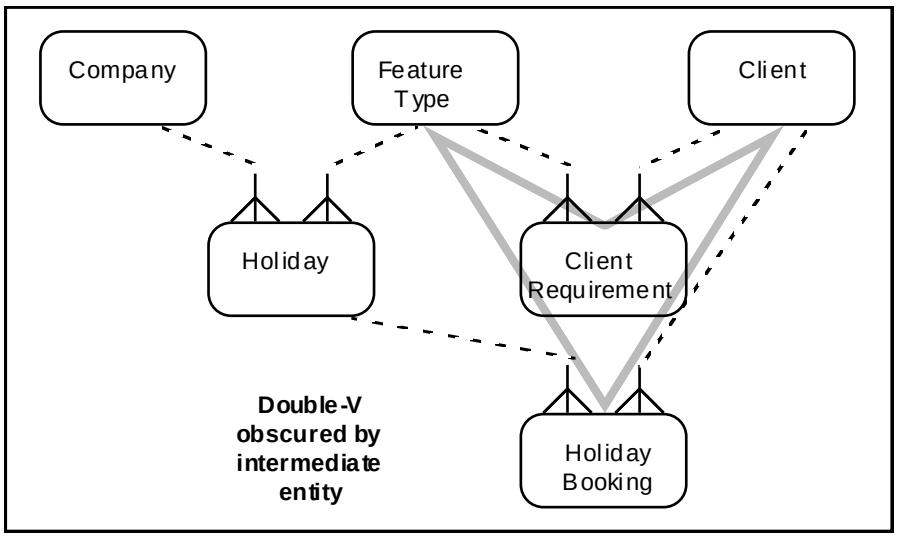
Another is pattern analysis, which has the advantage that it can be applied with less detailed knowledge of the required outputs. Fortunately, there are recognisable patterns and questions that lead you to transform an intuitive or poor design into a well-engineered design. These patterns help you to raise the quality of analysis and design work, and thereby improve the quality of the resulting systems.

The figure below shows a pattern called the double-V shape.



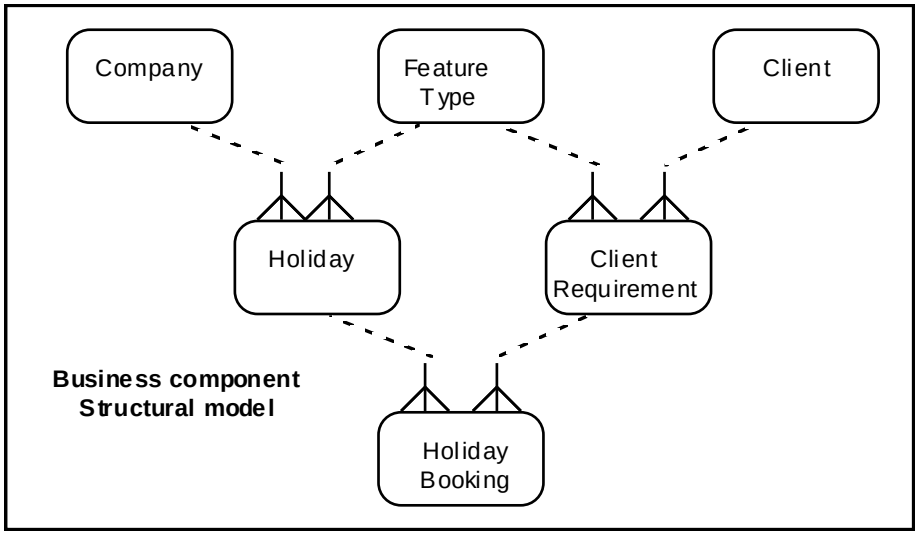
Ask of a double V shape: Can you tie an object of one detail entity to only one object of the other detail entity? If yes, connect the two detail entities by a relationship, to capture the constraint.

The basic pattern can be obscured by intermediate entities. The figure below includes a double-V shape, even though the Holiday entity sits in the middle of one side of one of the two V shapes.



Ask of this double V shape: Can you tie a Holiday Booking to only one Client Requirement?

Yes, a Holiday Booking is made to meet the Client Requirement for the Feature Type that classifies the Holiday. So Holiday Booking is a detail of Client Requirement rather than Client.



There is a quality benefit. It is now impossible for users to create a Holiday Booking for a Client who has not expressed an interest in the Feature Type of the Holiday.

There is a productivity benefit. Programmers do not have to navigate around the model to find the relevant Client Requirement for a Holiday Booking, or sort Holiday Bookings by Client Requirement within Client.

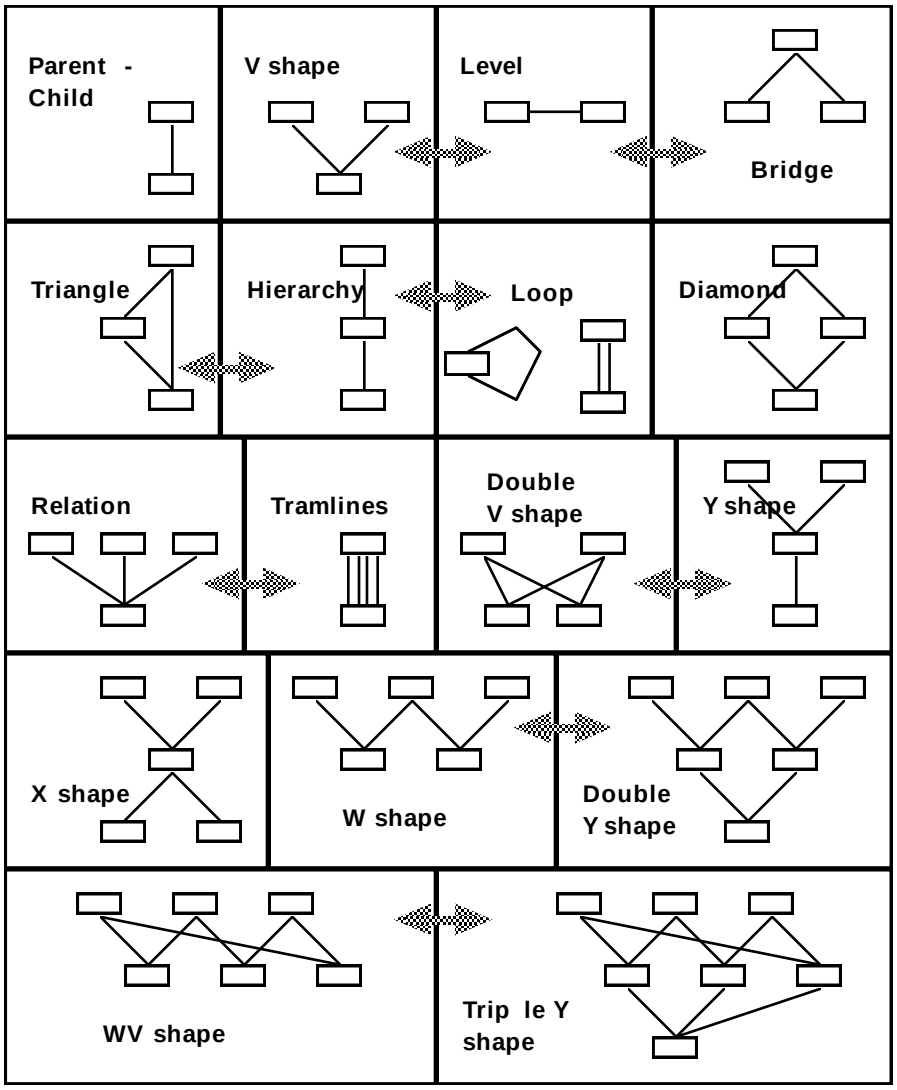
> “I wonder if normalisation could or should lead to this result? It seems somehow similar to the transitive dependency issue, but only at an intuitive level.” Michael Zimmer

Perhaps, but you cannot rely on any one technique to reveal everything. See the Chapters on <Model transformations> for further examples.

7.2 Entity model transformations

Patterns occur in various kinds of specification, but among the simplest and most widely useful are those that involve the specification of relationships between entities. The figure below is an attempt to summarise and name the entity model shapes that I am most interested in. The arrows show some of the possible transformations.

Shapes, and some possible transformations



Our pattern names include: parent-child, V, level, bridge, relation, diamond, triangle, double-V and Y shapes, double and triple Y shapes, tramlines, X shapes and recursive shapes. The Chapters on <Model transformations> shows the transformations in the first row apply to both data models and process models. Other Chapters (not yet collected into a volume) detail the shapes and transformations indicated by arrows on the diagram above.

7.3 A tool for raising quality and productivity

Many people teach the mechanics of how to document system specifications. Plenty of CASE tools help you with these mechanics; they ensure you get the syntax right; they constrain you to use the

proper boxes, symbols and lines. But there are no tools that help you with the semantics - the difficult part - the thinking - the analysis

The skill of professional analysts lies in recognising patterns in specifications, especially in object and event models. They use standard shapes constructively to build up a large and complex picture. They also use them destructively, to analyse an existing specification, to take it to pieces and question whether a better construction should be put upon these pieces.

The patterns catalogue above is a chart of simple shapes with memorable names. I have listed the questions you should ask about each shape, and the possible transformations that you might need to apply. So I can now teach students:

- the name, meaning and use of a pattern in constructing a specification
- the analysis questions which discovery of the pattern prompts
- the design or redesign work that is necessary depending on the answers.

This new approach means that for the first time I can envisage a CASE tool that helps us with the thinking part of analysis and specification. It will help us to build better quality systems, not just better documented systems.

7.4 Automating pattern recognition

A pattern on its own isn't much help. What to do with the pattern that has been recognised? This is the expert knowledge I want to capture. A tool can highlight or report on patterns, and prompt its user to answer specific quality assurance questions.

If a CASE tool is to ask us questions about patterns, it must first have the appropriate pattern recognition functions. To recognise the named patterns, the entity at one end of each relationship must be declared as the 'parent', and the other must be the 'child'. E.g. given a one-to-many relationship I always nominate the 'one' end to be the parent. It is the parent-child hierarchy inherent in each relationship that makes the shapes recognisable, whether by a person or by a tool.

For people, always drawing the parent above the child imposes a hierarchical structure that helps us to display the known patterns in an easily recognisable form, corresponding to the shapes in the analysis patterns catalogue.

> "My preference is to draw the model this way up. Dave Hay prefers the 'dead crow' notation - really a matter of taste." Michael Zimmer

Of course, patterns are careless of the diagram symbols or the presentation form. As long as each relationship has parent and child ends, a CASE tool can detect a pattern if the model is drawn upside-down, or using different symbols, or written down in the form of text or code.

Mike Burrows has developed a CASE tool called Validator (see <www.asplake.demon.co.uk) that detects and reports on most of the structural patterns I have catalogued. It asks you analysis questions, and suggests some transformations that may improve your model. See the Chapters on <Model transformations> for further details.

8. Nine simple model transformations

I earlier applied various transformations to a relational database design from Halpin. Some of the same transformations appear in a classification developed by Petia Wohed (neé Assenova) and Paul Johannesson at Stockholm University.

Petia and Paul set out with the intention of making schemas more graphical, to make the rules fully explicit for the purpose of schema integration. Schema integration is a different job from schema design, so I will add comments and guidance from the view point of somebody who designs schemas for enterprise applications.

Also, their modelling language is different from ours. Notable differences are listed below:

Their term	Our term
attribute	attribute or relationship (see below)
single-value attribute	attribute or 1:1 relationship
multi-value attribute	parent-child relationship (from one master object to many child objects)
partial or total	optional or mandatory
total in union	at least one must exist
surjective attribute	parent-child relationship with at least one child

8.1 From optional attribute group to subclass

Given an entity with an optional group of attributes, you may move the optional attribute group into a subclass where it is mandatory.

Petia and Paul discuss thus under ‘transforming partial attributes’. Figure 5a illustrates their example.

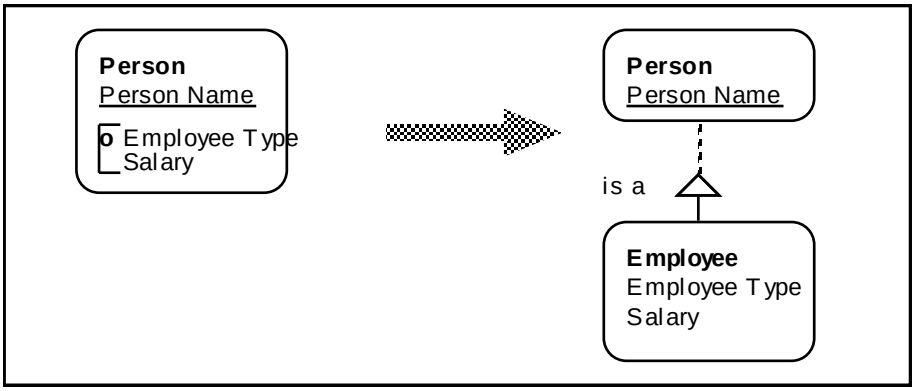


Fig. 5a

Figure 5b shows an example from chapter 4. The entity Paper has attributes that only apply to papers accepted for presentation. So you may move the optional attributes (Total Pages, Total

Figures, Total Tables) into a subclass where they are all mandatory.

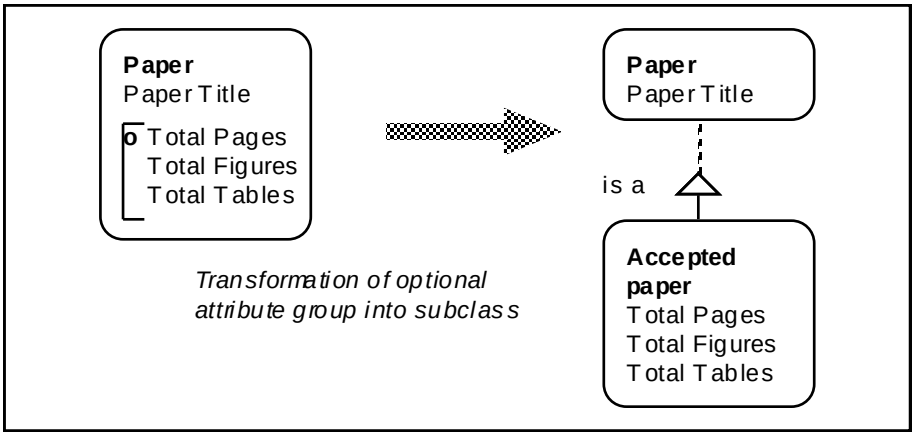


Fig. 5b

Chapter 6 suggests that people normally do the reverse in practical system design. They roll up optional data groups into an aggregate entity, partly to reduce the length and complexity of access paths by events and enquiries, and partly for other reasons explored in Later chapters.

8.2 From optional relationship to mandatory relationship

Given a parent-child relationship that is optional from the master's view point, you can make it mandatory by replacing the child by a subclass of itself.

P&P call this 'transforming non-surjective attributes'. An attribute is 'surjective' when each instance of its range (the master entity) is associated with at least one instance of its domain (the child entity). So a surjective attribute is an attribute whose inverse is a mandatory relationship.

Figure 5c illustrates their example.

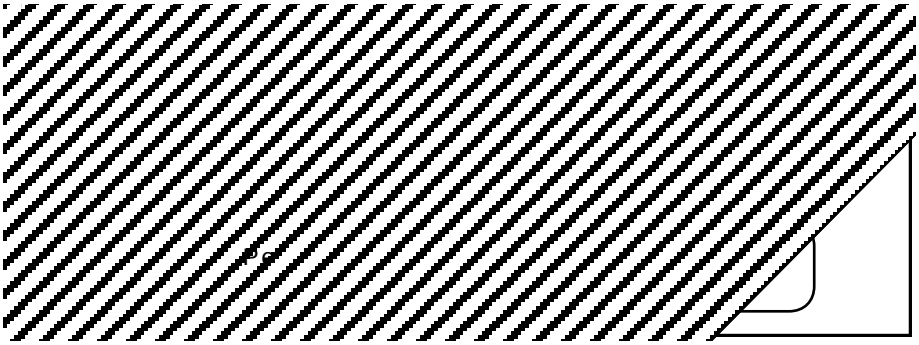


Fig. 5c

After the transformation, each object of the parent entity is associated with at least one object of the child entity. In this example, the relationship starts optional at both ends and becomes mandatory at both ends.

Again, people often do the reverse in practical system design. If a superclass has only one subclass, they would roll the data of the subclass up into the superclass, partly to reduce the

length and complexity of access paths by events and enquiries, and partly for other reasons explored in Later chapters.

8.3 From optional attributes where at least one must exist

Given optional attributes that are mutually exclusive, so at least one must exist, you can introduce a generalised attribute, a superclass of the mutually exclusive attributes.

P&P call this 'transforming partial attributes which are total in union'. Figure 5d illustrates their example.

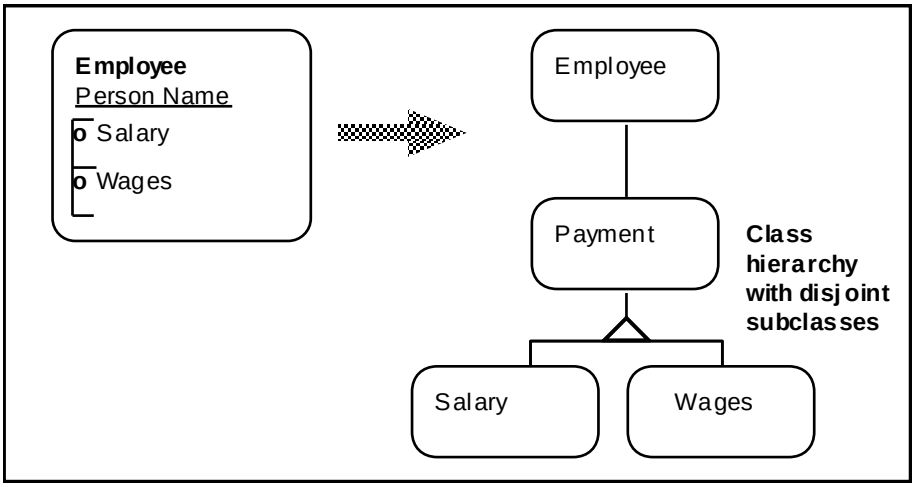


Fig. 5d

Figure 5e shows a different convention in database design - to turn the mutually exclusive attributes into mutually exclusive relationships.

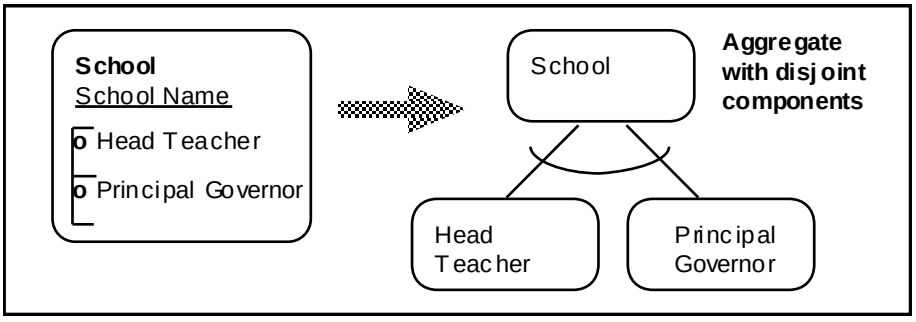


Fig. 5e

Later chapters explores the difference between a 'class hierarchy' as in figure 5d and an 'aggregate' as in figure 5e.

8.4 From optional relationships where at least one must exist

Given several parent-child relationships that are optional from the parent's view point, but

where at least one must exist, you can make them mandatory by introducing a superclass of the various children.

P&P call this ‘transforming non-surjective attributes which are total in union’. Figure 5f illustrates their example, where a Head Teacher is obliged to take responsibility for at least one course.

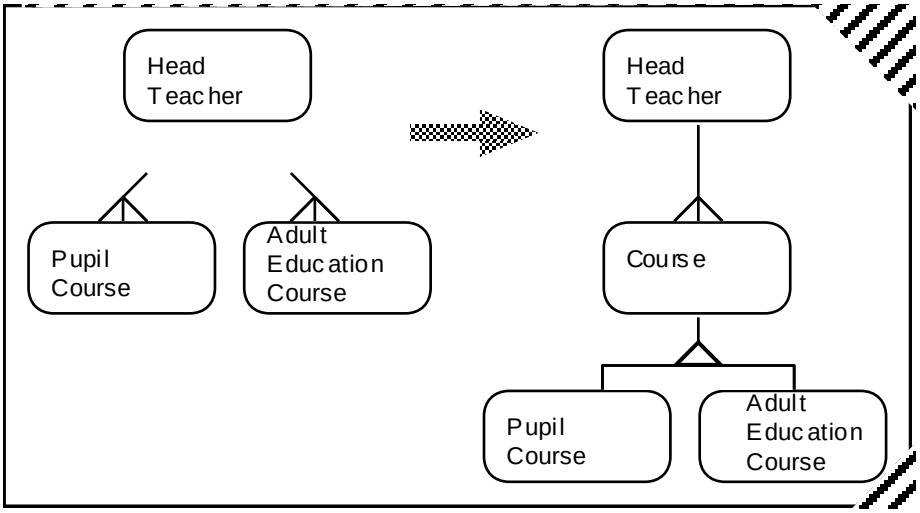


Fig. 5f

This transformation is unusual in practical enterprise application development. The requirement that a parent must have at least one child drawn from different types is not very common.

Designers are likely to apply the reverse transformation, that is, relax an ‘at least one’ constraint after it has been defined, because where a business monitors hundreds or thousands of objects, it is normally easy come up with counter examples, valid exceptions to the rule.

8.5 From N:N relationship to link entity

P&P call this ‘transforming m-m attributes’. Figure 5g illustrates their example.

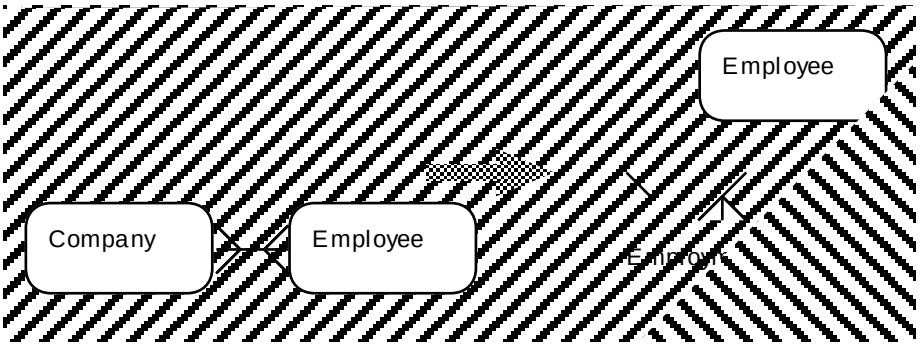


Fig. 5g

This transformation is very common in practical enterprise application development. So common that it is second nature to database system developers, not just because it is required for implementation reasons, but because resolving many-to-many relationships is a valuable step in analysis. See chapter 5 for further discussion.

8.6 From attribute to parent entity

Given an entity with non-key attributes, you can raise any attribute other than the primary key to become a parent entity connected by a 1:N relationship.

P&P call this 'transforming lexical attributes'. Figure 5h illustrates their example.

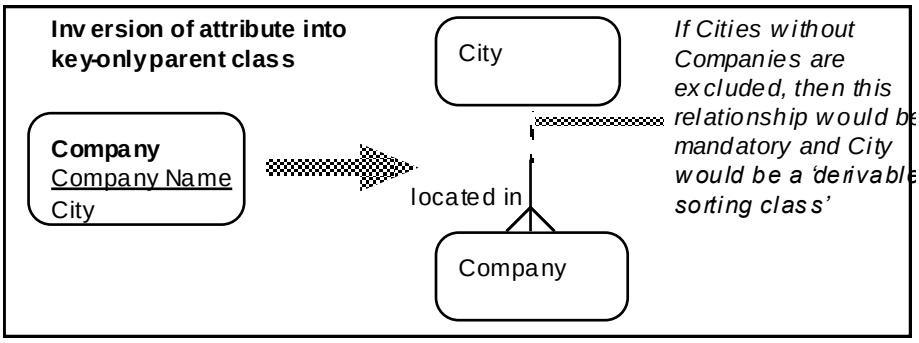


Fig. 5h

This is very common in practical enterprise application development. But why and when to do this?

> 'the schema is more stable, since it is easier to add extra attributes for cities. Some of the queries after the transformation become more complex, because the derivations cover a larger network of objects.' P&P

Let us focus on a tiny part of the model at the end of chapter 4. Figure 5i shows the non-key attributes of the Room entity raised to become key-only parent entities.

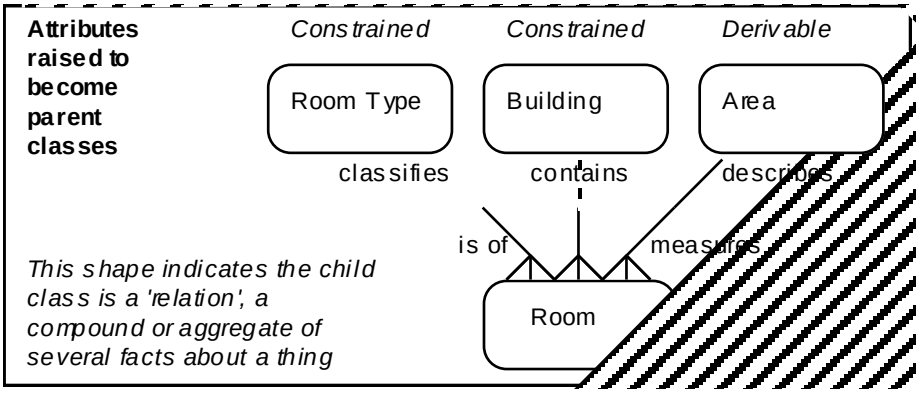


Fig. 5i

The best kind of analysis pattern prompts 'Ask of this pattern…' questions.

Ask of a non-key attribute: Is the range of values constrained? If yes, define the attribute as a parent entity.

E.g. Room Type is constrained (lab,lec,office) and Building is constrained (1...5). You can prevent mistaken classification of a Room under an invalid entity by defining these values as objects of a parent entity.

If no, nobody wants to control the range of values, then don't make it a parent entity. E.g. Say nobody cares too much about what is recorded as a Area. The Area entities are derivable from whatever values happen to be recorded.

Ask of a non-key attribute: Do users control the range of values? If yes, define the parent entity in the Business services layer.

E.g. Users might want to control the range of Room Types (lab,lec,office).

If no, or you want to stop users from change the system's rules by adding or deleting objects of the class, then define the parent entity in a layer of the design controlled by designers. E.g. you might define Building as a class in the UI layer or a table in the data storage structure.

Ask of a key-only parent entity: Does it have non-key attributes of its own? If yes, make it an entity like any other in the Business services layer.

E.g. you might record the total number of Rooms as an attribute of the Room Type entity. Even a derivable total like this turns the key-only parent entity into an entity like any other.

If no, then you may later treat the key-only parent entity differently from other entities in the data storage structure, perhaps define it as an index rather than a table.

Ask of any remaining non-key attributes: Do users regularly make enquiries that select or classify objects by a single value of the attribute? You may signify any further requirement for classification by drawing an entry-point arrow on the entity, showing which attribute is used for selection. Again, the attribute may become some kind of index in the data storage structure, rather than a table.

8.7 From fixed range attribute to class hierarchy

Given an entity with an attribute that has a small fixed range of values, you may transform the fixed range into distinct sub entities.

P&P call this 'transforming attributes with fixed ranges'. Figure 5j illustrates their example.

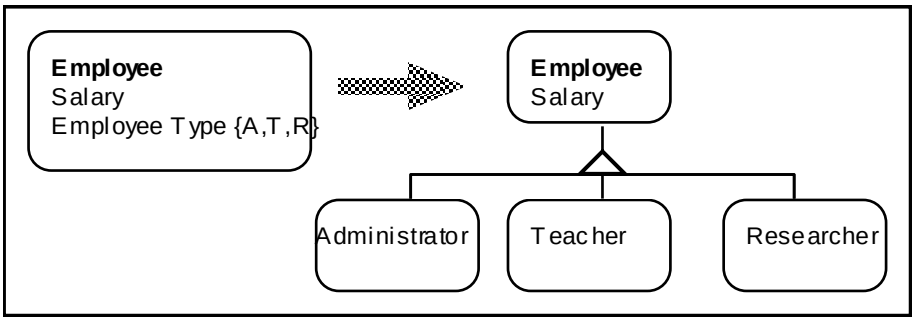


Fig. 5j

'It is easy to see the subclasses in the graphical notation. In contrast, it is more difficult to read and understand the definition of the lexical type Employee Type, which is not included in the graphical notation of the schema.' P&P

Given the attribute Room Type (lab, lecture room, office) in the case study in chapter 3, you may transform the fixed range into distinct subclasses.

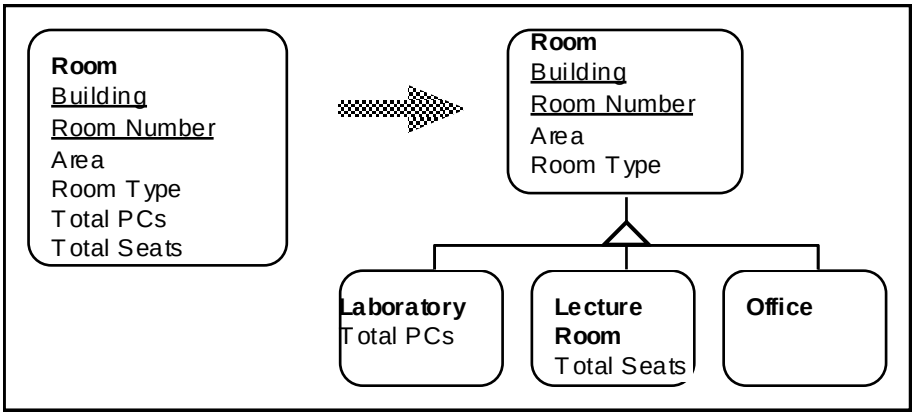


Fig. 5k

This transformation is rare in practical enterprise application development, for reasons explored in Later chapters.

8.8 From class hierarchy to network

Given a class hierarchy in which subclasses share properties in an orthogonal dimension, you can create a class network.

P&P call this 'transforming to lattice structures'. Figure 5l illustrates their example.

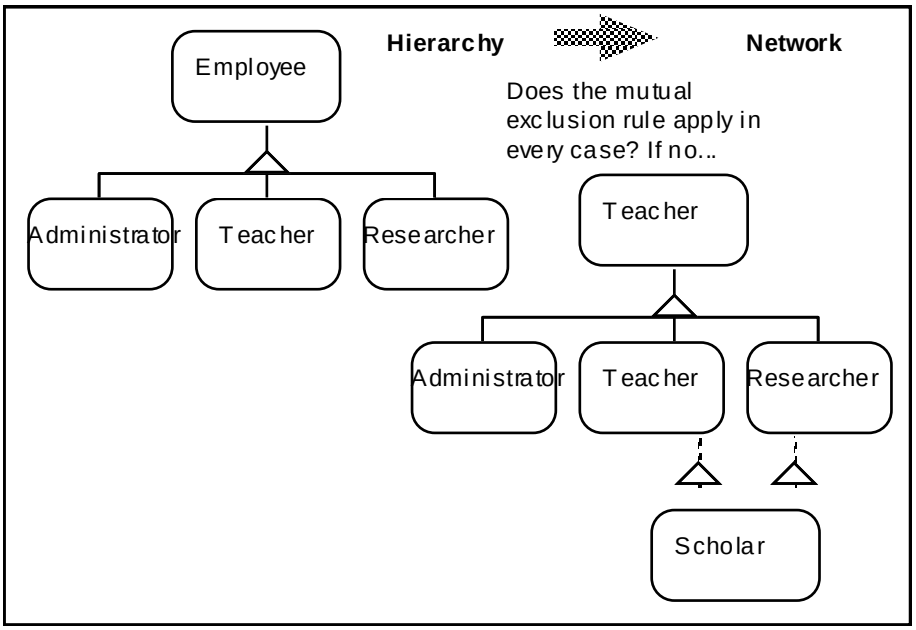


Fig. 5l

'It is easier to see that Scholar is specialisation of both Teacher and Researcher if they are drawn as boxes, compared to searching for and reading a rule that has the same meaning.' P&P

Later chapters explores this transformation in more child, but a little of the discussion is repeated below.

Figure 5m shows that a data structure in which Class Teacher and Head Teacher inherit from Teacher might be extended to include a subclass that inherits from both Class Teacher and Head Teacher.

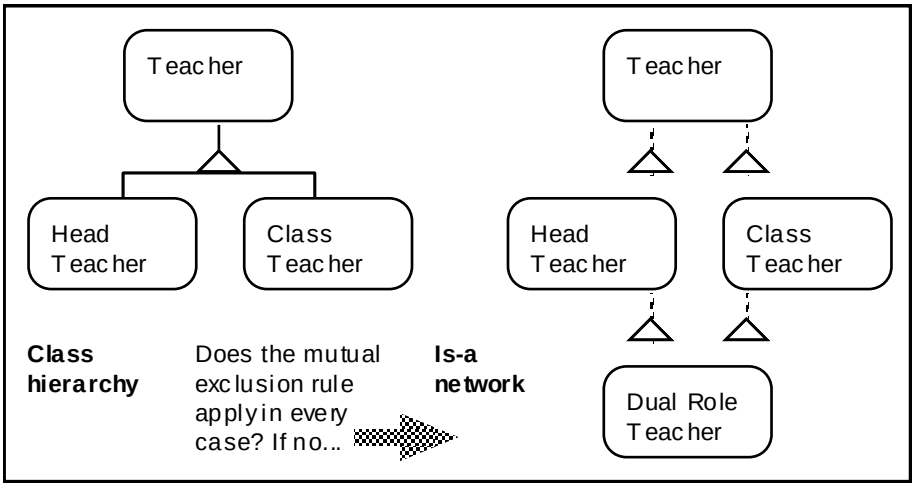


Fig. 5m

Figure 5m shows a diamond-shaped is-a tree in which a Dual Role Teacher entity has been

introduced to accommodate the few Teachers that are both Head Teacher and Class Teacher

- A Dual Role Teacher **is a** Head Teacher **is a** Teacher.
- A Dual Role Teacher **is a** Class Teacher **is a** Teacher.

Defining a diamond-shaped is-a tree may be a recognised practice in object-oriented languages that support multiple inheritance, but one should be aware that the meaning of the model is ambiguous, in the way described below.

The model does not specify the rule that a Dual Role Teacher is a single Teacher. It might equally well be read to imply that two Teacher objects are needed instantiate one Dual Role Teacher object.

One way or another, an object-oriented programming environment that allows multiple inheritance must work out that Dual Role Teacher inherits only once from Teacher. But the semantics of the diagram notation don't tell you this, and we want the conceptual model to act as a specification for relational database programmers as well as object-oriented programmers.

Diamond shaped structures are discussed further in Part Two.

8.9 Generalisation of similar attributes

Where an entity has a list of similar attributes, you can generalise these attributes into a relationship. P&P call this 'transforming non-unary attributes'. Figure 5n illustrates their example.

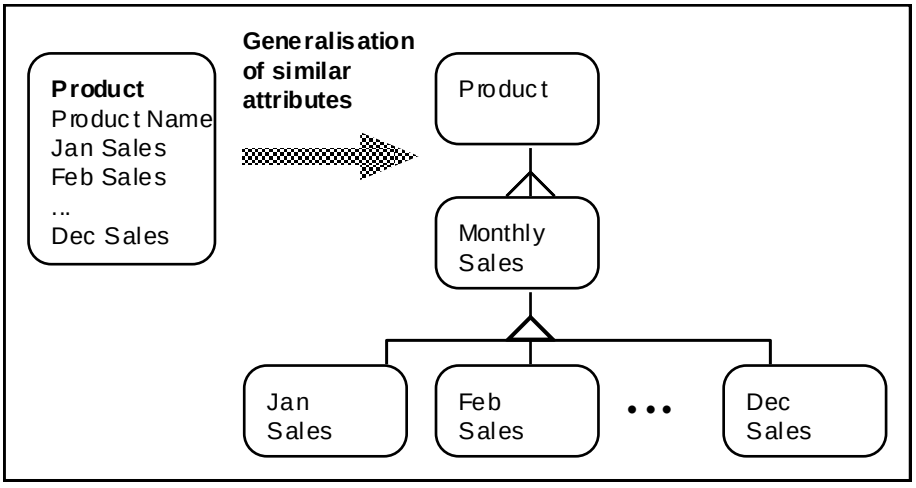


Fig. 5n

Once again, this transformation is not very common in practical system design. Figure 5o shows two more common transformations discussed in Later chapters.

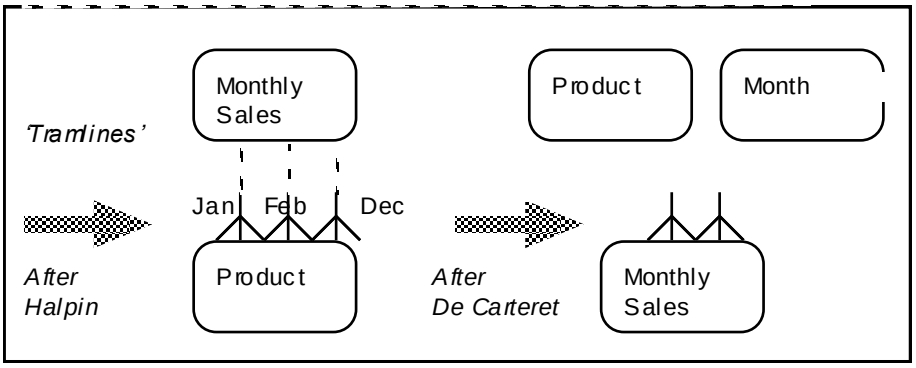


Fig. 5o

8.10 Different views of conceptual modelling

Schema integration v. schema design

Petia has commented as follows.

> 'All of our transformations increase the size of the schema (the diagram that is). Some don't like this. But it is the price you pay for a clear and explicit model of rules and constraints.
>
> 'People can specify constraints as rules attached to the attributes, hidden away in a data dictionary behind the model. But then it is harder to see concepts which may be important.
>
> 'The idea of conceptual modelling is to model the universe of discourse, capture its important aspects, in a graphical picture. So, our nine transformations make the presentation more explicit, more visible. The aim is to specify rules and constraints as relationship lines in a graphical model.'

Petia and Paul are interested in these transformations for the purpose of schema integration. Making things visible makes the process of schema integration easier. If you plan to merge two schemas, you do need to make all the current rules fully explicit.

But note that schema integration is a one-off exercise. You can be confident that the range of a type, the instances of a class, the rules of the business, will not change while you are working.

Building a conceptual entity model for long term use is a different matter. The model has to hold object data for years. It has to survive while objects are created, amended and destroyed, while the ranges of apparently fixed values are altered, while the rules evolve.

This gives the modeller a different perspective. The modeller will try to avoid fixing temporary rules (like a range of subclasses) into the data structure. I tend to avoid creating class hierarchies for this and the other reasons explored in Later chapters.

8.10.1 Different conventions

The important thing is to record the semantics of the problem domain, one way of another. Different diagram drawing conventions lead you to draw different-looking conceptual models.

Some people like to represent every term and every fact in a box of its own. You might specify

each fact about an object by drawing a rectangle. You might place a rectangle on each and every line between one named term and another named term. Figure 5p shows the kind of diagram that results from this.

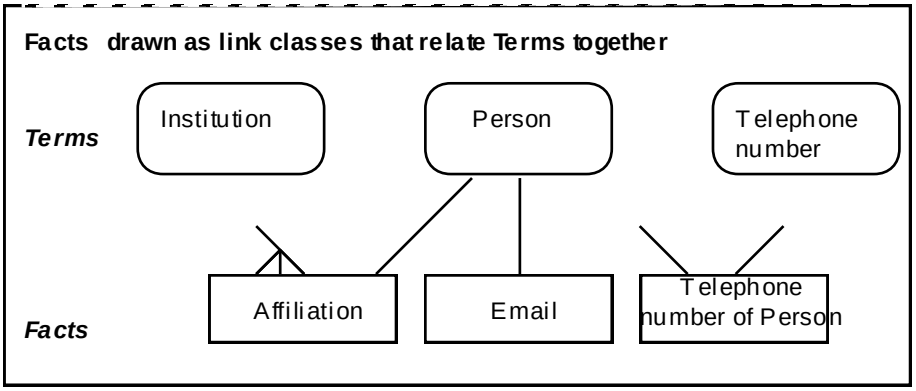


Fig. 5p

There is no law saying you have to represent every attribute or relationship in a rectangle. Doing this usually creates a diagram that is far too large for practical use.

When you are building an enterprise application with perhaps 2,000 data items; you cannot handle a picture that shows every data item in a box (let alone every value of every data item as some of the transformations in this chapter lead to).

It is more convenient to roll one entity up to become an attribute of the other. You may do this where there is a 1:1 relationship, or where one entity is a key-only entity, with no attributes of its own.

Figure 5q features both 1:1 relationships and a key-only entity. It can be condensed by rolling the 'key-only entity' into the 'state entity'.

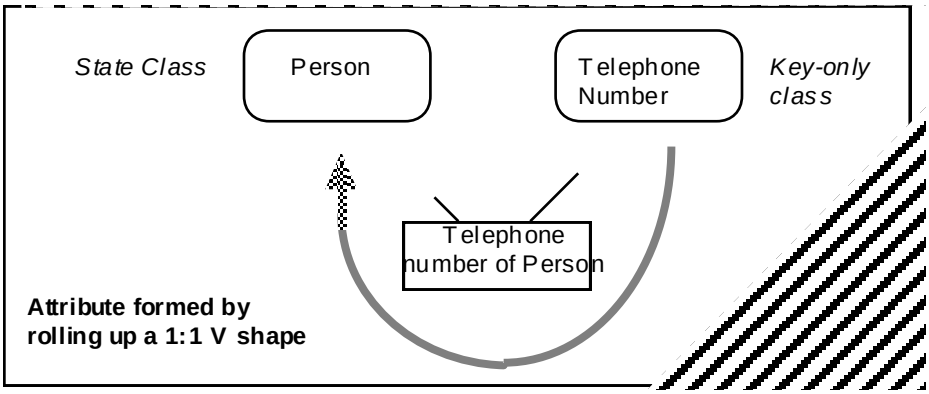


Fig. 5q

8.10.2 The importance of business perspective

Figures 5q and 5r shows that whether a business term becomes an entity or an attribute depends on the perspective of the system's users.

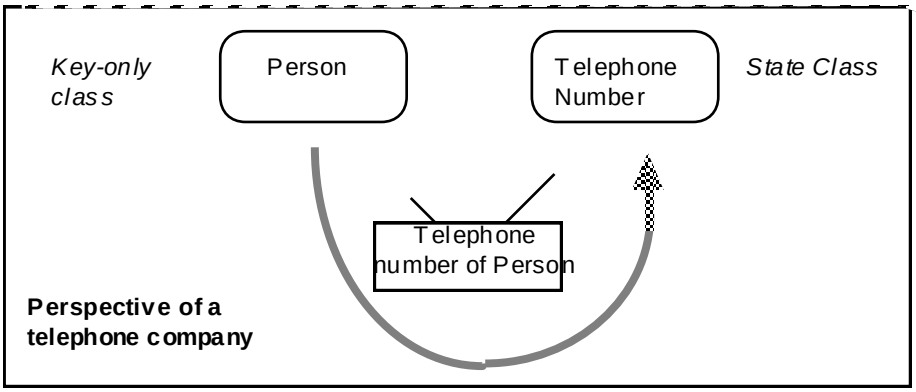


Fig. 5r

Colour might seem indisputably to be an attribute. But Colour might be easily be an important business entity in a company that manufactures paint.

By the way, figure 5s shows the term ‘state entity’ comes from one way to classify different kinds of entity in an entity relationship model.

<table>
<tr><td rowspan="2">Constraint object
Value that constrains business data</td><td>Universal value object
Defined outside the business (colour, month)</td><td></td></tr>
<tr><td>State object
created and destroyed by the business (customer, application)</td><td rowspan="2">Business object
Value that currently applies to objects in the business</td></tr>
<tr><td></td><td>Derived object
derived from values stored in other objects, not a constraint on them
(month of birth)</td></tr>
</table>

Fig. 5s

9. Patterns in simple relationships

How relationships prompt the analyst to ask questions.

6.1 Relationship notation

All the commonly used notations show entities as boxes and relationships as lines between them. I use a diagram notation based on that developed (I think) by Charles Bachman in the 1960s, from which a number of other variants have been derived. It doesn't matter if you prefer another notation (say, after Chen, or OMT) that expresses the same semantics.

To show the dependence of one object on another, or its independence of other objects, our notation uses a continuous line or a broken line:

symbol	**shows that an object at that end of the relationship**
broken line	can exist without the relationship
continuous line	cannot exist without the relationship

Fig. 6a

Fig. 6b

9.1 Parallel and optional aspects

A solid continuous line is a mandatory 1:1 relationship. The objects at either end share the same identity, even though they might be given different keys by a business.

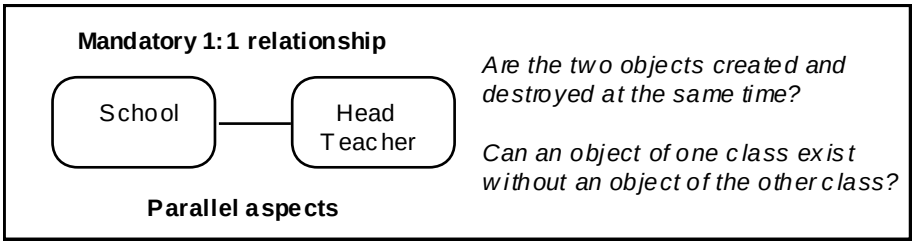


Fig. 6c

Later chapters shows you may draw a mandatory 1:1 relationship to connect the parallel aspects of an aggregate. But as a rule of thumb, you should assume that nature abhors a symmetrical or non-hierarchical relationship.

Ask of a mandatory 1:1 relationship: Are the two objects created and destroyed at the same time? If yes, then the two objects share the same identity. They are what I call an aggregate. For questions about aggregates, see Later chapters.

Ask of a mandatory 1:1 relationship: Can an object of one entity exist without an object of the other entity?

In this case, you may discover that a School can exist without a Head Teacher, but not vice-versa.

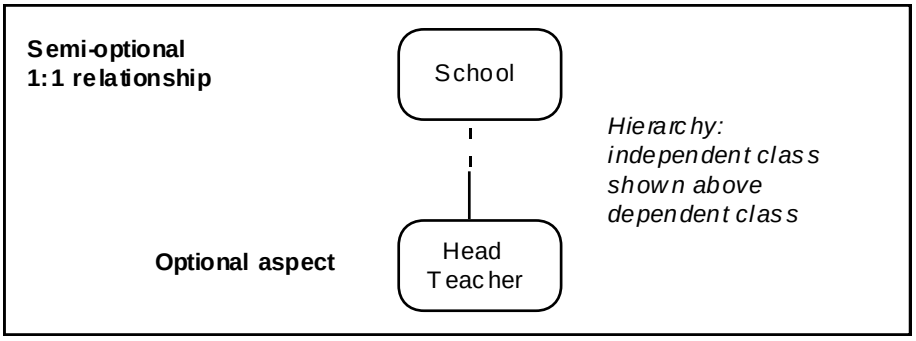


Fig. 6d

In a semi-optional 1:1 relationship, the independent entity is called the **parent** entity of the relationship. The dependent entity is called the **child** entity of the relationship.

The parent-child nature of relationships helps us to draw an entity model in a structured way, with parents towards the top and children towards the bottom.

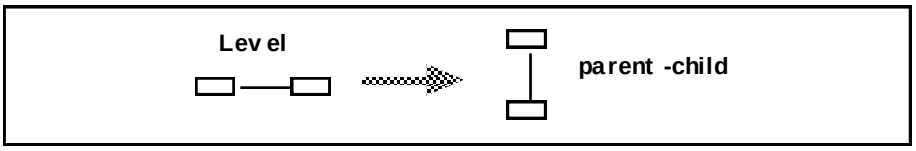


Fig. 6e

This hierarchical structuring gives us opportunities for naming standard shapes, recognising them in specification diagrams, using them to ask questions, and teaching the analysis and design implications.

9.2 Aggregates and is-a trees

Ask of a semi-optional 1:1 relationship: Might there be more child objects than parent objects? If no, the model is incorrect, since the population of objects contradicts the cardinality specified by the relationship.

Ask of a semi-optional 1:1 relationship: Does it describe an aggregate or a class hierarchy?

Figure 6f shows you can test the meaning by trying to write either 'belongs to' or 'is a' on the child or subclass end of the relationship, and 'may have' or may be' at the top.

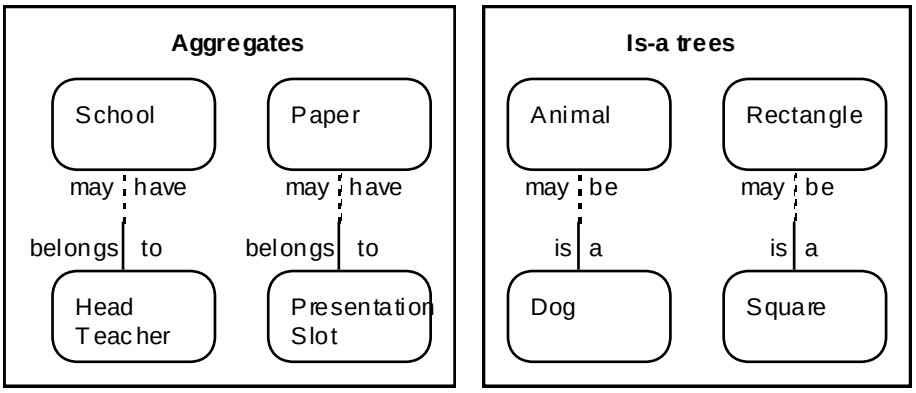


Fig. 6f

What I want is a graphical notation that combines the cardinality rules specified by a database structure notation, with the semantics specified by object-oriented notation. Figure 6g shows a notation you can use to express the different semantics, while retaining the cardinality information.

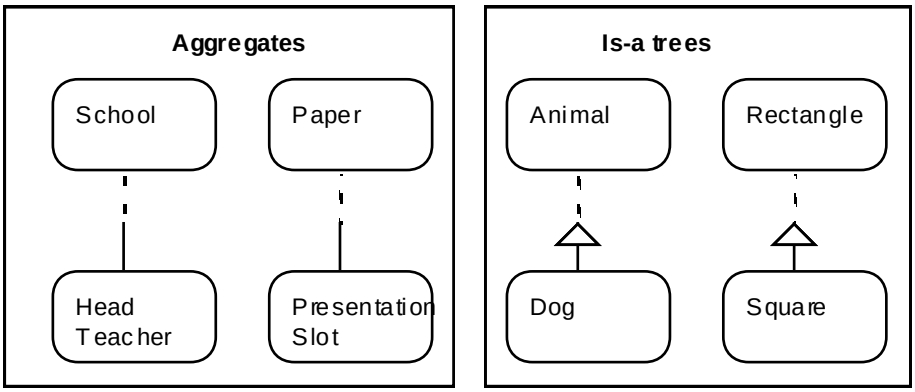


Fig. 6g

Figure 6h shows a deep is-a tree.

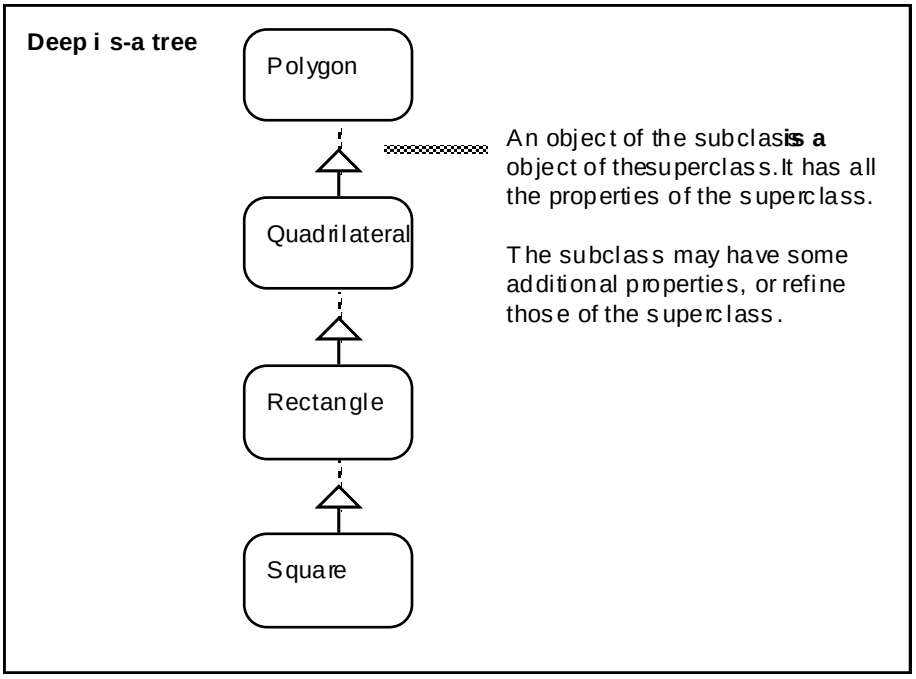


Fig. 6h

Figure 6i shows notations you can use to show aggregates and is-a trees with several overlapping children or subclasses. The fact that the lines are dotted at the top means the children or subclasses may not apply.

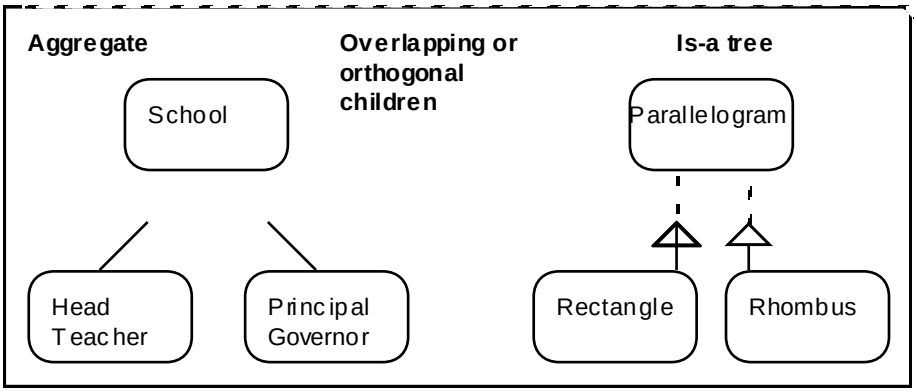


Fig. 6i

Figure 6j shows notations you can use to show that the children or subclasses of an aggregate or class hierarchy are mutually exclusive.

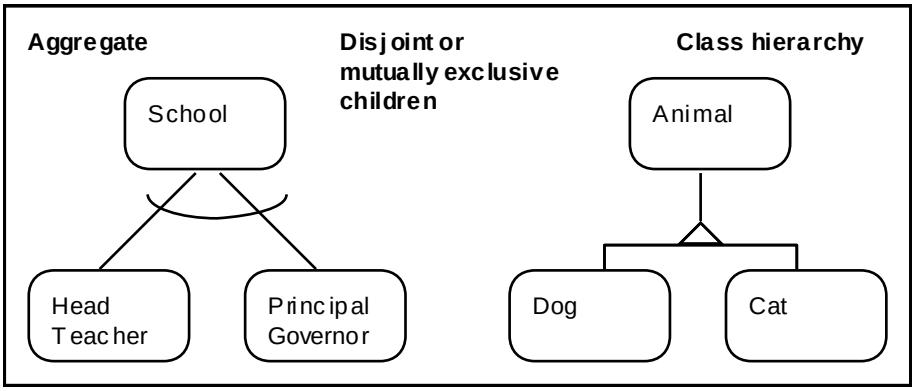


Fig. 6j

Both these diagrams say 'either one case or the other case'. If you wanted to allow 'neither case' as well, then you would draw the top half of the relationships with a dotted line

In this short section I have entered the territory of object-oriented modelling; see later chapters for much more discussion of aggregates and is-a relationships.

9.3 Optional aspects and states

Ask of a semi-optional 1:1 relationship: Can a parent object be related to many child objects over time? And do we want to record past children? If so, then the relationship becomes 1:N, as shown later. You should add historical entities and relationships to the model wherever they are needed in order to support users' requirements for information.

It is usually easiest to start by drawing the entity model without history. The model above will record for a School only the currently employed Head Teacher; it won't keep a history of past Head Teachers. Let us say we are not interested in this history.

Ask of a semi-optional 1:1 relationship: Is the child object a singular optional attribute of the parent object? If yes, you might make the optional attribute mandatory and roll it up. (See the questions about aggregates in Later chapters.)

However, if the child object is a *group* of attributes in 1:1 correspondence (so if one is present then all are) then the semi-optional 1:1 relationship saves you from specifying the rule of 1:1 correspondence between attributes of the group within a larger entity.

Ask of a semi-optional 1:1 relationship: Is the child object merely a later stage in the life of the parent object? If so, then you normally roll up the child entity into the parent entity, with the same benefit/cost tradeoff as above.

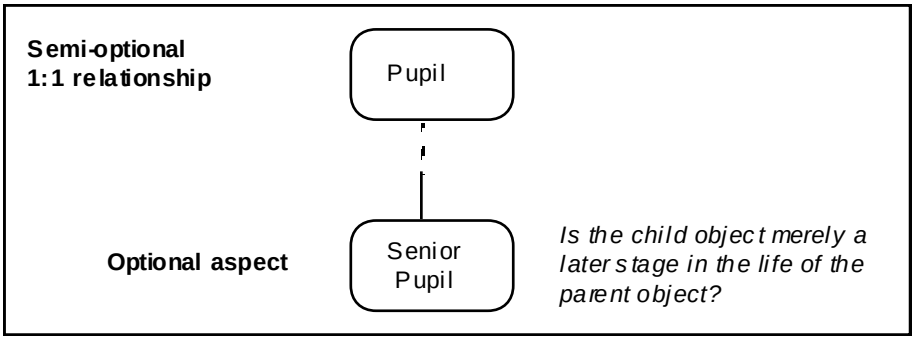


Fig. 6k

An object in this kind of model becomes divided between entities as it progresses through its life. Somebody who starts as a Pupil may later become a Senior Pupil as well. This is an unnecessary elaboration, leading to some redundant design and coding effort. Where the child entity represents merely a later stage in the life of the parent entity, you may roll the two entities into one.

The convention I favour is that objects don't normally change class.

Some people propose the reverse, that you should create a subclass for each state an object may pass through, showing them as mutually exclusive subclasses in a class hierarchy. But this adds to the design and coding effort. It increases the number of entities in the design and the complexity of coordinating separate objects during an enquiry or update process. If each entity becomes a database table, it slows down performance, since more objects must be retrieved and stored.

Later chapters say a great deal more about types and states.

9.4 Loose associations

Returning to the first example and first question, you should ask about the objects at both ends of the relationship: Can they exist without it? If yes, you should define the relationship as being optional at one or both ends.

In this case, you may discover that the system has to record both headless Schools and unemployed Head Teachers.

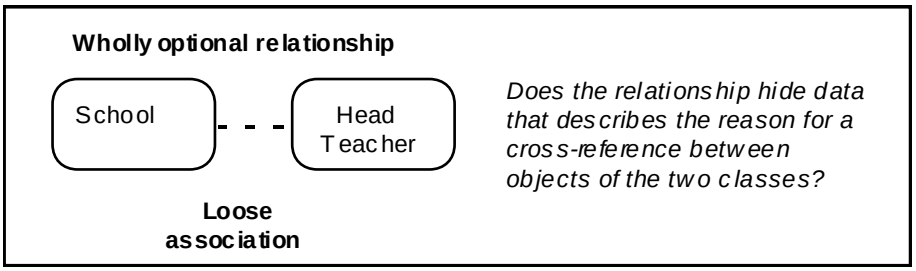


Fig. 6l

Again, nature abhors a symmetrical or non-hierarchical relationship. There are two ways to introduce a third entity into the picture.

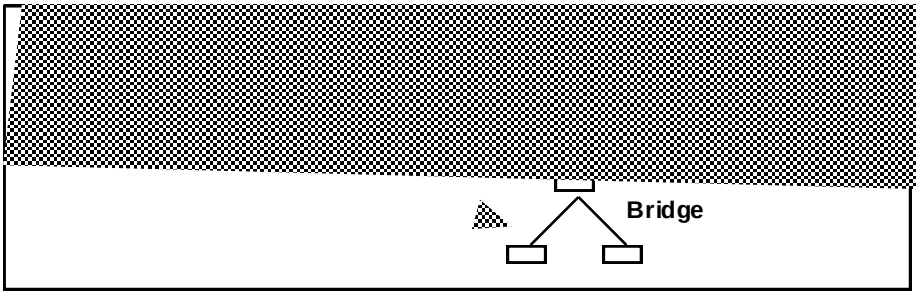


Fig. 6m

9.4.1 Constructive pattern: 1:1 V shape

Ask of a wholly optional 1:1 relationship: Does the relationship hide data describing the reason why the objects are linked? If yes, you should create a link entity.

E.g. you may discover that a School and a Head Teacher only become linked via a Contract. You can redraw the entity model in a hierarchical V-shaped structure.

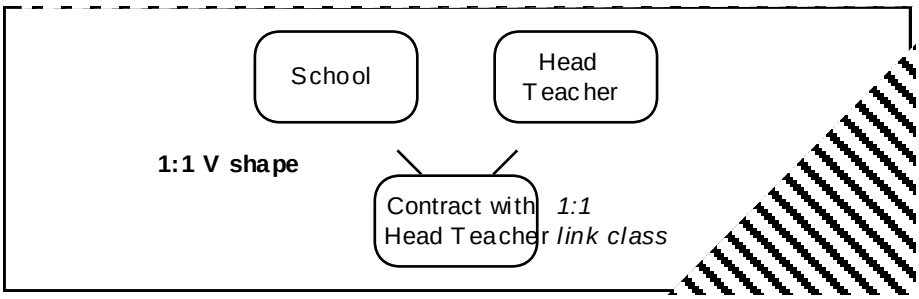


Fig. 6n

The link entity at the bottom of a V shape acts to constrain the relationship between the two higher entities. It gives the relationship a meaning. It restricts the possible links between objects of the two higher entities; you can only connect objects which are in reality connected by this meaningful relationship.

Remember: the identifier or key of an entity state record is not just a database concept, it is a necessary business concept. It enables you to:

a) distinguish that object from another of the same class and

b) map the entity state record onto a real-world object in the business environment.

Later chapters includes analysis questions that are relevant to a 1:1 link entity.

9.4.2 Constructive pattern: 1:1 Bridge

The 1:1 bridge shape gives two child entities a common parent. Use it where entities are additive roles rather than mutually exclusive subclasses.

For example, suppose that you wish to combine two legacy systems, one from Europe and one from the US, that maintain information about an overlapping range of stock types. The two systems identify their range of stocks by different numbering systems. Some European stock types are the same as those in the US, some exist in only one of the two regions.

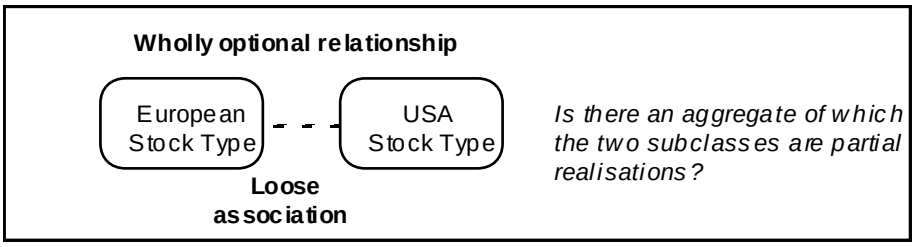


Fig. 6o

Ask of a wholly optional 1:1 relationship: Is there an aggregate of which the two entities are partial realisations?

In this case you might create an entity that sits over and between the two systems.

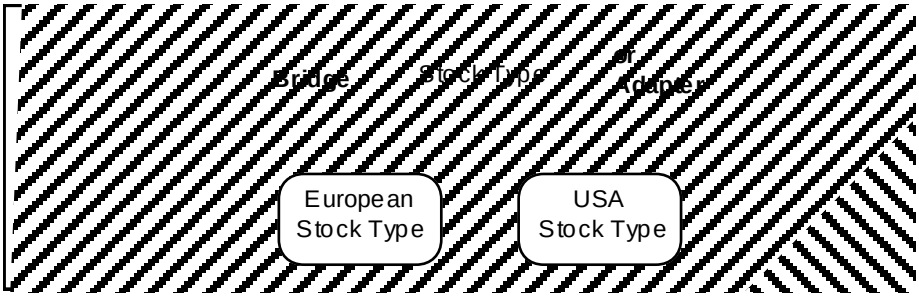


Fig. 6p

The entity model above says ‘either, both or neither’. It says you can instantiate a superobject that has no related subobject. Specifying an ‘either or both’ constraint to exclude ‘neither’ is beyond us here.

9.4.3 Aggregates

Both the 1:1 V shape and the 1:1 bridge shape are aggregates, and they prompt analysis questions. Aggregates and semi-optional relationships often transform in one of the ways described in Later chapters

9.4.4 Comparable object-oriented design pattern: Adapter

The Bridge shape is akin to one of Gamma et al. patterns called Adapter that is designed to 'convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.'

9.5 1:N relationships

A typical object-oriented system records only the current state of transient objects. A typical enterprise application records historical data about long-lived real-world objects. Both the real-world objects and the entity state records are persistent. History and persistence make for 1:N relationships.

> Ask of any kind of 1:1 relationship: Can an object of one entity relate to more than one object of the other entity over time? And do we want to record past children? If yes, you should show the manyness of one or both ends of the relationship.

The result of asking this question is that the majority of associative relationships in enterprise applications turn out to be 1:N, shown in our notation using a fork:

symbol	shows that at that end of the relationship
fork on line	there may be several objects
no fork on line	there may be no more than one object.

Combining the continuous or broken line with the fork, the notation can show four kinds of 1:N relationship, as illustrated below.

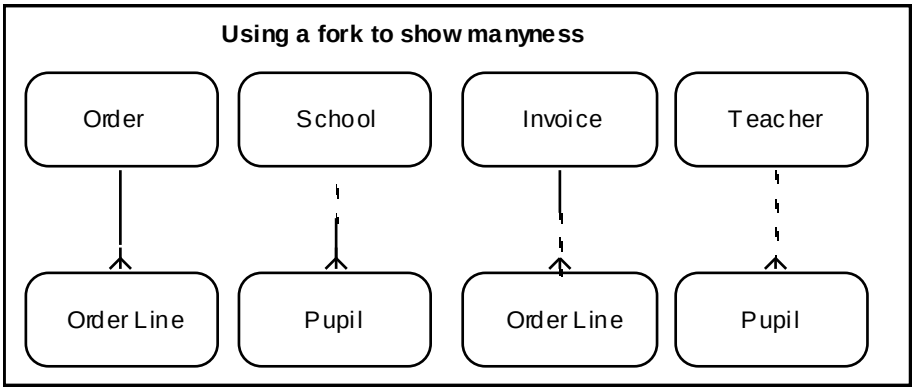


Fig. 6q

In a 1:N relationship, I call the entity at the 'one' end the 'parent' entity of the relationship; and the entity at the 'many' end the 'child' entity of the relationship.

> Ask of a 1:N relationship: Can a child object exist without a parent object? If no, a child *must* be owned by a parent, then the relationship line is continuous at the child end.

For example:

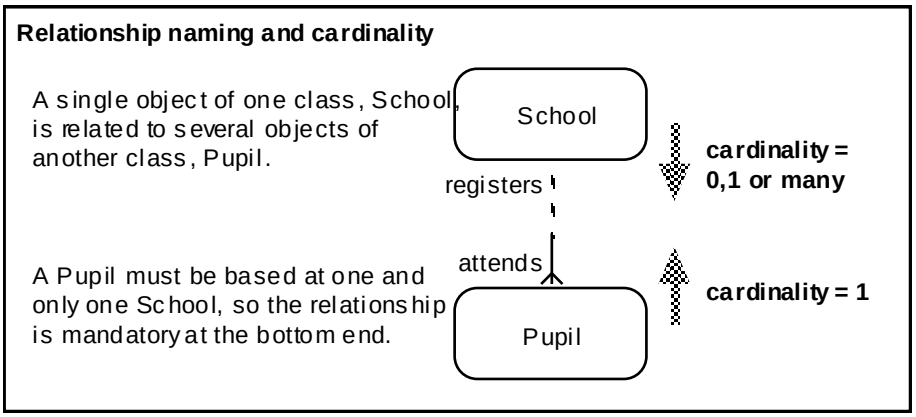


Fig. 6r

It is often helpful to name a relationship at both ends, as I have done here.

Ask of a 1:N relationship: Can a parent object exist without a child object?

The relationship that exists between Teacher and Pupil is optional at both ends. Not all Teachers manage a class. Not all Pupils are assigned to a class with a class teacher, only the younger ones. So the relationship line is broken at both ends.

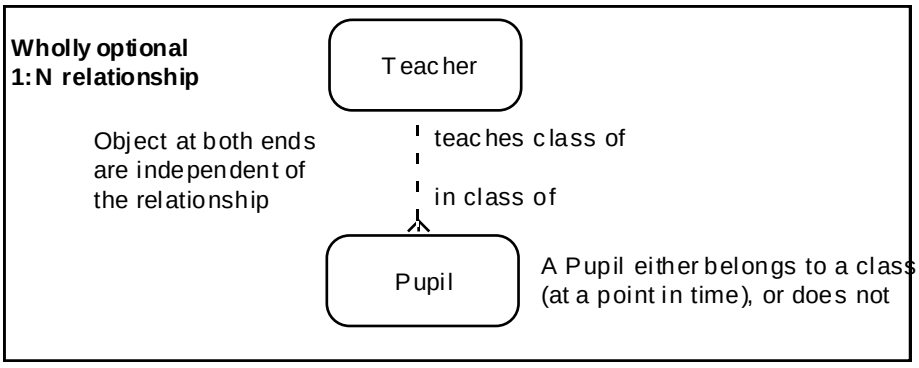


Fig. 6s

You may wonder about introducing School Class into the model. Thankfully, the concept is not recorded in our system, otherwise I would have to worry about confusing ‘Class’ with ‘class’ in our discussion here.

9.5.1 Three more questions about constraints

There are at least three more questions you should ask about any 1:N relationship.

- Can a parent object have more than one active child object at once?
- Does a parent object retain historic children as well as active children?
- Can a child swap from one parent to another?

It might be possible to extend the notation to show all the answers in a graphical form. But this way lies madness. If you try to show all constraints on an entity model, you end up with a

picture that is so large, so rich in semantics, and so complex in appearance that you cannot use it.

It is better to ask these questions during Event Modelling and object behaviour analysis, and document the answers there, in the diagrams for each persistent entity class and each transient event class.

Of course, you may revise or extend the entity model with new entities or relationships after you have answered the questions.

9.6 N:N relationships

You may at first include N:N relationships an entity model of business objects.

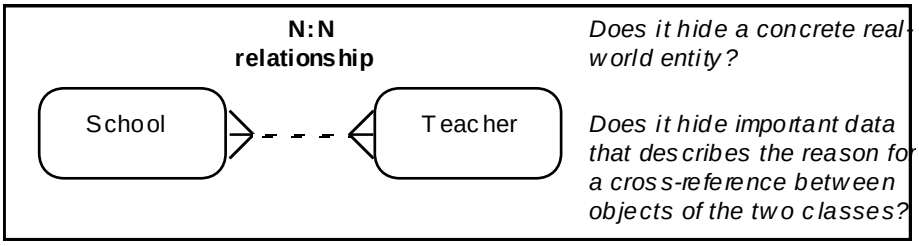


Fig. 6t

Again, nature abhors a symmetrical or non-hierarchical relationship. You may draw explicit N:N relationships in the early stages of a model, but you should always resolve them before completing the specification.

> Ask of a N:N relationship: Does it hide a concrete real-world entity? If yes, you should create a link entity.

E.g. you may conclude that the relationship is established via a Pupil.

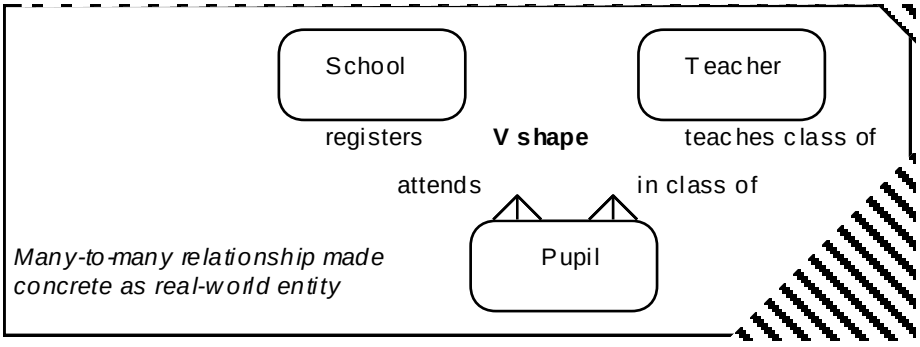


Fig. 6u

A Pupil is a very concrete entity, but a weak way associate a School with a Teacher. Not every Teacher is a class Teacher. Not every Pupil is assigned to a class Teacher.

> Ask of a N:N relationship: Does it hide important data describing the reason why the objects are linked? If yes, you should create a link entity.

Asking about this case, you might discover the Employment Contract. You can reveal the

hidden data by simplifying the N:N relationship into two or more 1:N relationships.

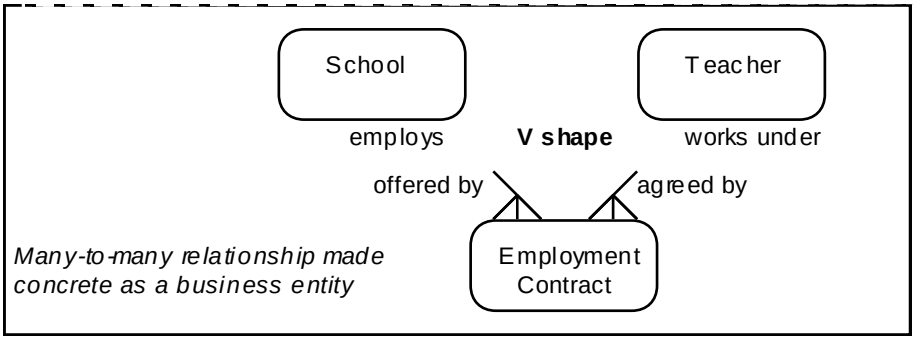


Fig. 6v

I will return to discuss how to use the V shape to constrain a system's behaviour, and some design issues raised by it.

We've looked mainly at questions about single relationships. Part Two discusses some of the ways in which relationships may form larger and perhaps more interesting shapes.

10. Patterns in relational data analysis

This chapter reviews traditional data analysis techniques. As Winston Churchill said in a very different context: 'It may be unfashionable, it may be unpopular, it may be unpalatable, but its the truth.' Well, it is part of the truth. I add a few analysis questions to be asked during data analysis.

10.1 From relations to entities

Do not confuse a database view in the UI layer with the data structure of the underlying business. You must decompose aggregate objects displayed in the UI layer for processing inside the system.

Business rules belong to the entities in the underlying application, not the aggregate objects in the UI layer (though these might unfortunately be called 'business objects').

Relational data analysis is a good way to reduce the aggregates of data items found on forms, screens or data files, into a set of simple normalised relations. Allow us to equate the concepts of entity and normalised relation for the time being.

Normalisation, a technique used in data analysis, is a fine example of generative patterns, of model transformation by question and answer. It reduces complex data structures to the simple building blocks from which they are made. It reduces unnormalised data in stages through successive normal forms.

The starting point for the example below is the data to be found on a batch of Sale Returns emailed to head office by a salesman.

UNORMALISED	**1ST NORMAL FORM**	**2ND NORMAL FORM**	**3RD NORMAL FORM**
Separate the entity from the repeating group	**Salesman**	**Salesman**	**Salesman**
	Salesman name	Salesman name	Salesman name
Sale return	**Sale**	**Sale**	**Sale**
Salesman name	Salesman name *	Salesman name *	Salesman name *
Sale date	Sale date	Sale date	Sale date
Product name	*Product name*	Product name *	Product name *
Product price	*Product price*	---	---
Item quantity	Item quantity	Item quantity	Item quantity
Cust Num	Cust Num	*Cust Num*	Cust Num *
Cust name	Cust name	*Cust name*	---
	Remove attribute who value depends on part of the key	**Product**	**Product**
		Product name	Product name
		Product price	Product price
		Remove attribute who	**Customer**

		value depends on part of the key	Cust num
			Cust name

Fig. 6a

10.2 Don't dismiss normalisation

Some object-oriented designers dismiss normalisation, because it is entirely data-oriented, but there are many things to be said in favour of it.

For one, it encourages you to think in detail about the users' requirements for information. Several of the case studies published to illustrate object-oriented methods appear to be complex and challenging – finding the right classes is relatively difficult or mysterious. My study of the case studies suggests thet would be easier if the authors had defined some input and output messages at the start, then (dare I say it) applied a little relational data analysis to those inputs and outputs!

For another, the normalisation process depends on the analyst choosing an identifier or key for a data group. The key is underlined in our examples. When a data group is in third normal form, each attribute is 'determined by' or 'dependent on' the key. Given the value of an object's key there is only one possible value for any given attribute of that object.

Why is thinking about keys helpful?

10.3 Look for keys to reveal the business perspective

Choosing a key may seem merely an implementation decision. Indeed, you might not decide between various possible candidate keys for an entity state record until relatively late in the design process.

But the *intention or desire* to give an entity state record a key is not just a database concept, it is a business concept. When you choose a key during relational data analysis you are making a statement about the business perspective you are taking of the real world.

Users need a key that will enable them not only to:

• distinguish one object from another of the same class, but also to

• map the entity state record onto a real-world entity in the business environment.

One reason for taking required output reports, or a legacy database, as the source documents for data analysis is that these sources will reveal the things the users already care about enough to have awarded keys.

10.4 Choosing a key for an entity

The key must uniquely identify an object, and not have more than one value for it. In other words, the values of the key must be in 1:1 correspondence with objects of the class.

You may have to choose between several candidate keys. Since users need keys that help them map entity state records onto real-world entities, you should favour natural attributes over artificial identifiers.

If you have to make up a key from a long list of attributes, then so be it. The important thing as far as data analysis is concerned is that you have established *the business need for a key*.

In the old days, designers might have said to choose numbers over text, short items rather than long ones, and few items rather than many, but the ability of users of a graphical user interface to select objects from lists now saves people from having to type long multi-item keys.

10.5 Teaching normalisation as graphical model transformations

Students normally learn normalisation by completing a table such as the one drawn above. This is a bit like practising scales when you start to play the piano.

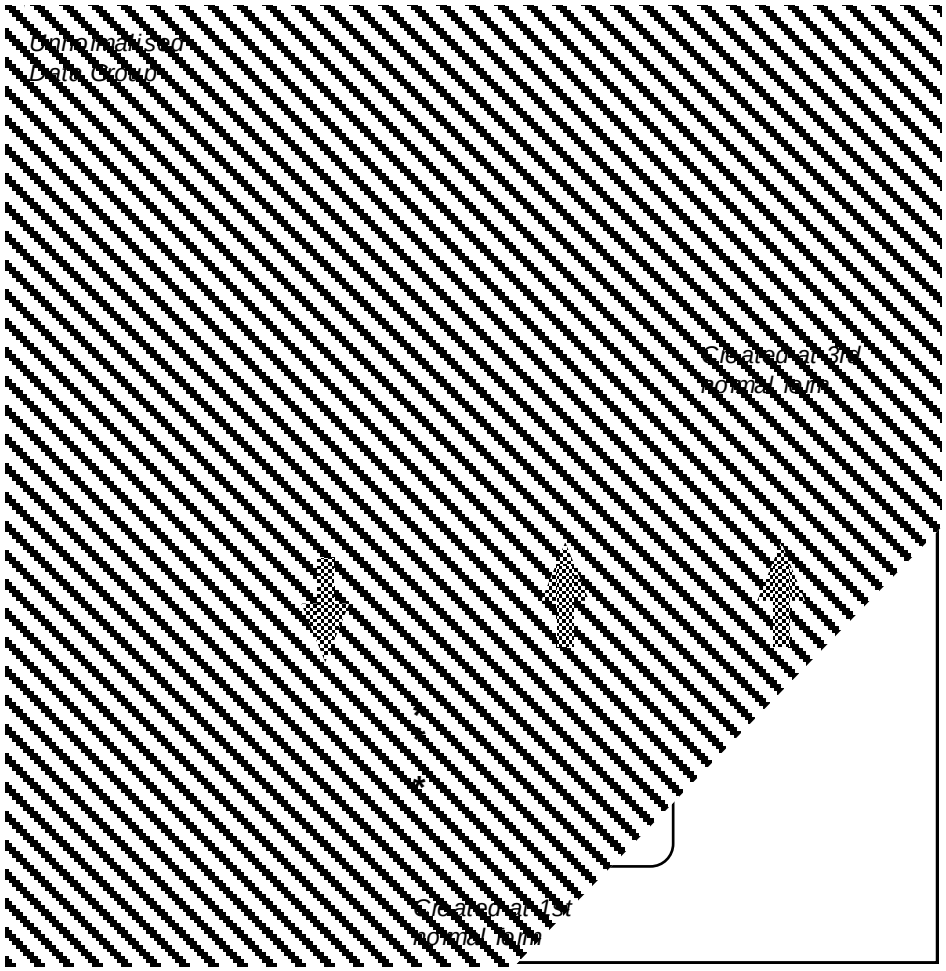
Few if any professionals do data analysis the way students are taught to, just as a concert pianist almost never plays a scale during a stage performance.

Professionals use data analysis to place facts into an existing entity model, albeit an informal or provisional entity model. They:

- reconcile input and output data flows with an existing entity model
- refine an informally defined entity model
- reverse engineer entities out of an existing database schema

Data analysis is a technique for both forward and reverse-engineering. Nowadays data analysis is a common way to start reengineering a legacy system, it helps you take advantage of the effort that has already gone into to defining the legacy database.

The analogy between analyst and concert pianist is a poor one, because it is possible to teach novice analysts to do data analysis the way the professionals do it. The trick is to focus on the way normalisation reshapes an entity model, on graphical model transformations.



10.6 Choosing the key for an unnormalised data group

The analyst starts normalisation by choosing an identifier or key for the entire unnormalised data group. This is not always easy. The key should uniquely identify at least one other data item. So the choice of key in figure 6b is a poor one.

Choosing a poor key for unnormalised data has little effect on the entities defined at the end of the analysis, but it dictates the path that normalisation takes, and it leaves you with a key-only entity. This key-only entity may turn out to be redundant, not interesting to the business.

10.7 From unnormalised to first normal form

Following the rule that a fork grabs an asterisk - a relationship grabs a foreign key - you can draw the data groups that result from data analysis as entities connected by relationships. So let us repeat the data analysis of the example by following generative patterns.

Figure 6b shows the standard pattern for the first normalisation step is to drop out a child

entity.

Notice the instruction to choose a key for each new child entity. We'll come back to talk about this in a moment. Consider the example in figure 6b.

If you had chosen Product Name as the key for the unnormalised data in a Sale Return, then Salesman would not end up as an entity on its own. But having chosen Salesman Name as the key, every other data item is immediately removed as repeating data, data that has several values for the key, so you end up with Salesman Name as a key-only entity.

In other cases, you might choose to drop the key-only entity. But in this case, the Salesman is probably an important entity and worthy of record. You may well discover an attribute for Salesman during data analysis of another form, screen, report or file.

10.7.1 Choosing the key for a repeating group at first normal form

A choice between candidate keys often arises when choosing a key for an entity revealed at first normal form. Typically the revealed entity is a link or bridge between two or more entities with simple keys of their own.

Ask of a link entity: What uniquely identifies an object of the link entity?

Consider the choice of key for a Sale object linking Product, Customer and Salesman. Of course you can manufacture a unique identifying number as in figure 6c.

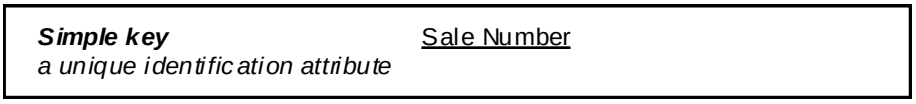


Fig. 6c

But this is not user-friendly. If the users do not already use Sale number in their business, you might do better to use a compound of those parent entities users do have keys for.

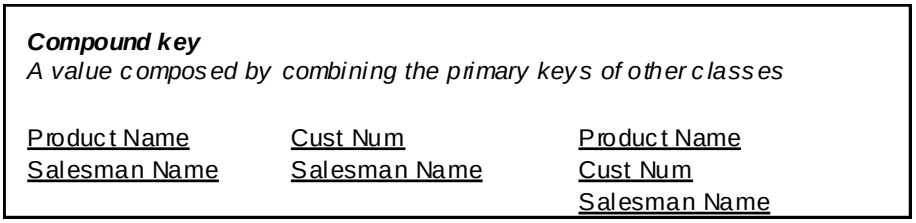


Fig. 6f

The trouble with a compound key is that objects of the parent entities can only be linked once, by one link object. If you want to allow duplicates, you have to extend the compound with date and time attributes, or with some other qualifying element or sequence number.

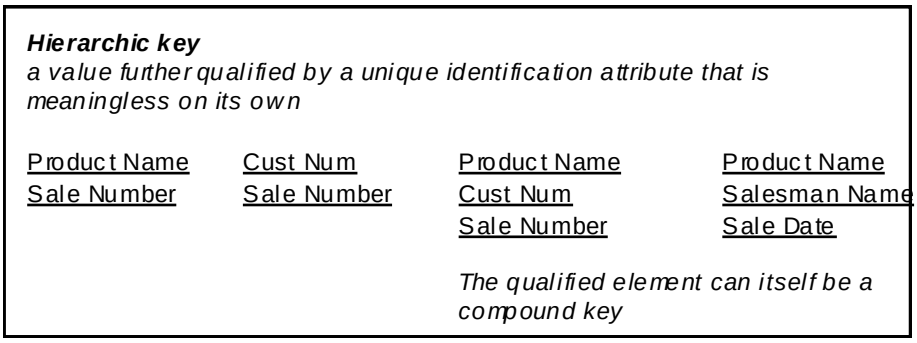


Fig. 6g

The Sale Number in these examples is used to extend the range of values provided by combining the keys of parent entities. All of these hierarchic keys for a Sale allow the possibility that two parent objects (Product and Customer) can be joined by several Sales. We'll come back to consider this kind of case in Part Two, since it implies the V shape is really a Y shape.

10.7.2 Recognising the structure of repeating groups

There is another difficulty with the way data analysis is taught. Listing data items in an unnormalised data group means you lose sight of the data structure you started with.

Given a complicated document or file, you may need to divide it at first normal form into a complex structure of parallel and nested repeating groups.

It is impossible to visualise this structure by looking at a list of unnormalised data items. It is much easier if you record the repeating data groups as distinct entities from the outset, or draw boxes around data groups on the original document.

10.8 From first to second normal form

The standard pattern for the second normalisation step is to raise a parent entity with a key that is part of the key of the child.

In general, given any multi-item key it is well worth asking about the classes that might exist identified with one part of the key as their own key. See Part Two.

10.9 From second to third normal form and beyond

The standard pattern for the third normalisation step is to raise a parent entity and assign the determining attribute(s) as its key, as shown below.

10.10 Other normal forms

Fourth and fifth normal forms are discussed in Part Two. Boyce-Codd normal form is a

variation of third normal form that eliminates possible anomalies where there are several candidate keys which share a common attribute, a complication that need not concern us here.

10.11 Merging of corresponding relations

You will normally merge the end results of various data analysis exercises. You can merge any classes whose objects are in 1:1 correspondence; this is usually indicated by their having the same key.

Two of the classes in our little case study pick up an extra attribute from data analysis of other documents.

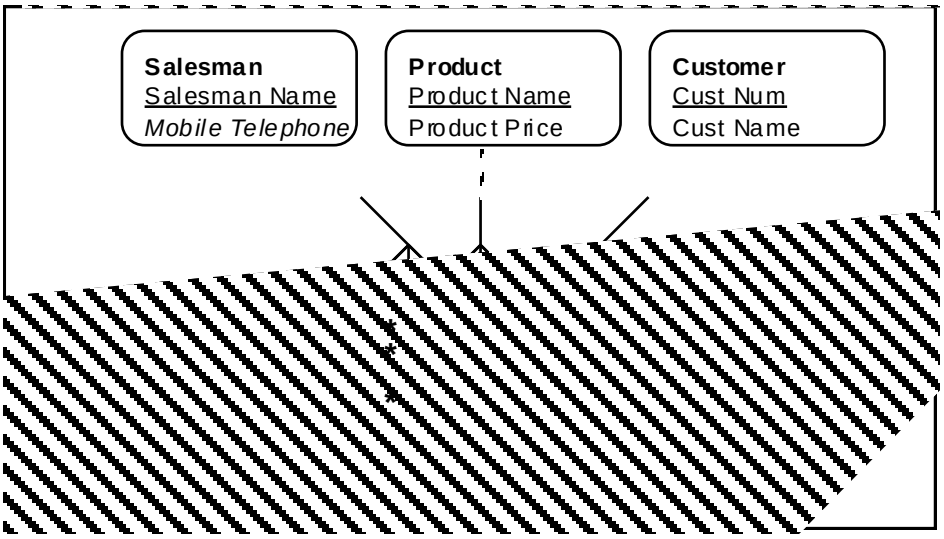


Fig. 6k

Notice that Cust Address has sneaked into the Sale class. This happened when analysing the Delivery notes given to drivers instructing them where to deliver the products that have been sold.

10.11.1 The two TNF tests

After data groups have been merged, items will have been brought together in new combinations, so you should apply two tests to ensure that the resulting classes are in third normal form (TNF).

> Ask of a class: the first TNF test: Is there only one value for each data item in the class, given a single value for the key? If no, there has been a mistake and a first or second normal form reduction to be investigated.

> Ask of a class: the second TNF test: Is the value of an item determined by a non-key item, rather than the key? If yes, there is a third normal form reduction to be investigated.

For example, is Cust Address really determined by Cust Num, and best moved into the Customer class? Or perhaps the address is recorded afresh on each Sale, because users use

it only as the delivery address for the Sale, and a Customer can have many delivery addresses? Or perhaps the users really need two addresses - one for customer and one for delivery.

10.12 Don’t overlook historic data

Data analysis is never as easy as presented on training course, because you have to ask business analysis questions about how data changes over time, and whether historic data has to be remembered.

10.13 The relation shape

A relation is an aggregate of several attributes. Chapter 4 showed you might define each attribute as a key-only parent entity in its own right. What I call the ‘relation shape’ is a class with three or more parents.

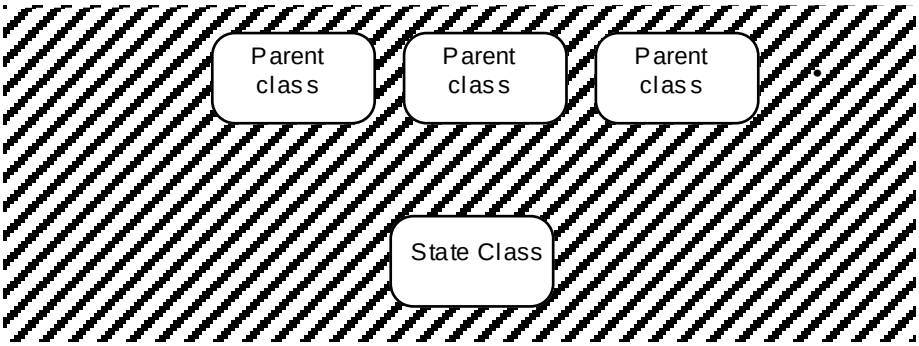


Fig. 6l

For example, let us say a Product is a type of Ingredient with a unique combination of four other characteristics, each of them user-defined. You can define each attribute as a class in its own right, as shown below in a fraction of the full model:

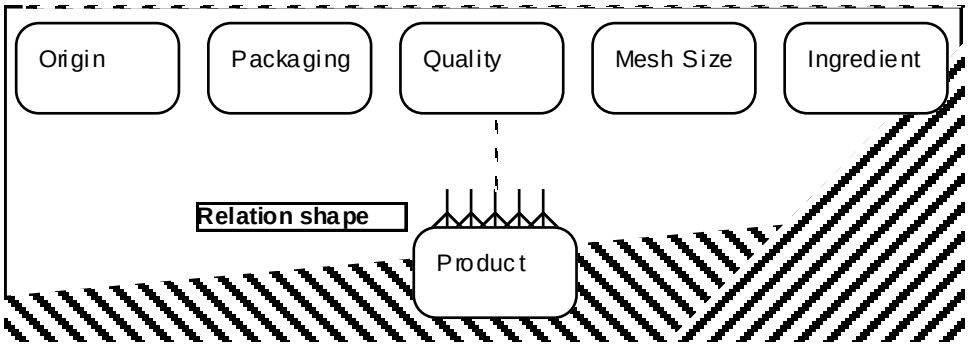


Fig. 6m

Users will define the valid range of each attribute by creating objects of the parent entities. Suppose it turns out that users start to record products in the database that cannot actually exist in practice, products with invalid combinations of size and ingredient. Four solutions might

be designed.

Management solution

A weak solution is to place some kind of security constraint, using a password perhaps, on who is allowed to set up products in the system.

UI layer solution

A weak solution is to code the rules as constraints on data items where they are entered into the system, in the user interface code. This may prove difficult to maintain as the code is added to several datan entry screens. A stronger solution is to record the constraints in reusable modules underlying the user interface.

Data services layer solution

A strong solution: specify validation rules applying to data items in a data dictionary attached to the database management system. Unfortunately, few database management systems come with a sufficiently clever data dictionary, one that can apply the rules dynamically to a live system. If you do have a clever enough data dictionary, then think of it as belonging in the Business services layer rather than the Data services layer.

Business services layer solution

A strong and practical solution: record the validation constraints as a cross-reference table, or link class, in the entity model of the Business services layer.

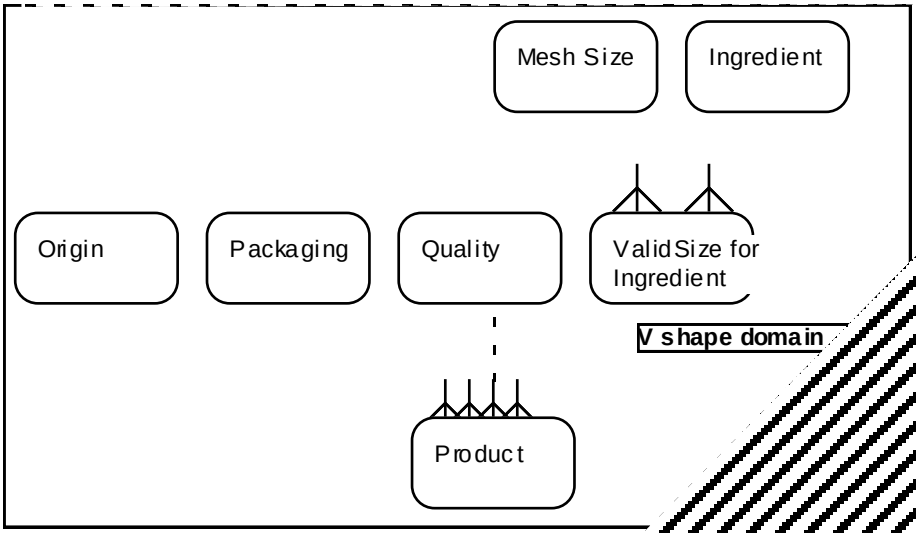


Fig. 6n

The introduction of a V shape domain above the State Class currently looks like the best design option for most applications.

> Ask of a relation: Will users control the valid combinations of different attributes? If yes, then create a V shape domain class.

11. Why entity modeling is not enough

11.1 Relational theory is not enough

There are weakness in the relational view of the world.

Data-centred, database semantics are not handled

People have attempted to extend the relational model to accommodate business rules. These approaches are sometimes called 'semantic entity modelling'. The difficulty is that these approaches tend to be so heavily data-centred that you have to think in rather abstract and difficult ways to discover and define the processing logic and processing rules.

In our view, object interaction and object behaviour analysis *is* a 'semantic entity modelling' approach, though you specify the rules on event models rather than the entity model itself. The data and process models are all part of one coherent conceptual model.

Objects are key-oriented rather than type-oriented

Relational theory does not account for mutually exclusive or optional data. I will show how object interaction and behaviour analysis deals with class hierarchies of super and subclasses.

Parents don't know where their children are

Relational theory suggests that a parent entity should not know about its children. There are no tables or lists, only foreign keys. This minimises data redundancy, but access from parent to child involves a great deal of processing redundancy. This is a big factor in slowing down system performance. Where a database is distributed it is almost inconceivable that a parent entity should not somehow know where its children are.

This is an issue for the Data services layer and you should not even have to think about when defining the Business services layer!

How access from parent to child is achieved is a matter for the Data services layer. The database designer may implement a relationship in either relational or network style. In network databases, tables and lists are allowed, especially for storing relationships. Thus, while data redundancy is thus permitted, access from parent to child involves no redundant processing.

Aggregate objects cannot be stored

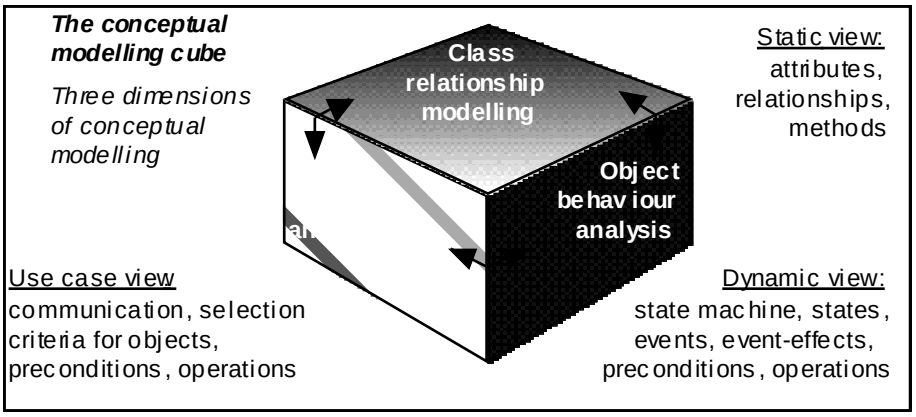
You cannot store objects without repeating data in a relational database, because relations must be in first normal form. I will show this is fine and correct for the Business services layer. The Business services layer must separate out the low-level normalised classes, partly for update efficiency and partly so that they can be viewed from many different perspectives.

You may however choose to design and process aggregate objects in the presentation and Data services layers. Some people use the term **business object** to describe an aggregate object in the UI layer. Once more, be careful not to confuse a database view with the database itself. Business objects in the UI layer must be decomposed for processing in the Business services layer.

To supplement data analysis and overcome the above weaknesses, object interaction and behaviour analysis techniques are needed.

11.2 Entity modelling is not enough

There are two main techniques that complement entity modelling. Both help to validate and improve the entity model.



Chapter 7 introduces event modelling techniques that can be used in the Event Modelling face of the cube. The volume “The Event Modeler” goes into much more detail.

11.3 Relationship cardinality is not enough to capture all constraints

You can specify static and invariant constraints (applied in every case and unchanging) as properties of data. You can specify validation rules governing the ‘domain’ of a data item, and you can fix referential integrity rules by specifying the optionality and cardinality of relationships.

But some constraints are dynamic or changeable, so it is not appropriate to build them routinely into implementation database structures, or even a logical entity model.

E.g. English law lays down a number of constraints governing a wedding event: a marriage must relate two partners, no more, no less; one partner (the husband) is male; one partner (the wife) is female; both partners must be over 18 years of age; a person can have only one marriage at a time; a person can only have marriages in their sex of birth. And there are further 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.

You need ways to make all constraints explicit, not just referential integrity rules. 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 you can specify all remaining constraints as preconditions on events.

Chapter 7 includes an illustration.

12. Event Modelling

An overview of Event Modelling techniques that specify how events hit objects in the entity model.

12.1 Introduction

Level of granularity

>>

Enquiry identification

>>

Event identification

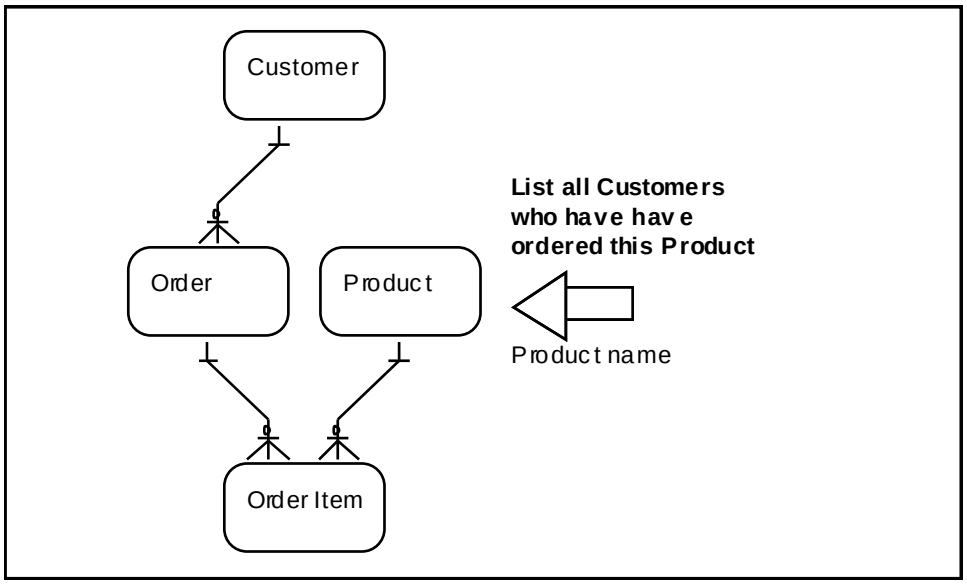
>>

12.2 Enquiry models

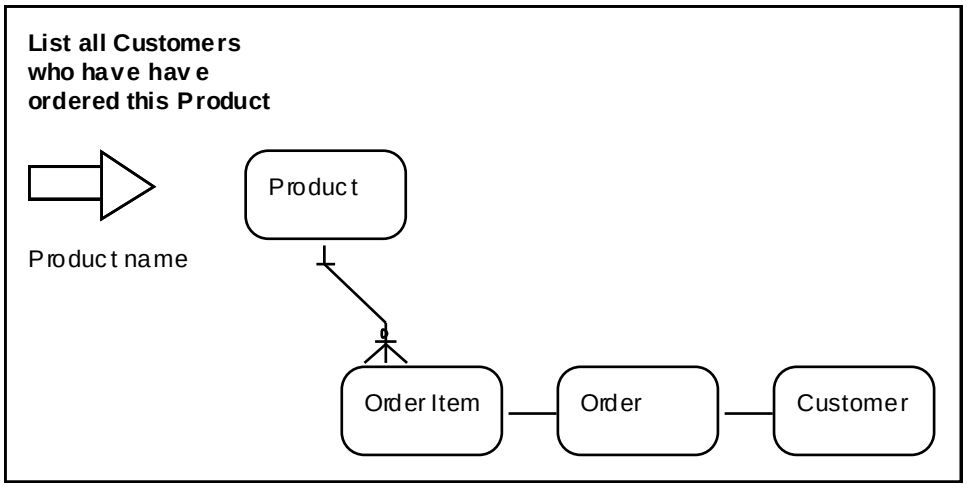
Validation of the entity model during analysis

You should validate the classes and relationships by testing that they support all the enquiries that users say they want to make of the required system. In simple cases, programmers can do this by defining an SQL query. At an early stage of analysis, especially for complex cases, it helps to define the enquiry model for the enquiry requirement.

To verify that atomic enquiries within the system functions can get the information they need by accessing entities - to test that every known output data flow (message, report or file) can be derived using the relationships in the Entity model - you can draw every enquiry access path as an enquiry model. This means defining the entry point object (identified by the input data parameters) and the navigation path along relationships to collect the required output information from other objects.



You can redraw the enquiry model from the perspective of the specific enquiry.

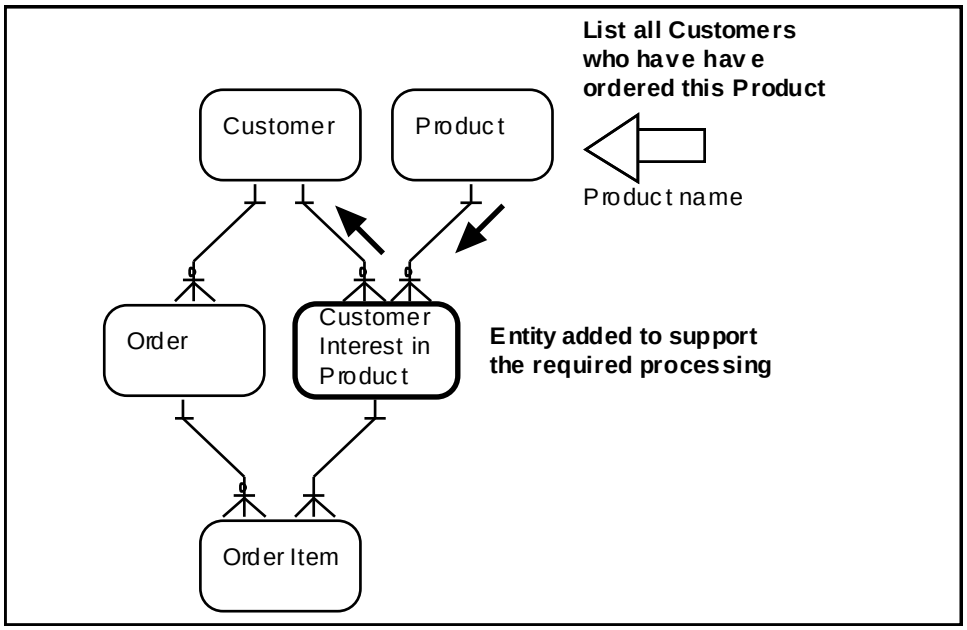


Notice that an enquiry process that follows this particular access path will find the same Customer many times.

12.2.1 Eliminating redundant accesses

An enquiry may perform redundant processing, retrieve more entities than are necessary for the required output data flow. If the enquiry is infrequently made, you may assume the output data flow will be sorted and duplicates removed.

However, if the enquiry is a primary system function, triggered many times a day, you may perhaps prefer to refine the Entity model so that no redundant accesses are made, by adding a derivable entity into the Entity model.

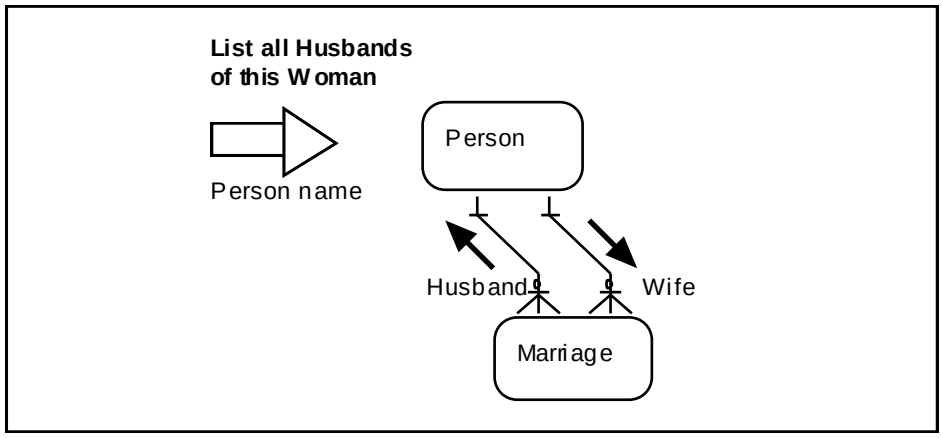


The new entity is not entirely a matter of performance optimisation. The fact that users enquire about 'Customer Interest in Product' so often shows that this associative entity (derivable though it may be) is a matter of concern to users in running their business.

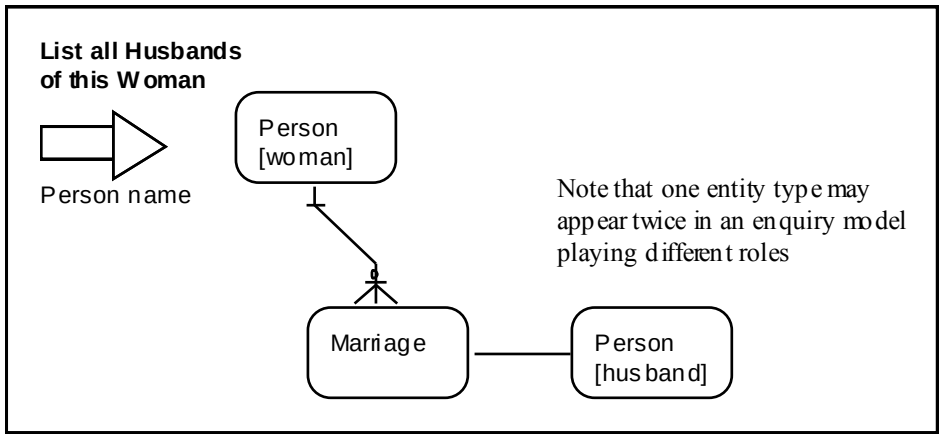
If you do include such an entity, make sure the text description of the entity starts with the word DERIVABLE, and name the requirements that is used for. Designers may choose either to store the entity as a database table, or write relatively complex enquiry processes.

12.2.2 Choosing between alternative paths

If there is only one route through the Entity model from the entry point, then the access path is obvious from the enquiry model. Otherwise, you have to specify which of several relationships are followed. You can draw arrows to show this.



Or you can draw the enquiry model from the perspective of the specific enquiry.

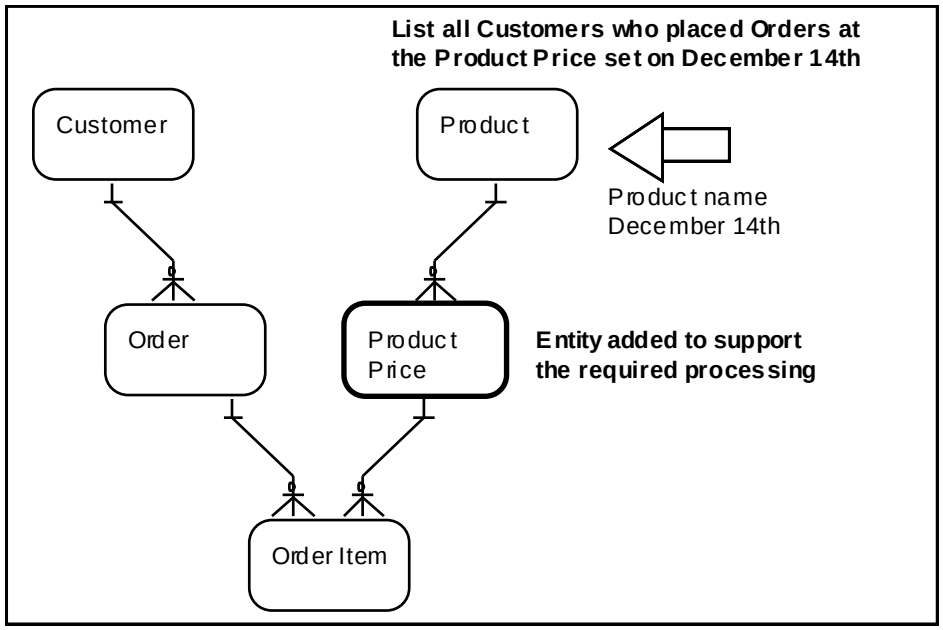


12.2.3 Specifying multiple entity roles

Note that one entity type may appear twice in an enquiry model playing different roles. One convention is the name the entity role in brackets after the entity name.

12.2.4 Historical reports

Don't forget management and audit reporting requirements. Wherever you find historical facts are needed on output,you should include historical attributes and relationships in the Entity model. Occasionally, you might be led to include an extran entity, and restructure the entity model accordingly.



Triage in enquiry access path analysis

Only document those enquiry models that are not obvious from the specification of the output data flow and Entity model. Under pressure of time, analyse only the outputs of primary system functions.

12.3 An event model

An event is like an enquiry except that it updates one or more of the objects it hits.

>>

Events are more complex than enquiries. Events require more careful analysis. You can use object behaviour analysis techniques to analyse events and define the behaviour of each class as a state machine composed of event effects.

>>

12.4 Implementation of enquiry and event models in design

The volume 'Event modelling for enterprise applications' shows it is not far from an event model diagram to either a procedural program, or to object-oriented programming.

For example, suppose a recruitment consultant wants to discover which Jobs are available for an Applicant you must find out:

what Skills the Applicant has, and

what Skill Type each Skill is classified under.

what Jobs are available under that Skill Type.

The graphical representation below shows how the relationships provide the path to select the objects that satisfy the enquiry.

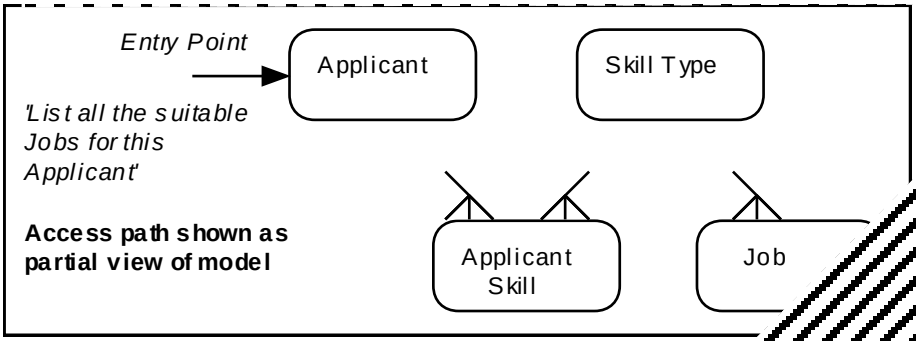


Fig. 7w

A CASE tool can mechanically convert figure 7w can into Figure 7x.

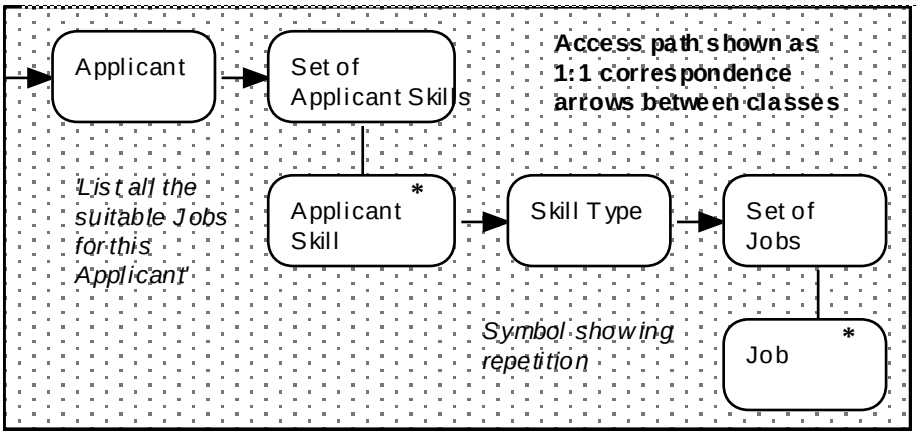


Fig. 7x

A CASE tool can mechanically convert figure 7x can into a Jackson-style program structure, with read statements allocated at the correct points.

The ability of an entity model to support an enquiry access path (however it is documented) is very important. The access path tells you which relationships are needed to select objects. It also enables you, in physical design, to design a database which has records, representing classes, stored so as to provide efficient paths for retrieving information.

12.5 Embedded systems

Our focus is on enterprise applications, but event modelling techniques are useful for other kinds of system - especially process control or embedded systems.

Methods for designing embedded systems normally focus on behaviour and process modelling techniques, and pay little or no attention to the entity model. But embedded systems do have an entity model.

The objects in a process control system may not be numerous or persistent enough to be stored in a database and connected therein by pointers. However, there is an entity model behind the scenes, and if you draw it, the model does tell you something. The relationships specify the paths along which objects may communicate.

13. More entity model shapes

How knowledge of patterns such as double-V shapes and diamonds can give productivity and quality benefits, helping people not just to draw entity models, but to get them right.

13.1 Triangles and diamonds

A class can participate in several relationships, either as parent or child. Fig. 1a shows a School is both the parent of many Pupils, and a child of one Local Authority.

The figure also shows three classes that appear in the shape of a triangle. This shape prompts an analysis question.

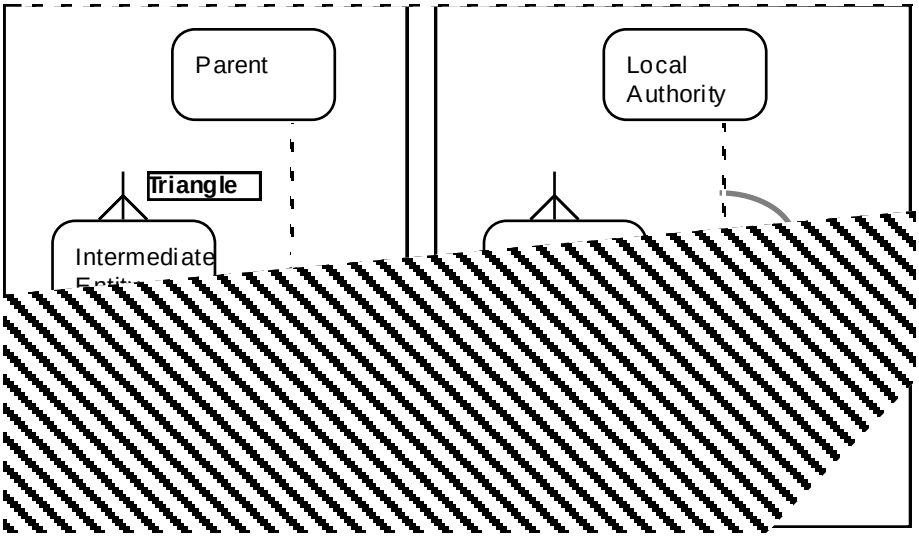


Fig. 1a

Ask of a triangle: For a given parent, are the same child objects discovered down the long direct relationship as down the two shorter indirect relationships?
If no, then keep the long relationship. You might need it because another relationship is optional at the child end, or because the bottom-level child has two different parents.
If yes, then remove the long relationship, even though it is a true statement.

Why? First, the long relationship is redundant; it says nothing that is not said without it. Second, it may wrongly permit the end-user to attach the bottom-level child to two different parents via the direct and indirect routes. Later sections discuss the question of double parents in triangles and diamonds.

13.1.1 Diamond shape or boundary clash

Fig. 1b shows that adding Teacher into the picture creates a diamond shape.

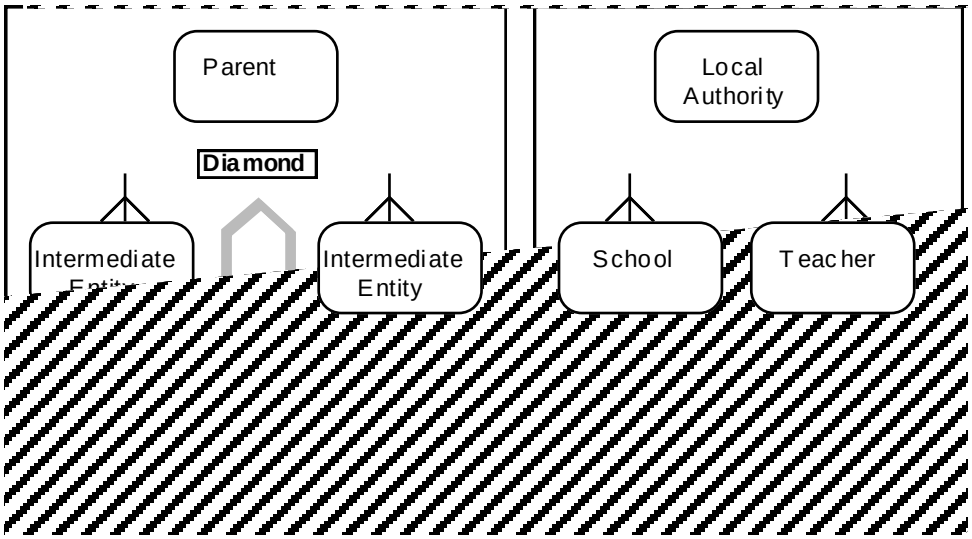


Fig. 1b

The two sides of the diamonds represent what might be called a 'boundary clash' between two conflicting ways for low-level objects to be grouped into a batch or collection.

13.2 Derivable sorting classes in Y shapes

Ask of a V shape: Can there be more than one link entity for one combination of the two parents? If yes, then consider transforming the V shape into a Y shape with a derivable sorting class at its heart.

E.g. Fig. 1c shows the Customer Interest in Stock class clusters all the Orders for the same combination of Customer and Stock.

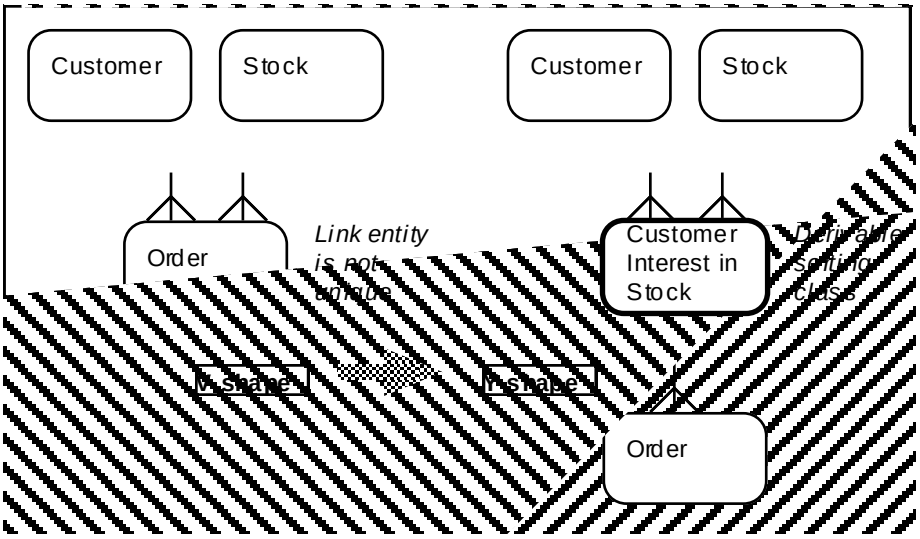


Fig. 1c

You may discover a key-only relation as a result of applying relational data analysis to an output report. For example, a report of Orders listed by Customer within Stock, or by Stock within Customer, may lead you to specify Customer Interest in Stock as a key-only relation or sorting class.

You may also discover a derivable sorting class when defining an access path to create such a report.

Some database designers are obsessed with removing all derivable data from the entity model, careless of the expense of redundant programming effort and processing time. The three-tier architecture gives us an opportunity to reexamine this assumption.

Entity classes in the Business services layer

If the users' requirement is for frequent reports that sort Orders by a combination of Customer and Stock, then the derivable sorting class surely belongs in the entity model. You can now specify simple enquiry processes that return the results the users want. You can code these enquiry processes in the Business services layer on the assumption that the derivable sorting class exists.

Soft classes in the Data services layer

What if the derivable sorting class is missing from the data storage structure? Perhaps the database designer rejects it, or you have inherited a legacy system without it?

This can complicate the specification or the coding of enquiries that generate the required reports. Since this complication is caused by the database designers' requirements, not by users' requirements, you should hide the complication from Business services layer processes, in the Data abstraction layer to the Data services layer.

The idea is that any enquiry process that wants to read a Customer-Interest-in-Stock object will call the Data abstraction layer to sort through the stored data, manufacture the object and return it. Such objects, manufactured by the Data abstraction layer rather than stored in the data storage structure, might be called 'soft objects'.

The notion that some derived data rightly belongs in the Business services layer runs against the received wisdom, so I return to soft objects and derived data in later chapters.

13.3 From double-V to Y shape constraint

Is there any similarity between the two entity models in Fig. 1d?

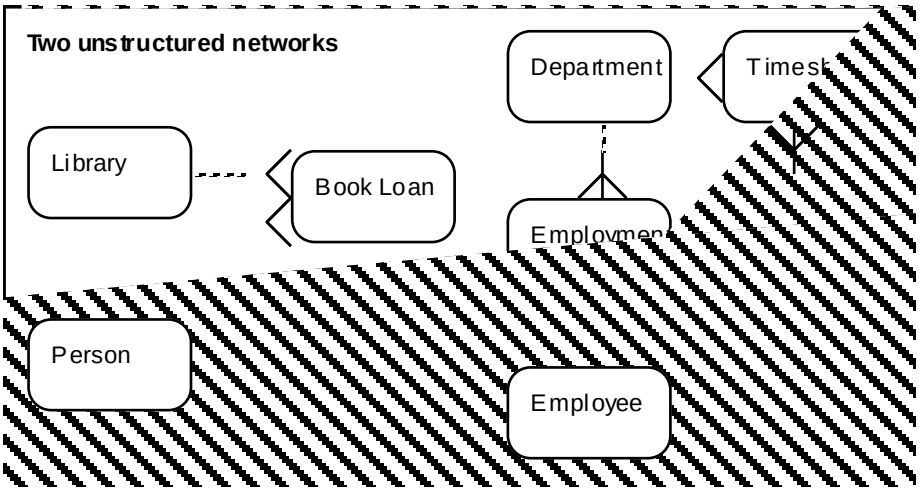


Fig. 1d

It is hard for us to spot that these unstructured models are two instances of the same general pattern. Tools don't care how messy the diagram looks, but people do.

A tool can help us by redrawing the models in a hierarchically structured fashion, placing the parent of each relationship above the child. If the analyst always draws the relationship from the parent to the child, and the tool constrains them to draw one-to-many relationships in this direction, then the tool can easily remember which end of a relationship is parent and which is child.

The analyst may request: 'Please reshape my diagram for me in a hierarchical fashion.' A tool can respond by redrawing the two models as in the Fig. 1e below.

(By the way, algorithms that try to avoid crossing lines become increasingly useless as the complexity of a network diagram grows.)

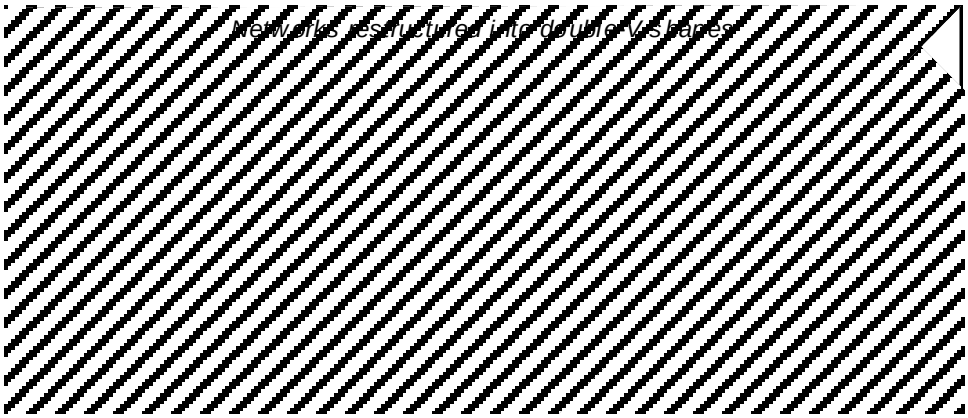


Fig. 1e

After a person or a tool has rearranged the diagrams hierarchically, it is much easier to see these are both examples of the double-V shape shown in Fig. 1f.

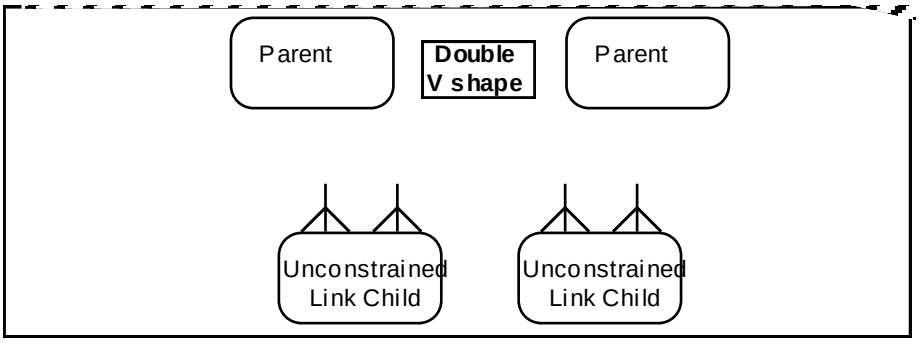


Fig. 1f

This shape is a generative pattern that prompts you to ask an analysis question.

13.3.1 Turning a double-V shape into a Y shape constraint

The analyst may request: 'Please highlight or report on any double-V shapes for me.' A tool might respond by thickening or colouring the questionable relationships, then ask the analyst the following question.

> Ask of a double V shape: Can you tie an object of one child entity to only one object of the other child entity? If yes, connect the two child entities by a relationship, to capture the constraint.

E.g. Fig. 1g shows a book can only be loaned to someone who is a member of a library; and a time sheet must be submitted by an employee within an employment.

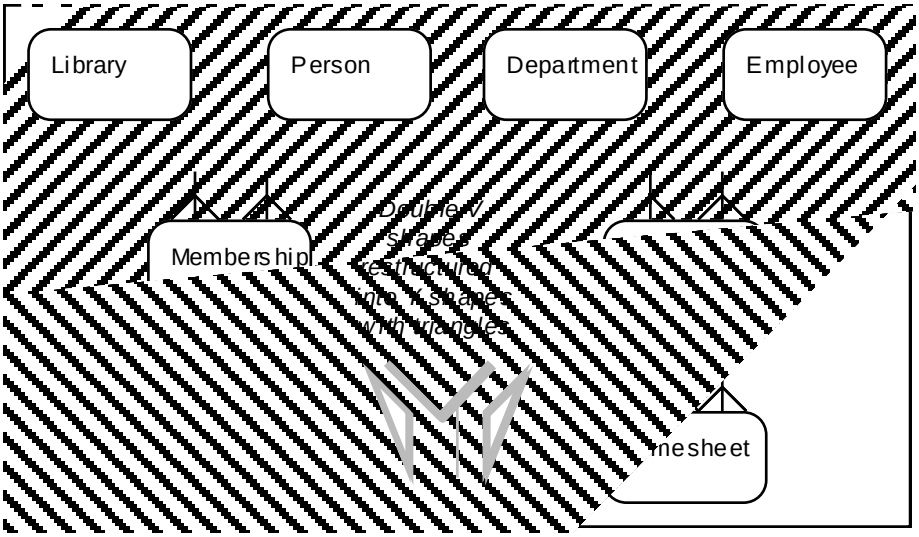


Fig. 1g

Hierarchical arrangement makes it easier to see triangles. After asking the earlier question about triangles, you are left with two Y-shape structures.

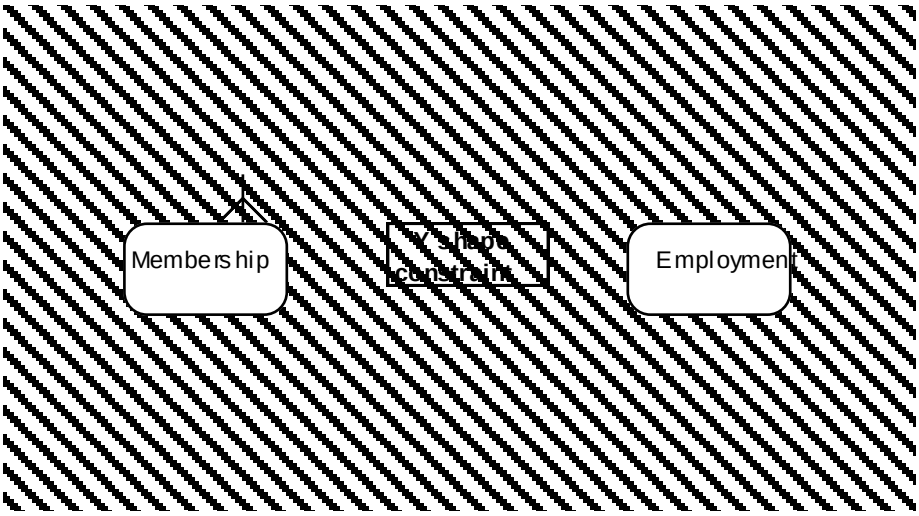


Fig. 1h

The classes at the heart of the Y shapes in Fig. 1h represent real-world entities. Users create objects of these classes in order to constrain the creation of objects of the class at the bottom. But there is another kind of Y shape.

13.3.2 Two kinds of Y shape

The examples so far have revealed two kinds of Y shape. Fig. 1i shows the class at the heart of the Y shape can be either a domain class or a derivable sorting class.

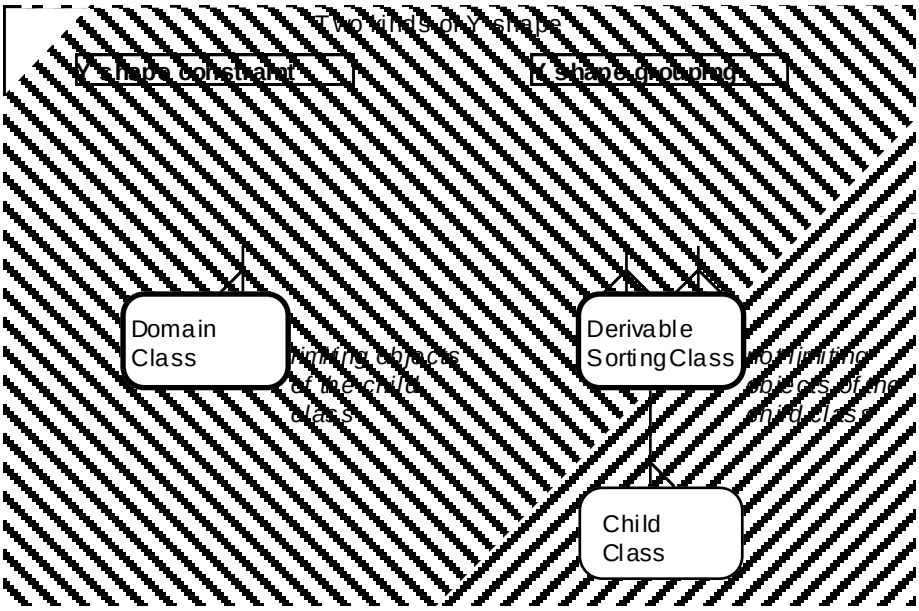


Fig. 1i

Objects of a domain class are created by users. The domain class at the heart of a Y shape might represent a business entity with attributes of its own (like Membership and Employment

in Fig. 1h), or it might be no more than a key-only link class that relates its two parents.

Objects of a derivable sorting class can be derived from the existence of child objects. An example of a derivable sorting class appears as part of the solution to the problem described in the next section. But first, a warning that some of our patterns can appear in disguise.

13.3.3 Patterns in disguise

The basic patterns or shapes can be obscured by intermediate classes. Fig. 1j includes a double-V shape, even though the Job class sits in the middle of one side of one of the two V shapes.

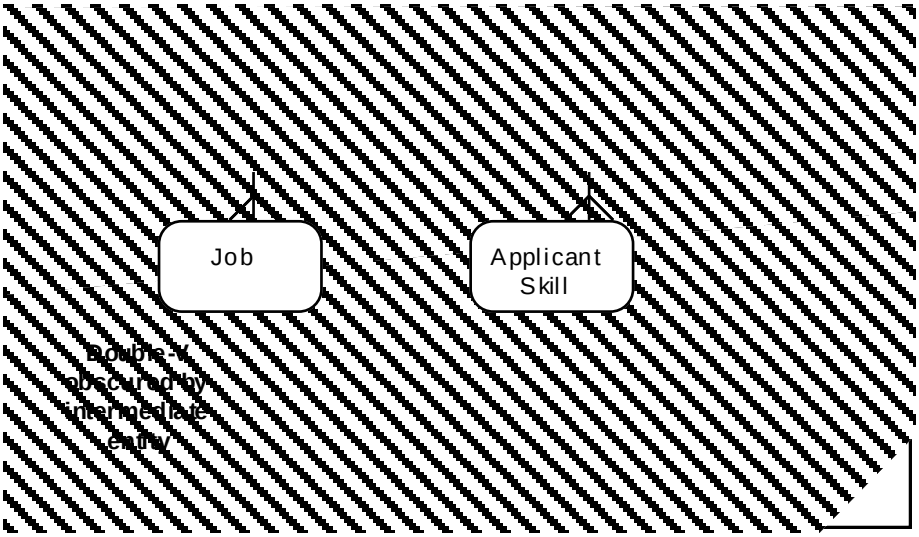


Fig. 1j

Readers may like to consider ways to resolve this double-V shape for themselves. A possible refinement of the structure appears later in this chapter.

13.4 Using patterns in quality assurance

The idea of teaching patterns is that analysts should save money by getting the system right first time. But the patterns are just as useful if you are trying to correct or improve a system that isn't working correctly. What follows is based closely on a real example.

The business has an enterprise application for recording what it does to meet customers' needs. The business supplies ingredients to food manufacturers. Ingredients are packaged in various ways, by size, quality and so on, to make distinct products, each with a distinct price. People ('Contacts' below) enquire about products. They may be sent a brochure and/or samples. They ask for quotes; they are given prices for specific products. They place purchase orders for a quantity of product at either the current price (an attribute of product) or the price given to them in an earlier quote.

The manager asked for our help. He had already set up a database, using an application generator, to record customers orders, and requests for information about products. Fig. 1k

shows the structure of the database.

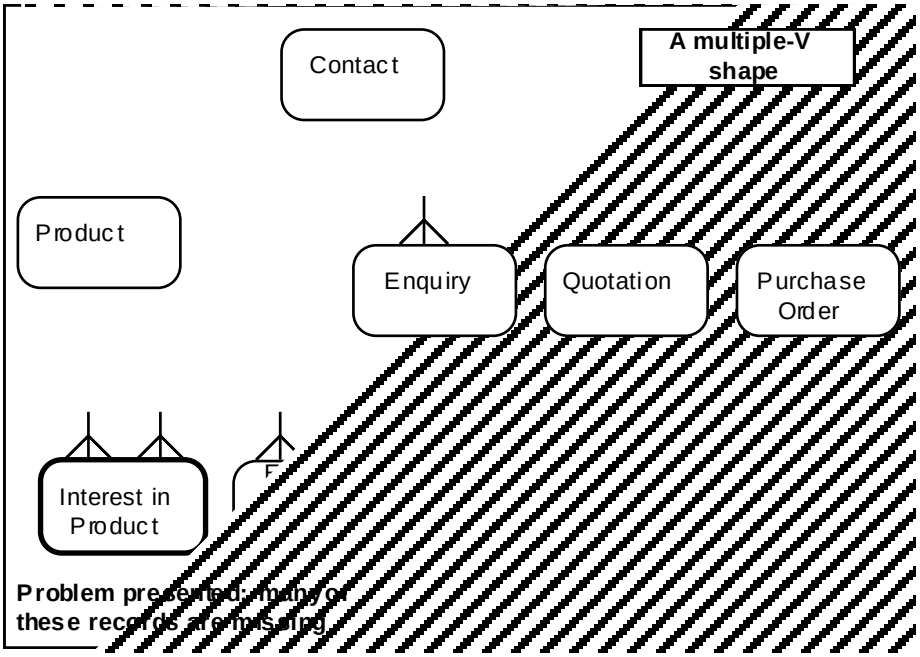


Fig. 1k

The manager had quickly generated a system to maintain this database, but problems were now being experienced with the quality of the information in it. The problems centred on the multiple-V shape, that is, the four child entities owned by the same two parents, Product and Contact.

13.4.1 First reported problem

The historical record of a contact's interest in a product was patchy, incomplete and out-of-date. Users forget, or cannot be bothered, to set up an Interest in Product record every time they record an Enquiry, Quote or Purchase Order.

Spotting the multiple-V shape prompts us to ask the question: Is an Interest in Product related to the various possible reasons for that interest?' Of course it is. Fig. 1l partially resolves the double-V shape by setting up explicit relationships in the data structure.

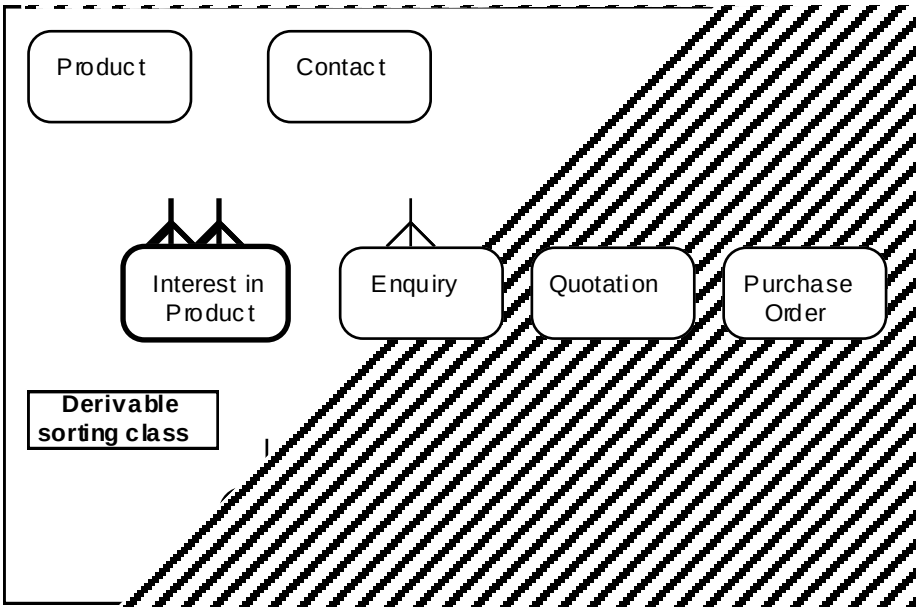


Fig. 1l

An Interest in Product record is now created automatically, whenever the detail of an Enquiry, Quote line or Purchase Order Line is recorded for a new combination of Contact and Product.

Note that the model does not match the pattern in Fig. 1i in one way; objects of the new derivable sorting class need not have any children.

Fig. 1m illustrates the transformation described in the volume ‘Introduction to rules and patterns’ whereby you might elaborate the model to show the rule that there must be at least one ‘Reason for Interest’.

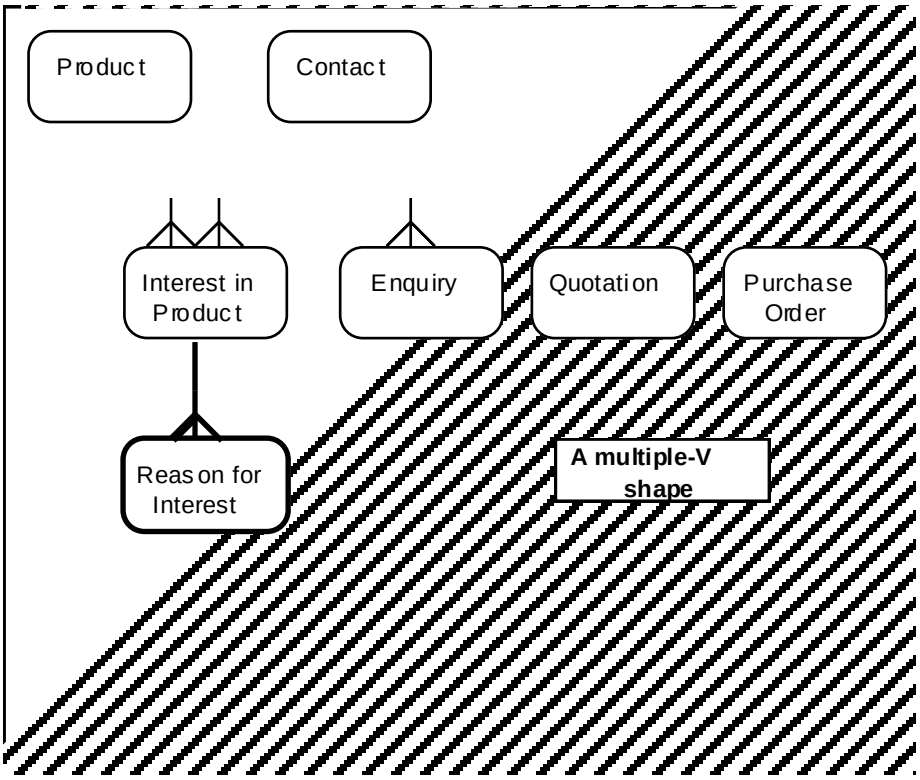


Fig. 1m

The trouble with introducing this rule is that it constrains us never to maintain an object of the class Interest in Product without a reason. The ‘at least one child’ rule is more rigid than is required by this business, so I will relax it again.

There is still a multiple-V shape in the model. The three bottom-level classes are all owned by both Product and Contact parents.

13.4.2 Second reported problem

End-users cannot record for historical analysis whether the price they give for an order line is the current price, or the price given on an earlier quote (they have some discretion to price order lines in either way). Also, they lose track of which quotes have been successful, that is, which quotes have resulted in orders. Fig. 1n resolves the multiple-V shape.

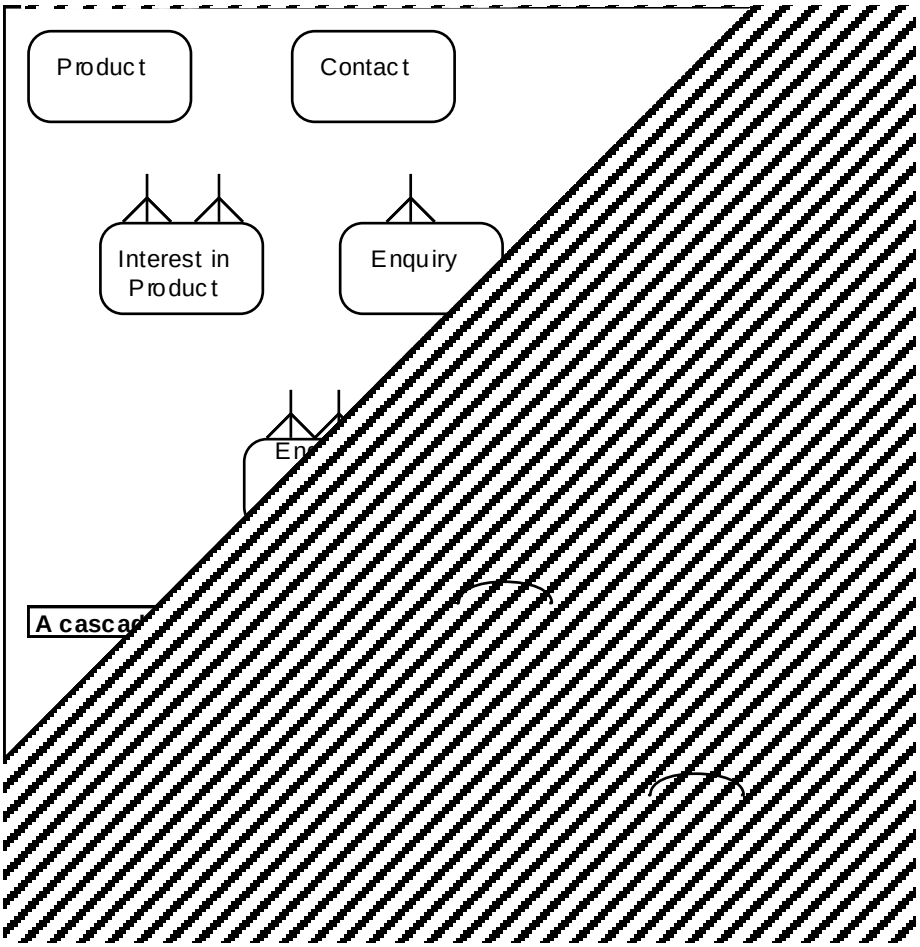


Fig. 1n

Further analysis of the child entities jointly owned by both Product and Contact may lead you to ask: Are users interested in whether a quote line results from an enquiry? or an enquiry led to a quote? or a quote resulted in an order? If so, you might add further relationships to the model. The exclusion arcs show that not all order lines come from quote lines, and not all quote lines stem from enquiries.

13.5 V shape domain classes in Y shapes

A third reported problem in the case study centers on another kind of pattern. The reported problem is that users are recording products in the database that cannot actually exist, products with impossible combinations of size and ingredient. Fig. 1o shows a pattern I call the relation shape.

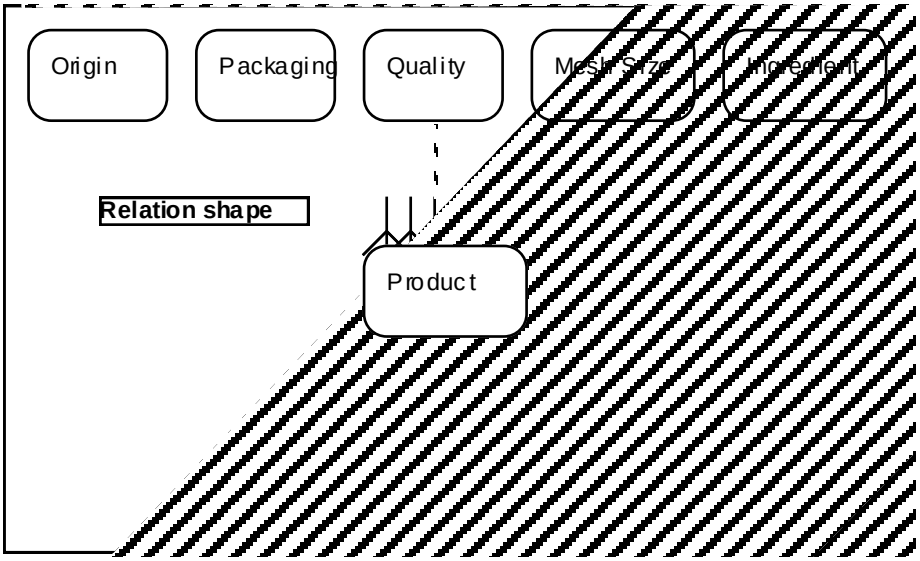


Fig. 1o

> Ask of a relation: Will users control the valid combinations of different attributes? If yes, then create a V shape from the domain classes.

Fig. 1p introduces a V shape domain class.

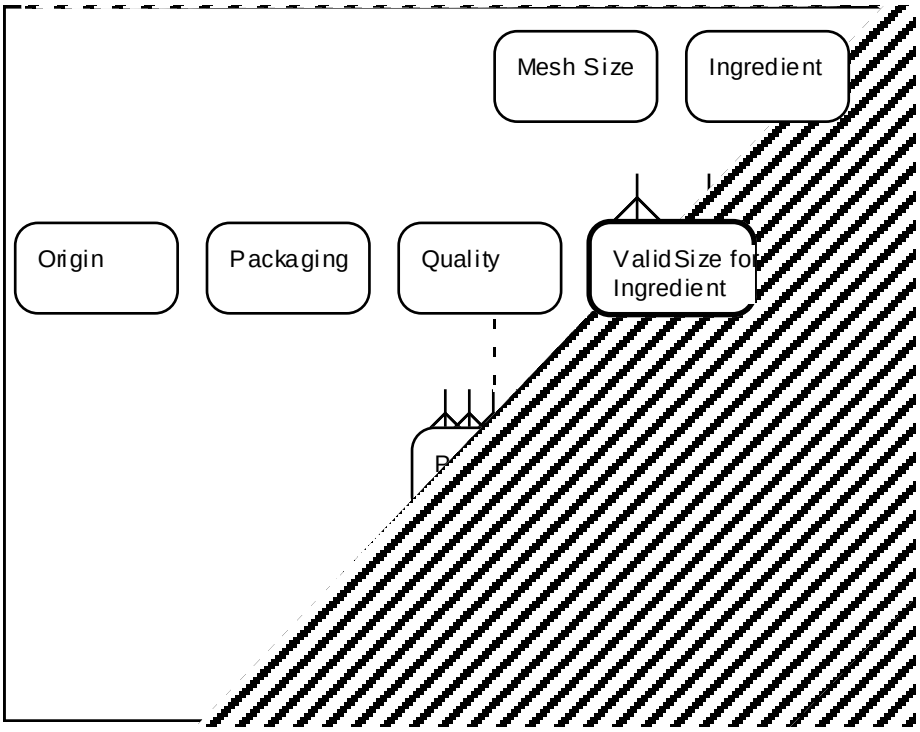


Fig. 1p

The introduction of a V shape domain class above the relation currently looks like the best design option for most applications.

13.6 Triangles and double parents

It is important to realise that not all triangular or double-V shapes are bad. It would be a mistake for a tool to automatically remove all such structures from a specification. Below are three cases where a triangle is a valid structure.

Children with optional parents

Fig. 1q shows a triangle that is valid because one of the short indirect relationships is optional at the bottom end.

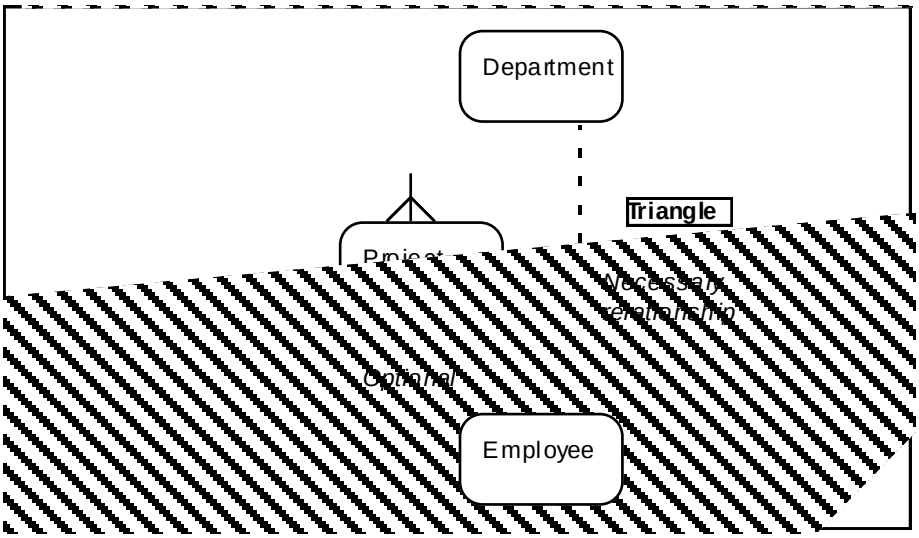


Fig. 1q

This case is well-known and has been illustrated by many others. Cases where all relationships are mandatory at the bottom end are more interesting.

13.6.1 Current and historic relationships

Fig. 1r shows triangle that is valid because there is a current 1:N relationship in parallel a historic N:N relationship. The current relationship to the link class is monochronous (one at a time); the historic relationship to the link class is polychronous (several at a time).

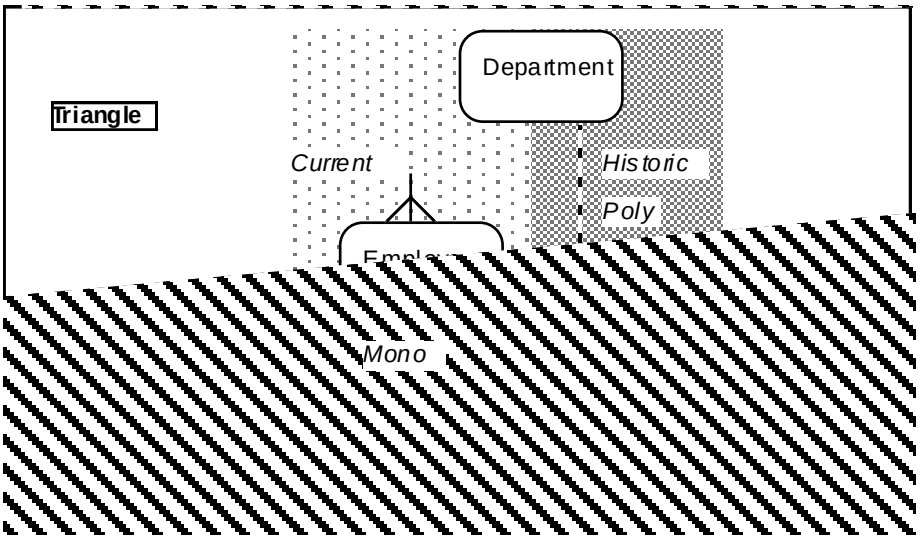


Fig. 1r

Some argue the current relationship is redundant because it is a subset of the historic relationship; but removing the current relationship creates redundant processing.

Without it, to find the current Department of an Employee you have to hunt through the historic memberships for the latest one, and then perhaps check that is still active. This redundant processing is avoided by making the current relationship explicit.

13.6.2 Double parents

Fig. 1s shows a triangle that is valid because the bottom-level child may have two different top-level parents.

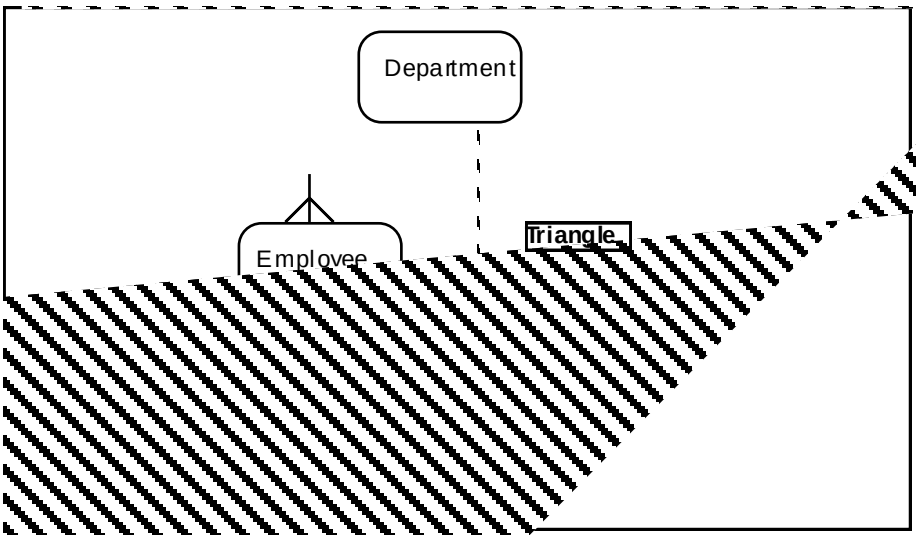


Fig. 1s

Ask of a triangle: Is the bottom-level object related to the same top-level object via both sides of the triangle?

Specifying the constraint that parents are the same

To say that a Task can only be done in the same Department that the Employee is contracted to, you should remove the long direct relationship.

Specifying no constraint.

To put no constraint on what Department a Task is done in, you can define the Task as having two Department attributes, one direct and one via Employee.

Specifying the constraint that parents are different.

To say (bizarrely) that a Task can never done in the same Department the Employee is contracted to - you specify the constraint by defining the Task with two Department attributes (foreign keys inherited by different routes) with the rule that these cannot match each other.

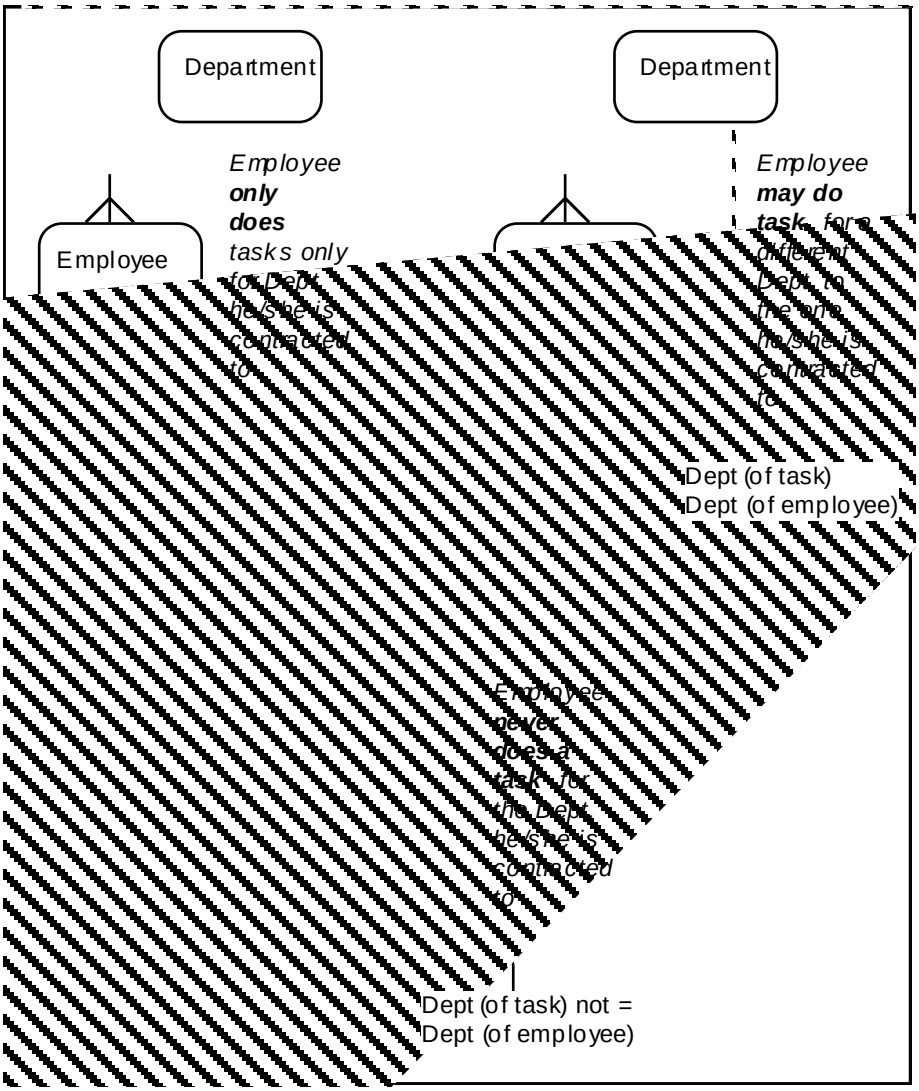


Fig. 1s

13.7 Single and multi-value constraints

There is another way to specify the last rule - that a Task can never done in the same Department the Employee is contracted to. Fig. 1u shows you introduce a V shape domain class.

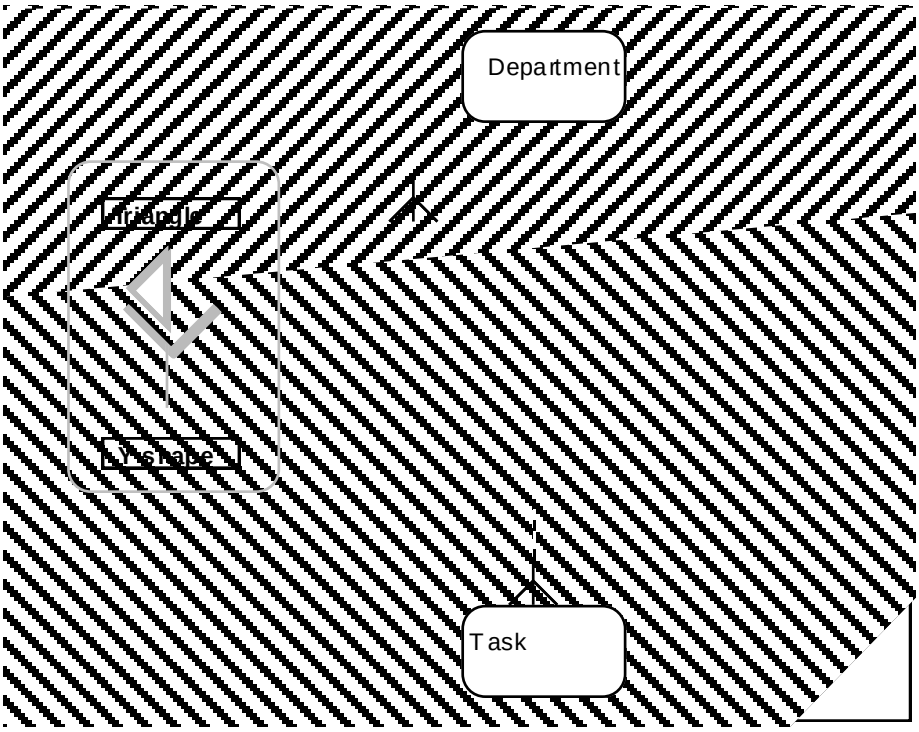


Fig. 1u

Introducing a V shape domain class in this case is an exceedingly clumsy solution, because all but one Department is valid for each Employee. Constraints that exclude a single value from a range are normally specified as a rule restricting the domain of an attribute of a class, as shown on the previous diagram.

However, multi-value constraints are normally better specified in the form of relationships. If there were a range of Departments for which an Employee is allowed to do a Task, then the structure above would be a good specification of this constraint.

13.8 Diamonds and double parents

You cannot remove any of the relationships in a diamond shape (unlike a triangle shape) without loss of information from the specification. But you should still

> Ask of a diamond shape: Is the bottom-level object related to the same top-level object via both sides of the diamond? If yes, you can specify this constraint by defining for the bottom-level object just one foreign key attribute identifying the top-level object. In section 2, a Pupil has just one Local Authority name. If no, you can specify this degree of freedom by defining for the bottom-level object two foreign key attributes, one for each of the top-level objects.

Fig. 1v shows a Fire Appliance can be related to two Counties: the County where the Incident is that the appliance is attending, and the County where the Fire Station is that the appliance is based at.

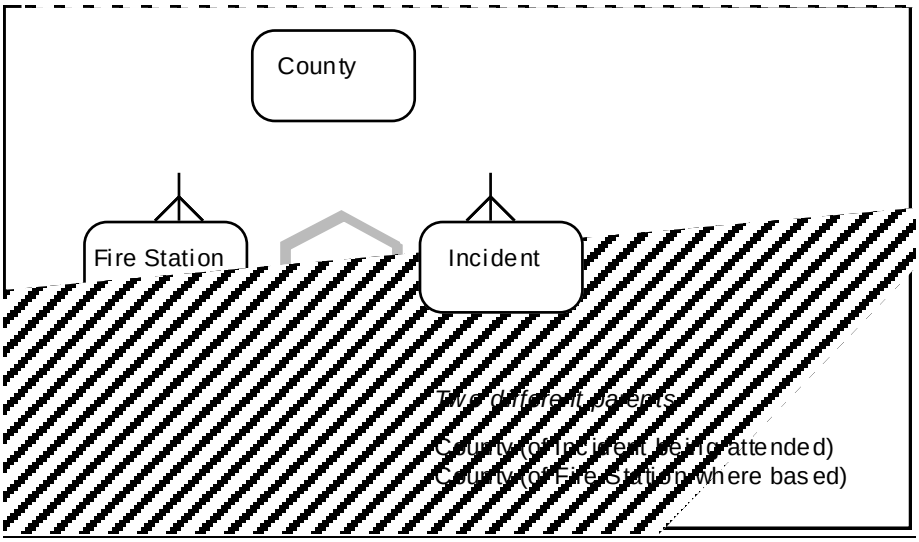


Fig. 1v

Fig. 1w shows the answer to the earlier question. It includes a diamond shape. So how do you specify whether the Interview has only one Skill Type, or may have two?

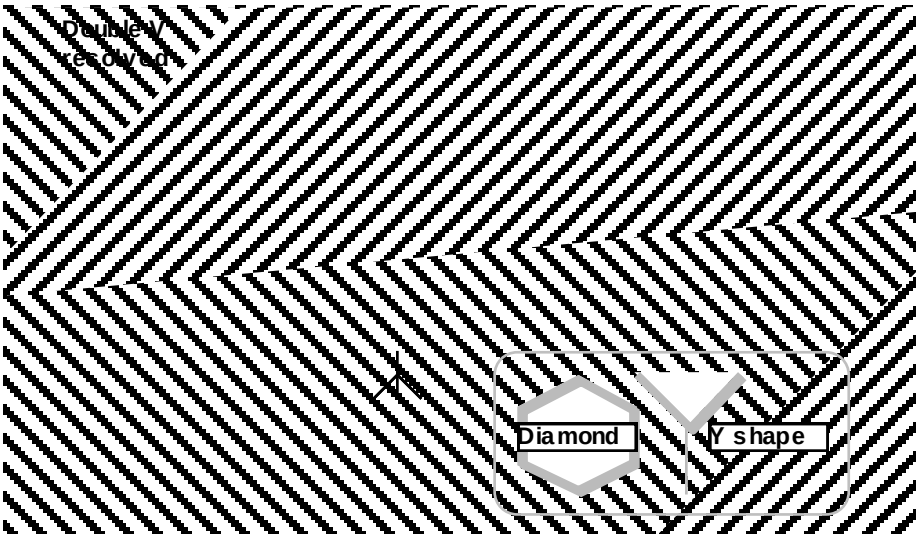


Fig.1w

Specifying the constraint that parents are the same

To say that an Interview can only be arranged for a qualified Applicant who has the same Skill Type as that of the Job, you can define the Interview as having only one Skill Type attribute (the same foreign key inherited by different routes).

Specifying no constraint

To say that there is no rule on whether an Applicant must be qualified for a Job or not, you can define the Interview as having two Skill Type attributes, without any constraint on their values.

Specifying the constraint that parents are different

To say (bizarrely) that an Interview can only be arranged for an *unqualified* Applicant, you can define the Interview with two Skill Type attributes (foreign keys inherited by different routes) with the rule that these cannot match each other.

13.8.1 Footnote

By the way, the classic example of a diamond shape or boundary clash is the 'Telegrams problem' described by Jackson (1975). Fig. 1x shows key elements of the specification. A paper tape is divided physically into blocks and logically into telegrams. Both blocks and telegrams are composed of characters grouped into words. A word may not span two blocks or two telegrams, it must be contained within one of each. The program must analyse the telegrams and print a report.

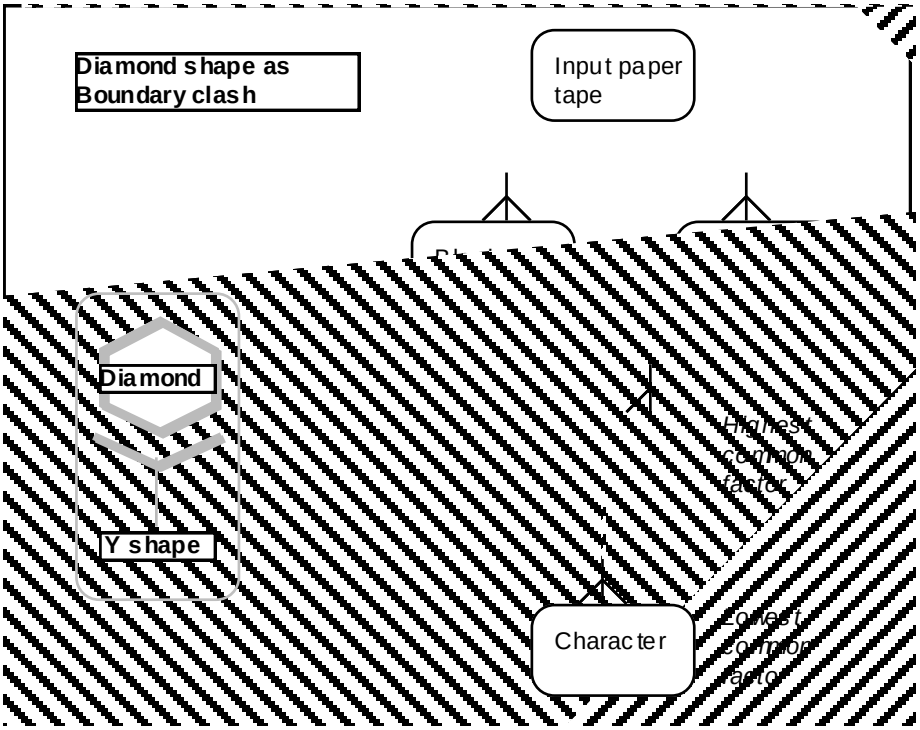


Fig. 1x

Jackson used the technique of structure clash resolution to design a system of communicating programs. The first program processed the input tape in terms of blocks, and wrote an intermediate file of words. The second program processed the intermediate file in terms of telegrams and produced an output report.

The need for this kind of two-pass serial file processing has been reduced by the introduction of network databases that can impose many clashing hierarchical structures on the underlying data. In terms of an entity model, Jackson's boundary clash appears as a diamond shape. The left-hand side is the input. The right-hand side is the intermediate file.

A later chapter discusses another design issue raised by the diamond shape - the possibility of a

process that travels from top to bottom, or vice-versa, via two different routes.

14. Advanced entity model shapes

Some relatively advanced techniques for analysing data structures, including reasons to contravene 4th and 5th normal forms by maintaining derivable sorting classes.

14.1 Analysis of compound keys

You may find, perhaps as a result of relational data analysis, that some classes have compound keys, but there are no parent entities with elements of the key.

> Ask of a class with a compound key, what classes exist with keys made out of its parts? Given a two-way compound key, then try transforming the class into a V shape with two parent entities.

Fig. 2a shows V shapes you can generate from the classes Holiday Feature and Client Requirement in a Travel Agency

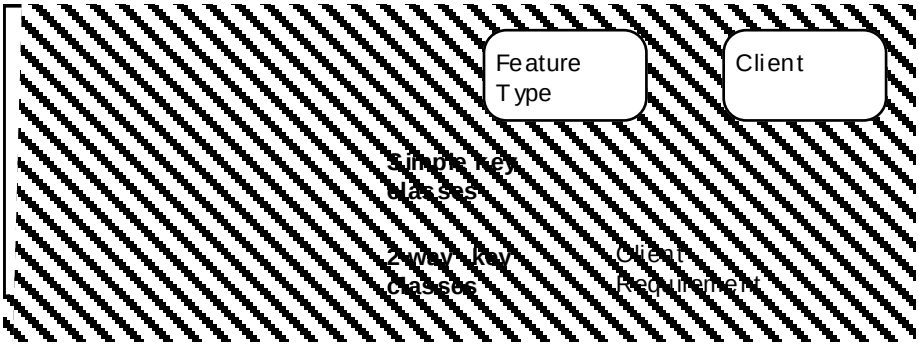


Fig. 2a

Fig. 2b shows V shapes you can generate from the classes Patient Admission and Employment Contract in a hospital system.

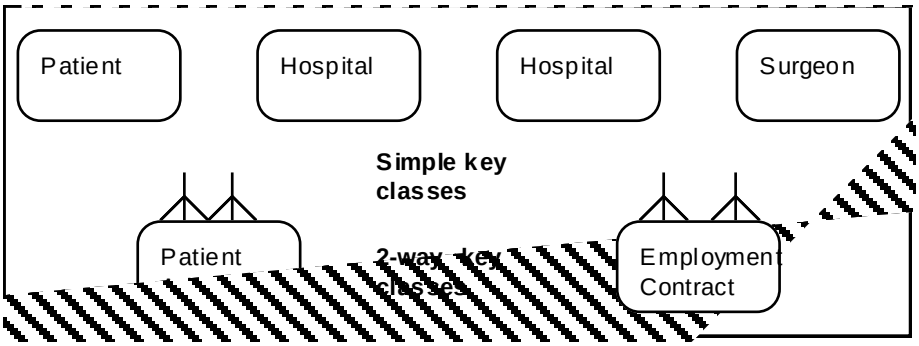
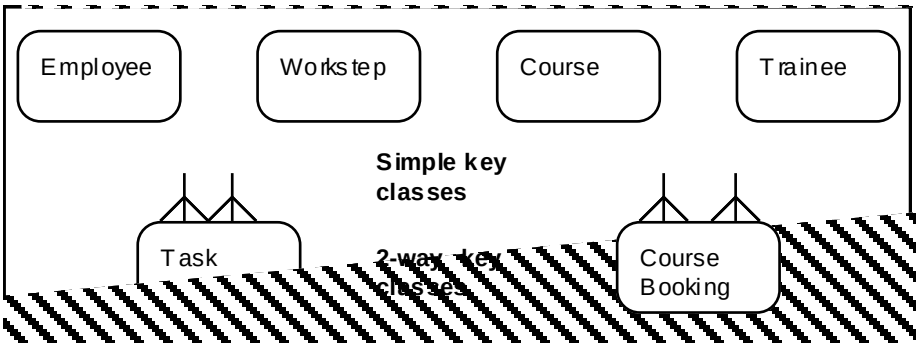


Fig. 2b

Fig. 2c shows V shapes you can generate from the classes Task and Course Booking in a personnel system.

Fig. 2c



Given a three-way compound key, then try transforming it into either a double Y shape or a triple Y shape, as shown below.

E.g. Suppose Surgical Operation has a compound key of Patient, Hospital and Surgeon (perhaps date and time ought to be included as well, but I shall gloss over this). You may draw a shape with three simple key classes, and two or three two-way key classes.

Assuming all Surgeons in a Hospital are allowed to operate on all Patients in the Hospital, analysis may reveal the classes shown in Fig. 2d.

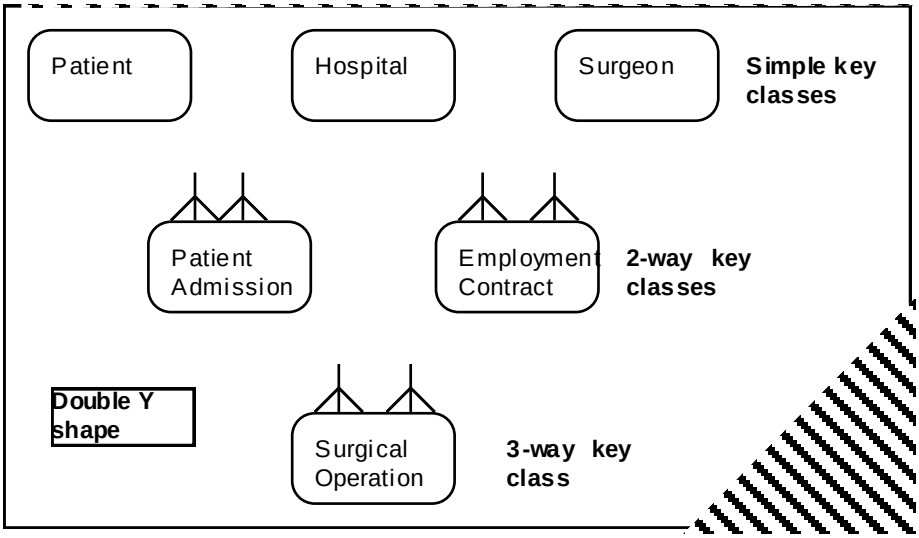


Fig. 2d

Fig. 2e introduces an extra class to model the constraint that Surgeons in a Hospital can only operate on a Patient in the Hospital after the Patient and Surgeon have both signed a consent form.

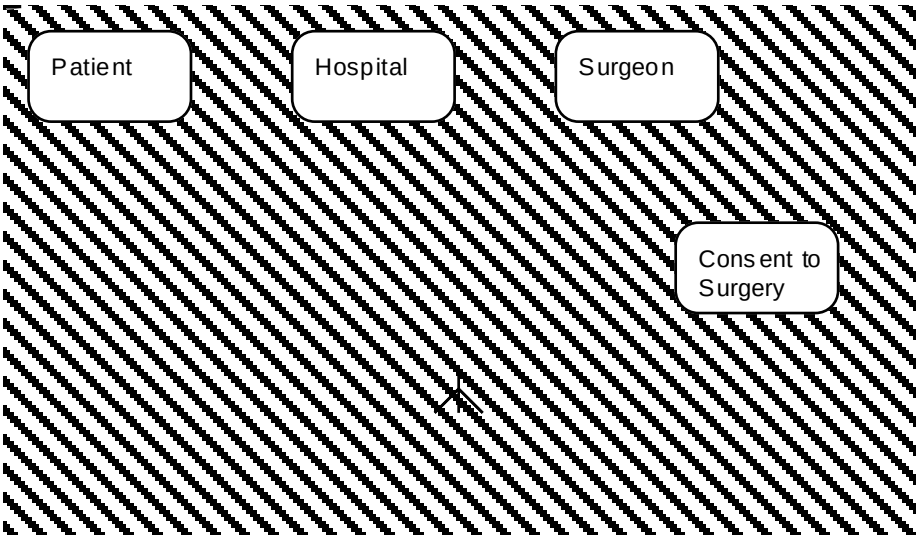


Fig. 2e

The three-way key class is necessary. A Surgical Operation records an event in the real world and it has attributes of its own. But some three-way key classes are redundant. They result from data analysis of a poorly designed input or output document, where there ought instead to be two or three two-way keys.

14.2 Double Y shapes

Reducing to fourth normal means replacing a derivable three-way key class by two classes with two-way keys. Fourth normal form is most easily explained in terms of a pattern.

> Ask of a double Y shape, does the class at the bottom: have only the keys of its parents (no additional attributes)? derive mechanically from joining its two parents? If yes, then the bottom class can be discarded, provided that the two parent entities are retained.

E.g. Fig. 2f shows that the Suitable Holiday class is merely a product of matching Holidays against Client Requirements. It can be derived at any time, and need not be placed in the model.

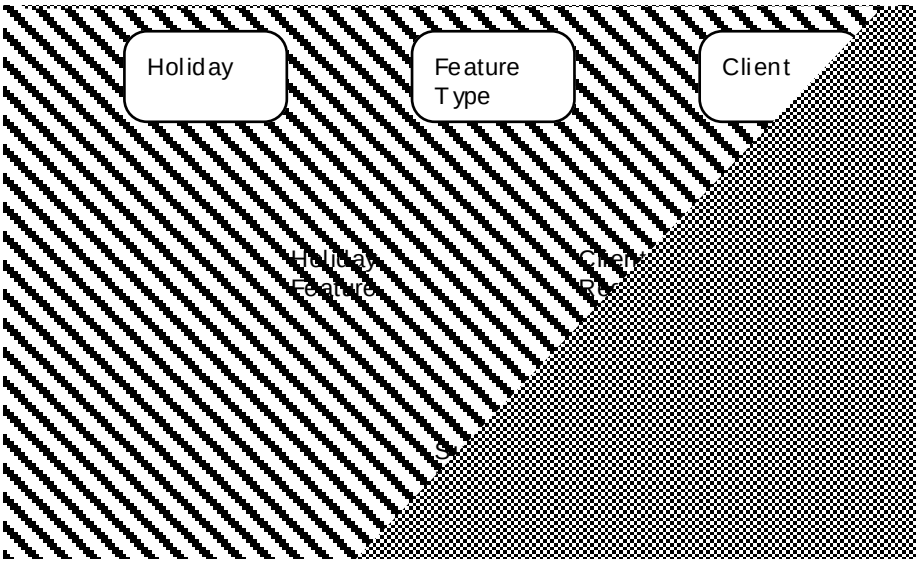


Fig. 2f

By way of contrast, Fig. 2g shows that the Holiday Booking class below is not merely a product of matching Holidays against Client Requirements. It is record of an event in the real world that users want to remember.

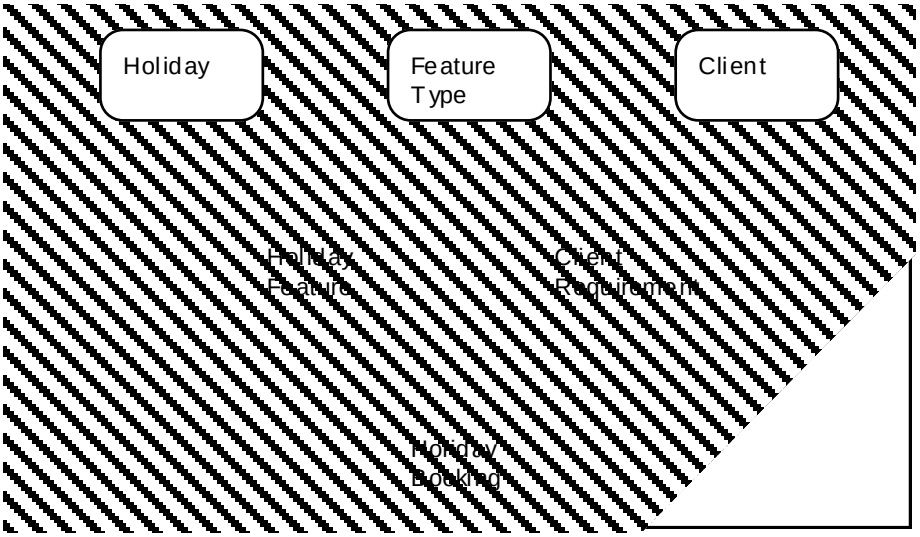


Fig. 2g

The Suitable Holiday and Holiday Booking classes give rise to a double V shape, resolvable in the normal way, as shown later.

A double Y shape may be incomplete at the top. Its essence is the V at the bottom - a class with unique compound of three attributes that appear in parent entities as unique two-way compounds.

E.g. Fig. 2h shows that if each Holiday is defined with only one Feature, the pattern is different.

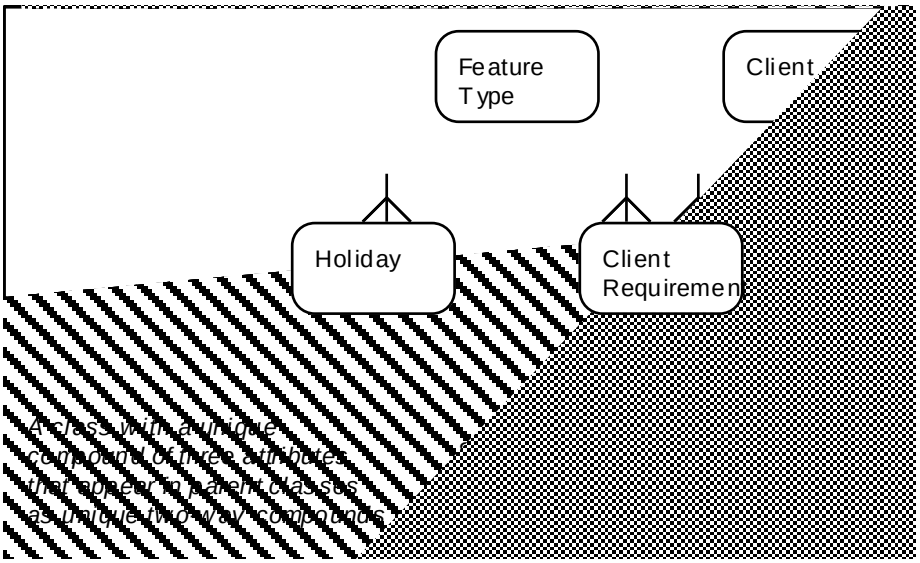


Fig. 2h

14.3 Triple Y shapes

Reducing to fifth normal means replacing a derivable three-way key class by *three* classes with two-way keys, where there is a 'join dependency' preventing all possible combinations of the key values from existing. Again, fifth normal form is most easily explained in terms of a pattern. Ask of a triple Y shape, does the class at the bottom:
• have only the keys of its parents (no additional attributes)
• not derive mechanically from joining any two parents?
• derive mechanically only from joining all three parents?
If yes, then the bottom class can be discarded, provided that its three parent entities are retained.

E.g. Fig. 2i shows that the Suitable Holiday class below is not merely a product of matching Holidays against Client Requirements. It is constrained also by the need for the Client to express an interest in the Holiday.

It can be derived from joining all three parents, and may be discarded from the model.

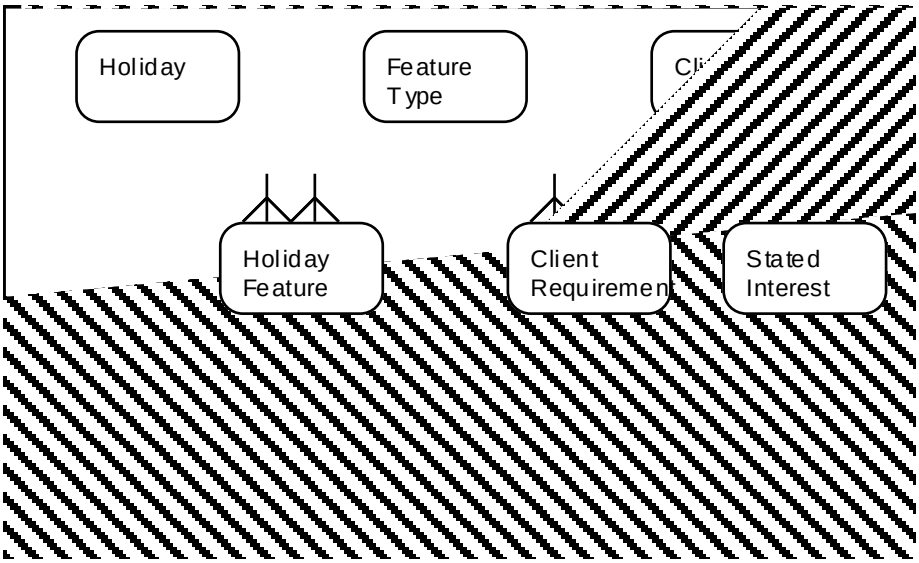


Fig. 2i

14.4 Contravening 4th and 5th normal forms

Designers are familiar with tradeoffs between:

• minimising redundant processing *versus* minimising redundant data

• simplifying enquiry processes *versus* simplifying update processes.

Theoreticians tend to advocate the latter option in each case. They say to eliminate all redundant data, including derivable key-only classes, and to minimise update processing. They don't say these options may conflict with each other. Consider the derivable sorting class called Suitable Holiday in the entity model below.

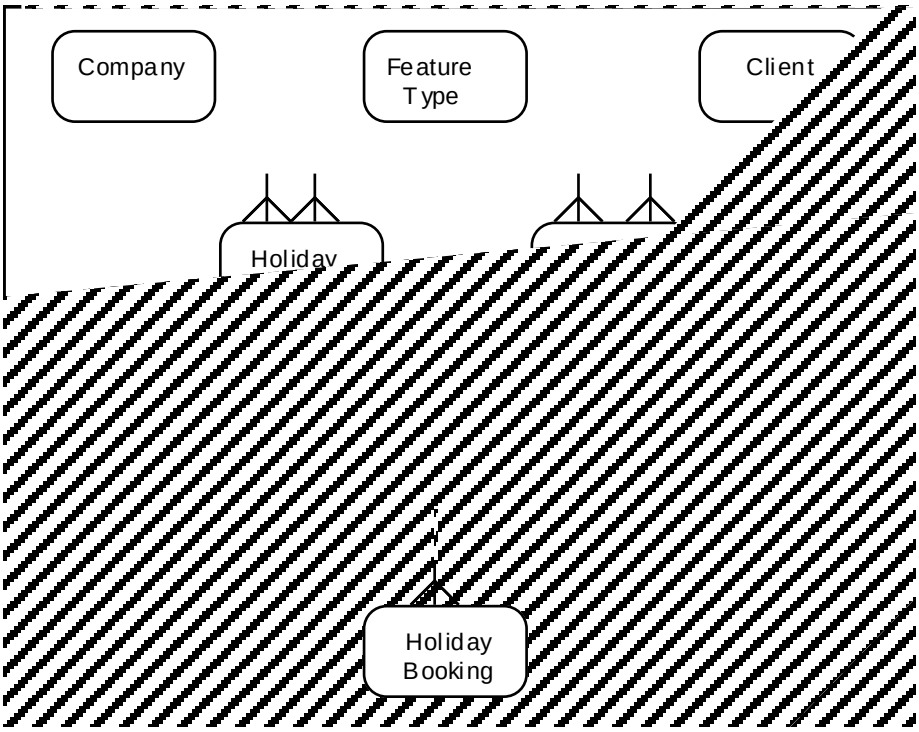


Fig. 2j

Since the system is designed to produce reports of Holidays suited to Clients, and reports of Clients suited to Holidays, the derivable sorting class will be useful.

Obviously, it will simplify and speed up enquiry processes. Without it, you will repeatedly have to manufacture Suitable Holiday objects in views of the data structure that users request for presentation. And you might have to account in some way for earlier Holiday Bookings on a Suitable Holiday.

Less clearly, the Suitable Holiday class can also simplify and speed up update processes. When a Holiday Booking is made, you can more easily check any history of previous Holiday Bookings. When a Client makes a Holiday Booking, you can more easily locate and check any Holiday Booking already made for the same compound of Client and Holiday.

Overall, it may prove cheaper to maintain Suitable Holiday as a sorting class than to leave it out. This contravenes the established view of physical database design. See the chapter ‘Clashing entity models’ for discussion of how and why you might maintain a derivable sorting class in the entity model rather than the data storage structure.

14.5 Tramlines shape

Where a class has a list of similar attributes, you can generalise these attributes into a relationship. Fig. 2k shows an example drawn from Assenova and Johannesson [1996].

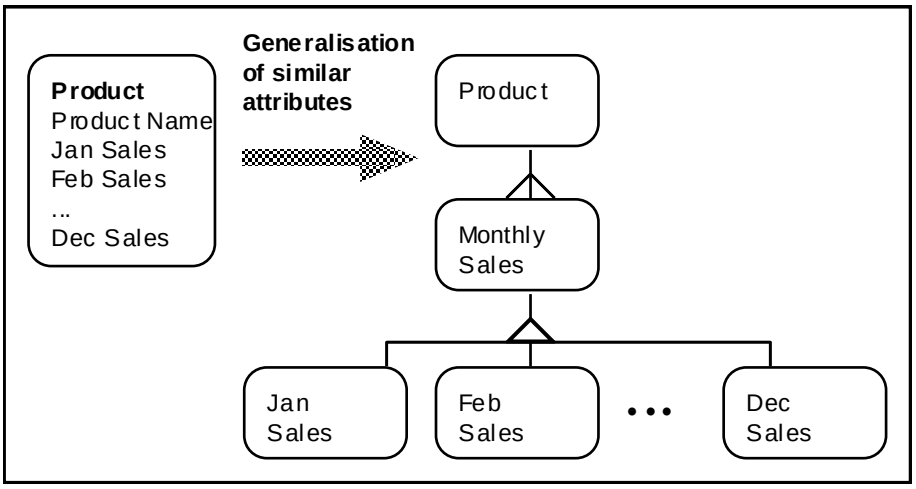


Fig. 2k

The transformation in Fig. 2k is not very common in practical system design. Fig. 2l shows two more common transformations.

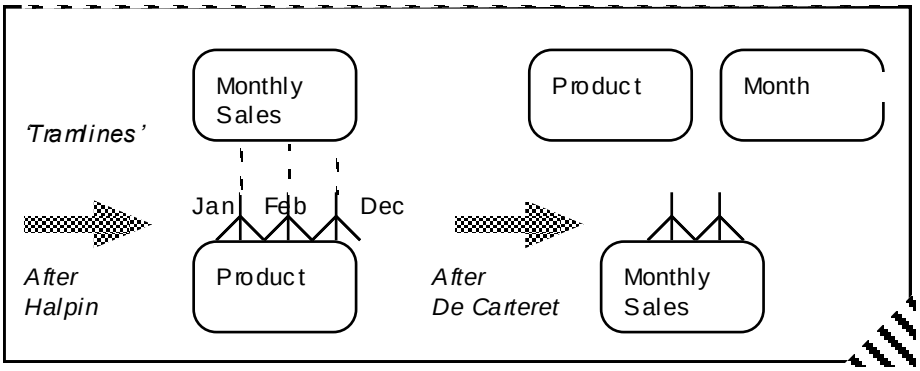


Fig. 2l

The patterns are discussed separately on the next page.

14.5.1 Creating a generalised attribute

Ask of a class with a list of similar attributes: can the attributes be generalised into a single type?

E.g. consider the three totals recorded in the Paper class in Fig. 2m. You might show the common properties of the three attributes by relating all three attributes to a single domain class. The resulting shape is called Tramlines.

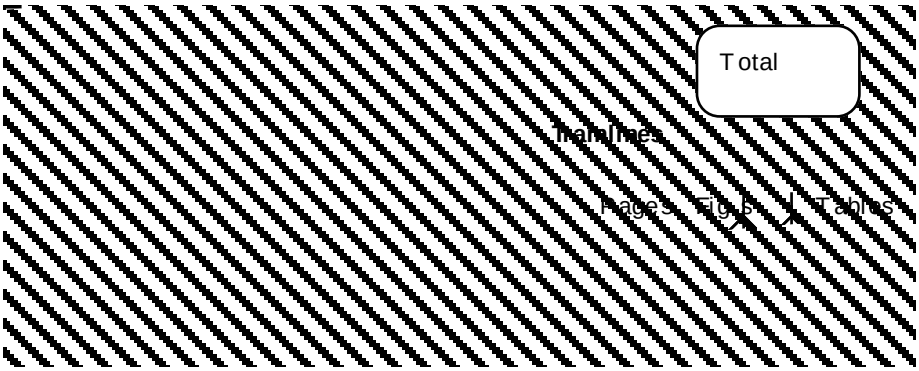


Fig.2m

14.5.2 Creating a generalised relationship

Ask of a tramlines shape: Can the relationships be generalised by creating a V shape with a child link class?

E.g. Fig. 2n shows the transformation of the tramlines in Fig. 2m.

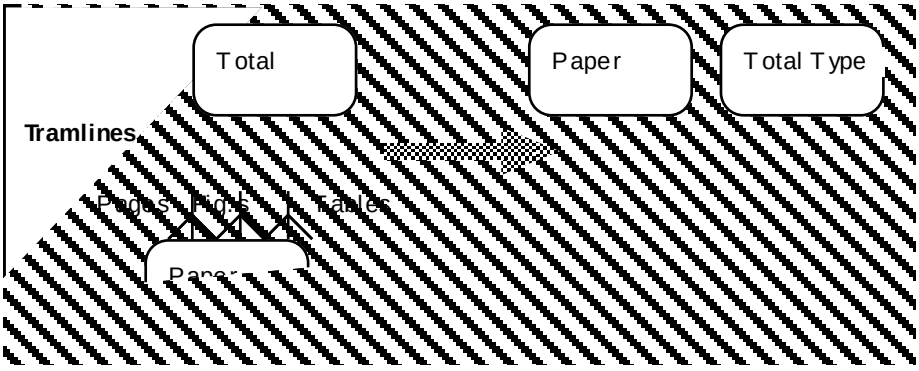


Fig. 2n

14.6 X Shape

Finally I come to a shape you often see in large business databases - a core entity, surrounded by many parents and many children. Fig. 2o shows this as the X shape.

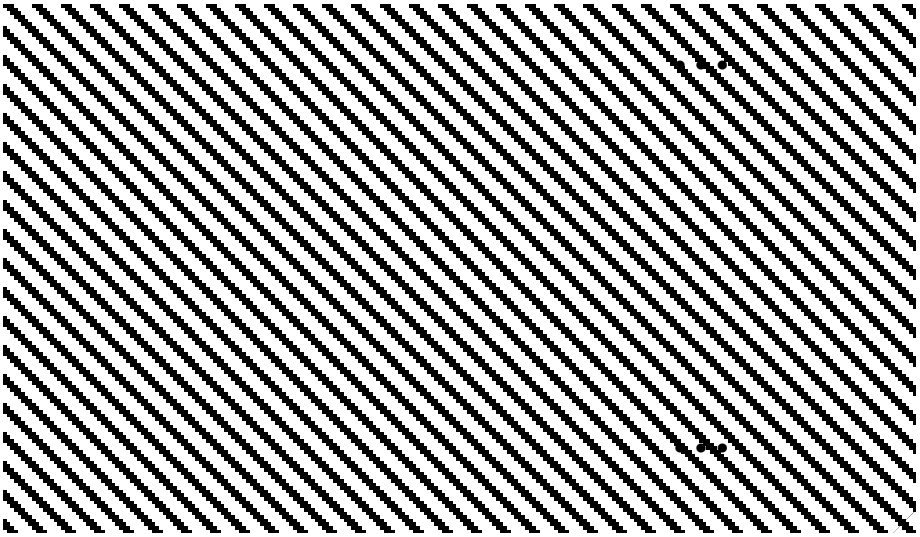


Fig. 2o

I call this the X shape (in line with the V,W and Y shapes) but you might better call it a star shape, since it can have many points, perhaps a dozen parents and a dozen children.

Ask of an X shape: Are there constraints between parents and children that are missing?

This is a rather vaguely-defined shape and a rather vague question. I don't say how many points the X shape must have before it is likely to reveal significant missing constraints. Nor do I prescribe what to do in response to the question. Further research may reveal further rules of thumb in this area.

15. Design for maintenance

Designing an entity model for maintenance, anticipation of amendments.

15.1 Correctness and maintainability

Surveys tell us that maintenance costs far outstrip initial development costs; 70-30 is a proportion often quoted. Some hold out 'design for maintenance' as a primary goal of system development.

Analysis patterns can help you to design for maintenance and facilitate amendments.

But maintainability cannot be the primary goal. Correctness must be the primary goal. You should strive to get the system right this time, not next time. If you don't strive, you won't succeed. And if you don't succeed, you'll have to spend more on 'maintenance' later.

Other surveys tell us it is cheaper to correct errors sooner rather than later. It is obviously much cheaper to revise analysis documentation than program code in a working system. So people have proposed ways of exposing errors in analysis and design as early as possible.

One way is to follow an analysis and design methodology that produces graphical design documentation. Current methodologies have many weaknesses. Above all, they lack effective quality assurance mechanisms. It is no use having paper mountains of analysis and design documentation if nobody can tell whether the documentation is any good or not, and programmers throw most of it away.

Analysis patterns provide a solution to this problem; they provide quality assurance questions.

Another way is to follow the path of 'iterative development', rapidly producing prototypes of parts of the required system. Prototyping makes design results more concrete, more visible, so you can more easily see if designers are going in the wrong direction and head them off.

An enterprise application will not be entirely right first time. Some amount of trial and error is necessary. Some amount of iterative development is inevitable. But setting out with the *objective* of delivering a wrong system, then developing it by trial and error, is likely to add time and costs to the overall project.

Iterative development stretches the costs of development over smaller cycles. In effect, it moves maintenance (which we know to be expensive) into the development phase. Change control and configuration management become bigger issues. So if you iterate more than a a couple of times, the overall project will cost more and take longer.

Iterative development runs counter to design for maintenance. Designers who are focussed on the next small increment won't take a long-term view. The code will grow haphazardly with each iteration into a pile of spaghetti that is hard to maintain. Agilists consequently promote "refactoring". And of course, good design up front will reduce refactoring costs.

Iterative development encourages low expectations. Designers who think it normal and acceptable to deliver unfinished code will not strive hard enough to get the system right before giving it to users. Designers have an excuse to escape from their responsibility to do their best work.

Is there a credible way to improve on iterative development? Current methodologies are failing us. We lack a methodology that embodies professional expertise about designing for

correctness and designing for maintenance.

Specification and design patterns address this problem; they encourage right-first-time design and can reduce maintenance effort.

15.2 Changes in requirements

In one sense there is no such thing as maintenance, there is only further development. The things you have to do in maintenance are the same as you have to do in development.

If it means anything, design for maintenance means designing the current system in a different way from how you would design it if no changes were ever expected.

It is meaningless to design for maintenance per se. Flexibility in every direction is impossible. Changes come from many different angles. You have to decide what changes are likely, and design with those changes in mind.

There are three basic design for maintenance strategies.

15.2.1 Separating concerns

Some changes are due to new technology, perhaps a new database management system or new user interface management system. The way to anticipate changes in technology is to isolate, as far as possible, those parts of a system that are technology-specific.

Other changes are due to new user requirements: people changing their mind about the way they want to system to operate. New user requirements may be subdivided into ‘correctness’ requirements and ‘usability’ requirements. The way to anticipate changes in these requirements is to isolate, as far as possible, those parts of a system that are specific to specific kinds of requirement.

You can separate these concerns using the high-level analysis pattern of the 3-schema architecture. This architecture is divides an enterprise application into subsystems that isolate different areas you may want to change.

Software layer	User concern	Technology concern
UI layer	user-friendliness	GUI environment
Business services layer	business rules and constraints	App server
Data services layer	Performance	database management system

15.2.2 Loosening constraints to accommodate exceptions

You can anticipate exceptional cases by not constraining the system to accept only normal cases. This can prove counter productive. Some of the tradeoffs are discussed in the next section.

15.2.3 Generalising the design

You can generalise the design so that it is easier to accommodate new cases and reconFig.

the system with new rules. See the section after next.

15.3 Loosening constraints to accommodate exceptions

After you have specified constraints within the Business services layer of code, you may find you have to relax them to deal with exceptions. Users often submit maintenance requests asking for the freedom to break the normal rules, record unusual cases not previously envisaged.

A natural reaction is to relax the constraints on datan entry. This increases the danger of incorrect system usage and gives users the opportunity to screw up the system. Users need a system that constrains datan entry, prevents garbage from being stored in the database.

The trouble is that to design for exceptions, rather than reduce the constraints on the current system, can make the system considerable more complex.

15.3.1 Specifying constraints on the normal case

Fig. 3a shows the entity model of the Marriage Registration system, introduced in the volume 'Introduction to rules and patterns'.

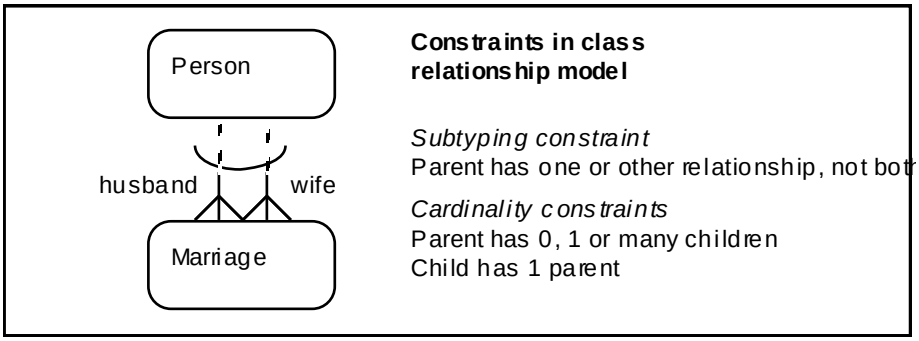


Fig. 3a

One problem might be that changing a Person's recorded sex would automatically invalidate all previous Marriage of that Person. All historic Marriages for that Person would now be in a state inconsistent with the rules of the system - recorded as being between two people of the same sex.

The solution is to record a Person's sex of birth separately from their current sex, and apply the validation constraint only to their sex of birth.

This tiny Marriage Registration system has caused much debate in our tutorials, on grounds ranging from design and coding style, to culture and political correctness. Please don't be offended if I go on to illustrate laws and societies you disapprove of.

An exclusion arc over the relationships implies that the class at the focus of the arc may be divided into subtypes. In this case, the two subtypes are man and woman.

15.3.2 Tightening constraints on the normal case

Suppose the Marriage Registration system is bought by a country where sex changes are

illegal and unrecognised. You might specify a fixed class hierarchy as in Fig. 3b.

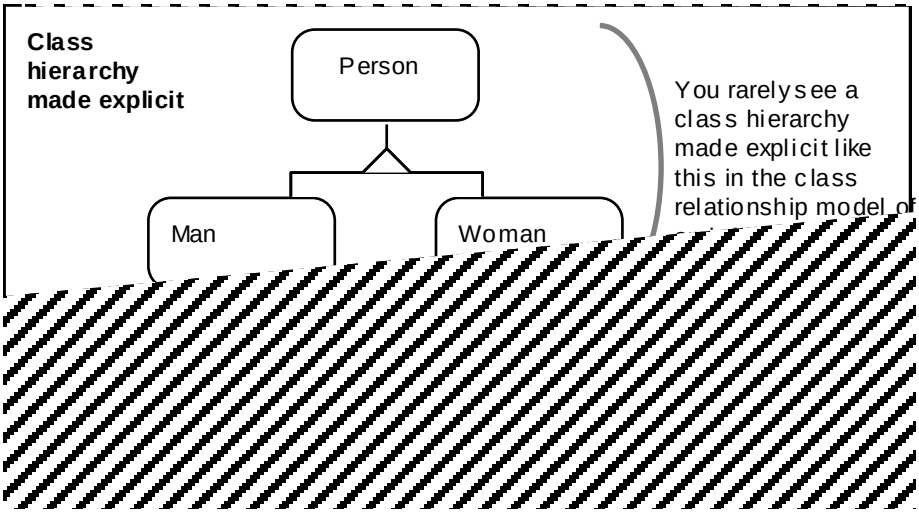


Fig. 3b

Types

It is normal for an entity state record to belong to many types. You might regard sex and job title as types of a Person.

Types are not normally represented as class hierarchies in the entity model of an enterprise application. One reason is that types often turn out to be additive rather than mutually exclusive - a Person can can have more than one job title at once - a bisexual Person might be recorded as having two sexes.

Another reason (the one that applies here) is that with the passage of time, an entity may change its 'type' many times. You may reasonably expect that most if not all of an entity's types can be altered during the life history of an object.

States

The longer an object persists, the more that a type (even one as fixed in real life as male or female) tends to become a temporary state.

I don't think of the object as changing class each time one of its types is updated. I think of it as remaining of the same class, but changing its state. Where a type change or state update constrains the future behaviour of an object, this is most naturally specified as a state-transition in the life history. So the type becomes a state variable.

You can specify the cyclical alternation between sex roles as state changes within the state machine of a Person. This state machine will record the current state, the current sex role, but not remember past ones.

15.3.3 Loosening constraints on the normal case

Suppose the system is bought by a country where transsexuals are allowed to contract a marriage in their new sex. After a few months, the users submit an amendment request:

"Can we please be allowed to record the exceptional case where, over time, a Person plays both husband and wife roles in different Marriages?"

You might simply erase the exclusion arc constraint, as shown in Fig. 3c.

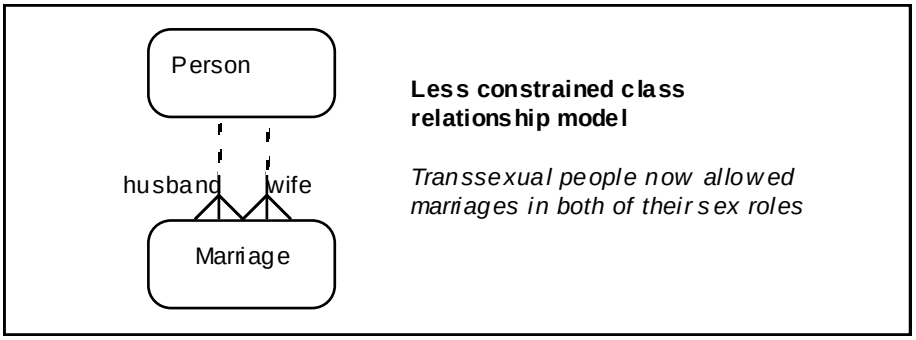


Fig. 3c

Is it worth removing the constraint on transsexuals remarrying under the new sex, just for the sake of just one or two individuals?

In general, relaxing a constraint may cause more trouble than it saves, by allowing some normal cases to be erroneously recorded as exceptions. You have to trade-off giving the end-users freedom to process rare cases, against specifying constraints that maintain the quality of stored data for the normal cases.

In this case, the danger is slight. A Marriage is still defined as connecting one Person of each sex. So users must change the recorded sex of a Person before they can record a Marriage under their new sex role. It would be difficult to do this by chance, in error.

Again however, changing a Person's recorded sex would automatically invalidate all previous Marriage of the same Person. These would now be in a state inconsistent with the rules - recorded as being between two Persons of the same sex.

The previous solution, of recording a Person's sex at birth separately from their current sex, only works if a Person can only change sex once. The proper solution is to record the life history of a Person's sexual roles, and attach each Marriage to the period of time that they play a given sex role.

15.3.4 Adding a history of roles

To keep a history of a Person's sex changes, and record the Marriages contracted within each sex role, you should extend the entity model as in Fig. 3d.

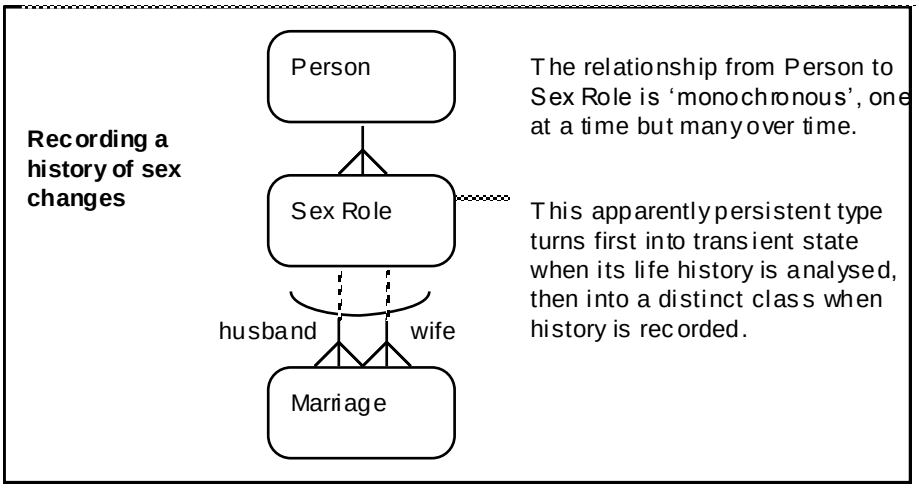


Fig. 3d

Is it worth enriching the specification to record history? Yes if the user wants to be able to inspect past occurrences of an object's state. Yes if it helps you maintain the constraints on system behaviour. It is now possible to change a Person's sex and record new Marriages, without invalidating all their previous Marriages.

15.3.5 Distinguishing exceptions from the normal case

Can you have it both ways? Can you place constraints on the normal cases, yet also give end-users the freedom to process rare cases? Yes, but at considerable expense.

You might design the user interface so that the user is presented by default with the normal case - the possibility of entering a Marriage between two people in their sex at birth. To enter an exceptional case, the user must make a conscious effort to pop up a menu and select an entry for entering an exceptional case - Marriages involving one or more Transsexuals.

Fig. 3e enriches the entity model to show all possible valid types of Marriage as distinct classes.

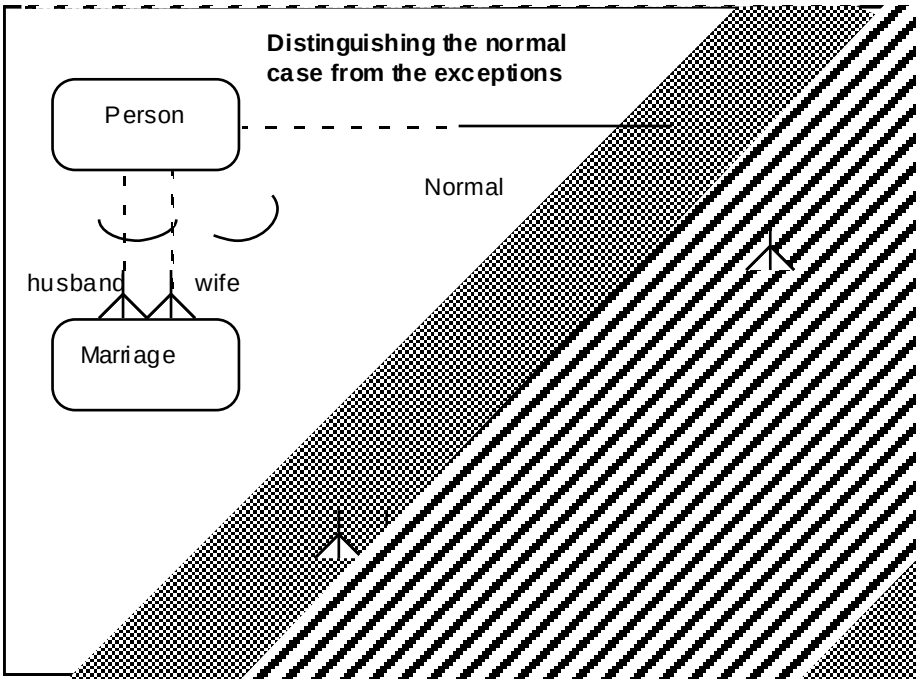


Fig. 3e

In Fig. 3e, more than 50% of the design effort is devoted to handling what are likely to much less than 1% of the cases.

Is it worth enriching the specification to distinguish normal cases from exceptions? You should present the development costs for users to decide.

15.3.6 Allowing user to define their own rules

So far, users must change the recorded sex of a person before they can record a marriage under their new sex role, since a marriage is still defined as having one partner of each sex.

You can anticipate more exceptional cases by giving control over the system's rules to the users. Fig. 3f generalises the specification so that users can define new kinds of marriage, with new combination of sexes, or even more than two people.

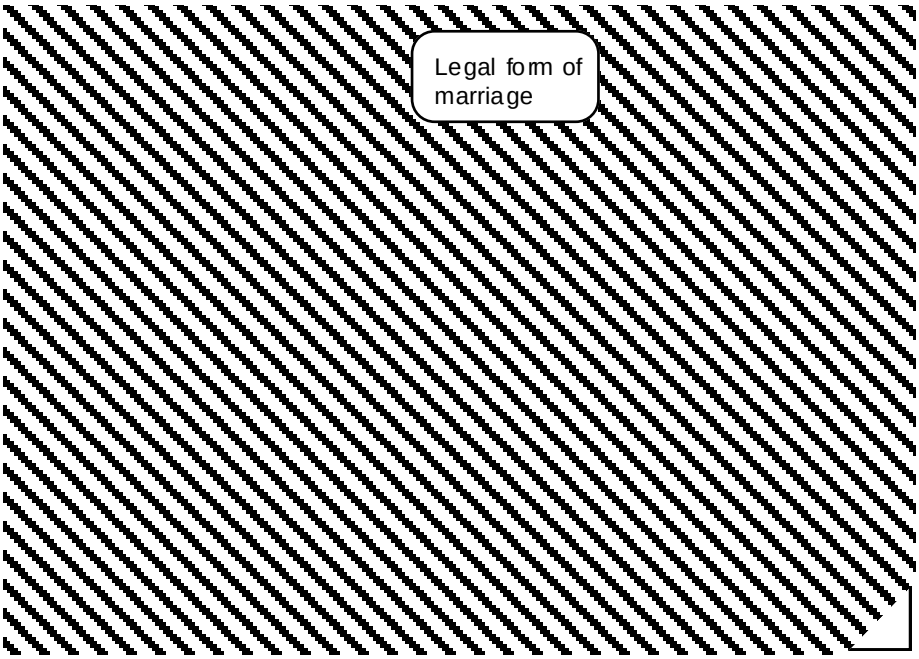


Fig. 3f

Fig. 3f is only an illustration, not a serious design. I look further at generalising classes in the next section.

15.4 Generalising the design

Focusing on the specification of constraints within the Business services layer of code, how do you anticipate changes, design for ease of amendment, ahead of time?

You can anticipate changes in requirements by generalising aspects of the design. There is a trade off however. Generalisation can make a system harder to understand, and harder program, and possibly harder to use.

A rigid hierarchy

Imagine a personnel system that records a company's organisation hierarchy. You might specify an entity entity model of the kind in Fig. 3g.

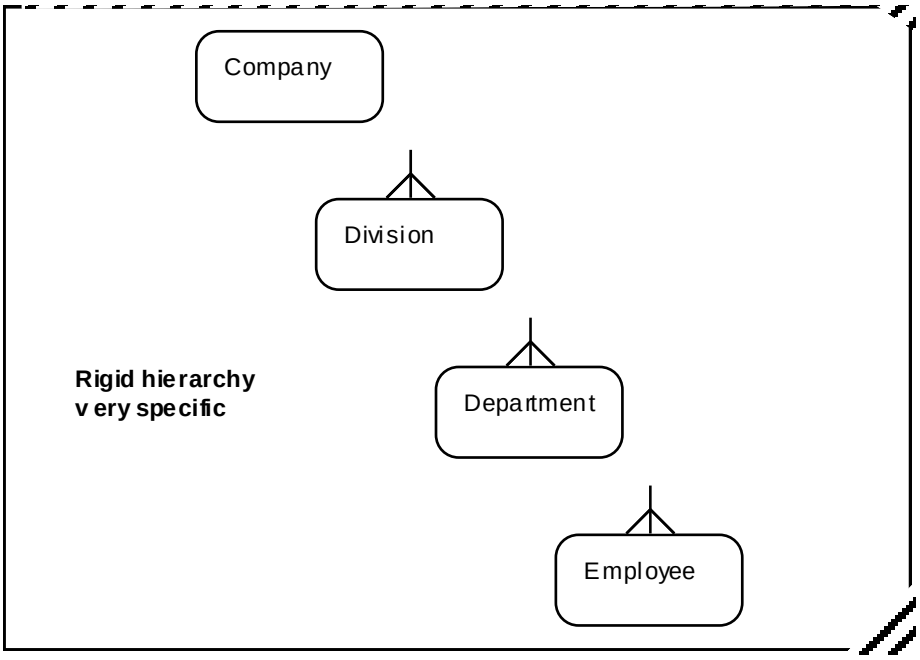


Fig. 3g

A rigid hierarchy is a generative pattern. You should ask: Is it possible for levels of the hierarchy to be omitted?

Suppose you find out the system has to record Companies that don't have Departments, Divisions that don't have Departments, and Company Employees who are not allocated to any Division or Department. Fig. 3h shows a more generic model.

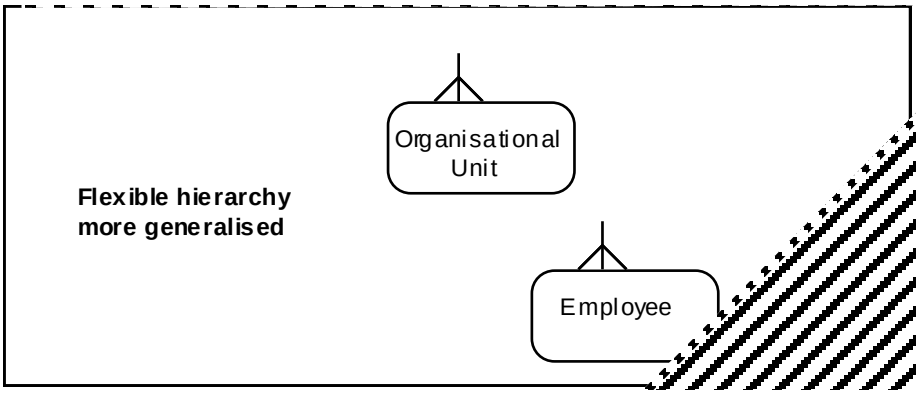


Fig. 3h

The entity model is smaller and more flexible. On the other hand, the system is a harder for designers to work with, and it is more difficult to give users the same kind of usability.

15.4.1 The multiple-V shape

Imagine a vehicle licensing system that records the various reasons why a Person is related to a Vehicle. You might specify an entity entity model of the kind in Fig. 3i.

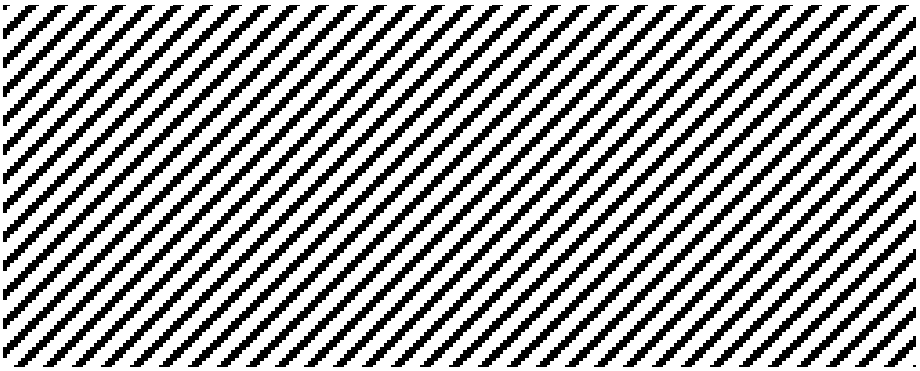


Fig. 3i

But how many other reasons are there to relate a Person to Vehicle? What about Thief? Damager?

Ask of a multiple-V shape: Is it better to anticipate extra relationships by generalisation?

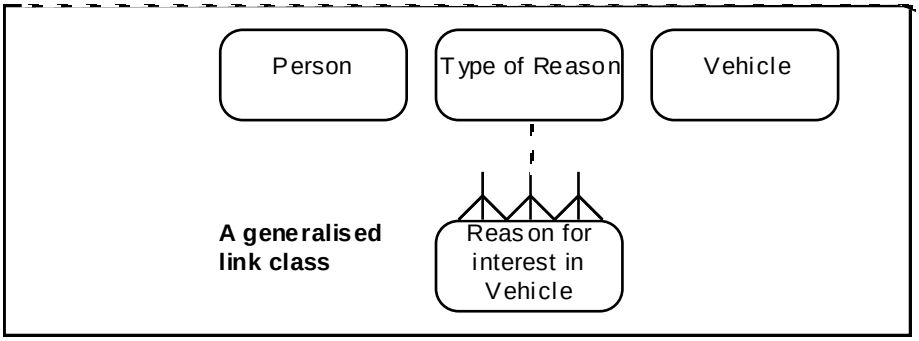


Fig. 3j

Again, the entity model is smaller and more flexible. On the other hand, the system is a harder for designers to work with, and it is more difficult to give users the same kind of usability.

15.4.2 Broader generalisation of classes

Imagine a simple accounts system that records sales. You might specify an entity entity model of the kind in Fig. 3k.

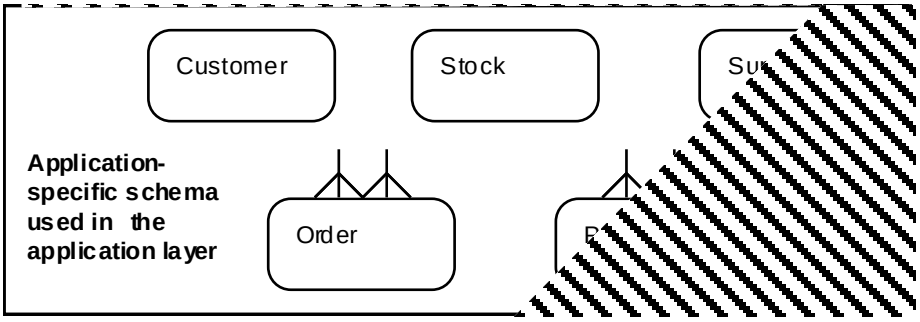


Fig. 3k

The model is clear and specific about what entities are to be recorded in the system. If you translate this model into a data storage structure, then designers can easily write enquiry programs that report on all the sales for one customer, or all the sales of one stock type. Both designers and end-users can readily see what the classes are for, and use them correctly.

But suppose your prime design objective is flexibility. Your brief is to make sure the system can be extended to record new entity types (a Return of Goods, a Salesman), and heaven knows what else.

To accommodate future requirements, you might specify only a few generic classes: 'Contact' instead of Customer and Supplier and 'Stock Transaction' instead of Sale and Purchase, as in Fig. 3l.

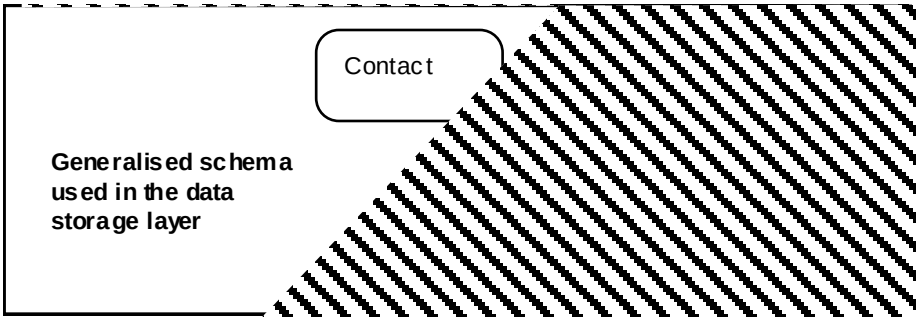


Fig. 3l

This model is more flexible. To record a Salesman, all you need to do is extend the range of 'types' allowed for a Contact. You don't have to change the structure. On the other hand:

• there is more danger of giving the end-users a system that fills up with garbage. It is easy to imagine people mistakenly entering Customers as Suppliers, or Sales as Purchases. Designers will have to work that much harder to constrain how the system used and give users the same degree of usability.

• the system is no longer so easy for designers work with. The programming is more complex. You will have to write extra code to test the contents of a Stock Transaction object, to find out what subclass it really is (Sale or Purchase) before you can process it.

• the system's performance may be degraded, because events and enquiries that require access to all the objects of a logical class (all Sales), will have to trawl through all the objects of the physical class (all Stock Transactions).

15.4.3 Over generalisation of classes

It is easy to get carried away with the idea of generalisation and take it too far. Nobody in their right mind would go the extreme shown in Fig. 3m.

A generalised schema for the data storage layer that will prevent you from having to restructure or reorganise the database

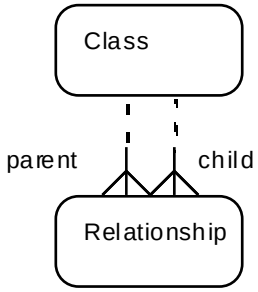


Fig. 3m

Or would they? There is a real motivation to do this, the cost of 'data migration'. Data migration is a serious issue in system maintenance and its one of the issues I come back to in chapter 6.

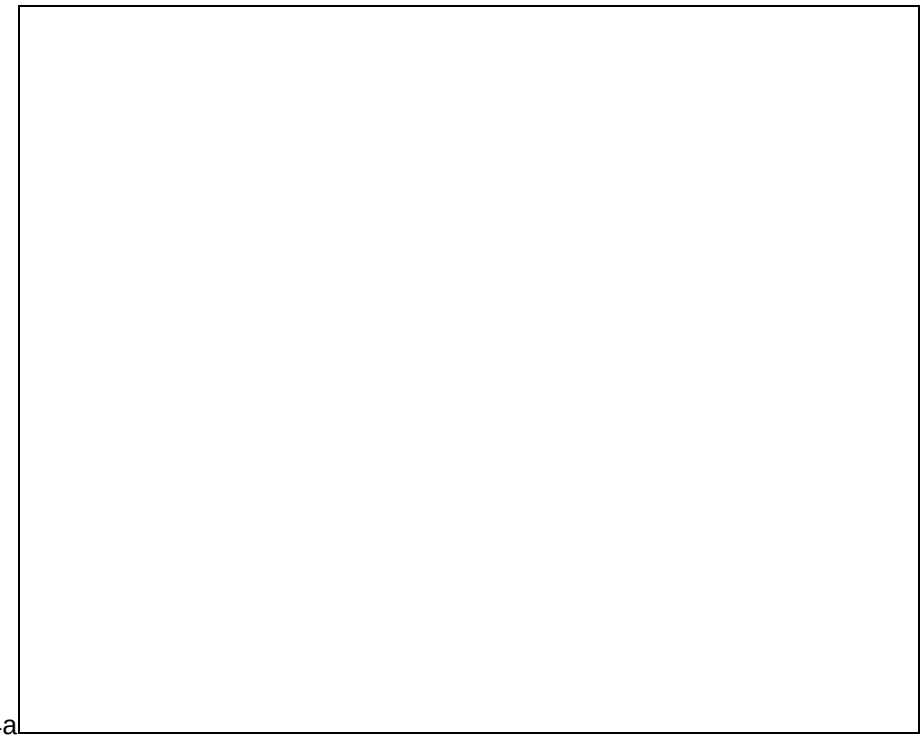
15.4.4 Remarks

Briefly, other chapters suggest you can have your cake and eat it too. You can separate the entity model from the data storage structure. You can code the entity model in the Business services layer on a business rules server - where it can be changed without the need for data migration. You can code the more generalised entity model as the data storage structure on a data server - where it will be sufficiently flexible to reduce the need for data migration. You may find it is not easy to do this using current technology. However, SQL is a natural tool for implementing the Data abstraction layer that is necessary to achieve this separation, and ODBC technology is also helpful.

16. PART THREE: RECURSIVE ENTITY MODEL SHAPES

17. Kinds of recursive structure

Fig. 4a places recursive structures into a three-by-three matrix. Remember, I always declare one end of the relationship to be the master (or top) and the other end to be the detail (or bottom).

Fig. 4a 

This matrix classifies recursive problems into different kinds whose entity models take different shapes.

17.1 From constrained to relaxed

The less constraints you have to build into your model, the simpler the design. The more constraints you have to model, the more complex the design.

So, it is generally easier to design a system with variable-depth-network recursion (top-right corner of the matrix) than with fixed-depth-linear recursion (bottom-left corner of the matrix).

In practical enterprise application development, you tend to find mainly the more relaxed kinds of recursion. So we'll start with simple examples of variable-depth recursion in the top row of the matrix.

In academic computer science courses, you often find the more constrained kinds of recursion. We'll come back later to consider some case studies that illustrate patterns for more constrained kinds of recursion.

(By the way, many of the published design patterns for object-oriented software construction

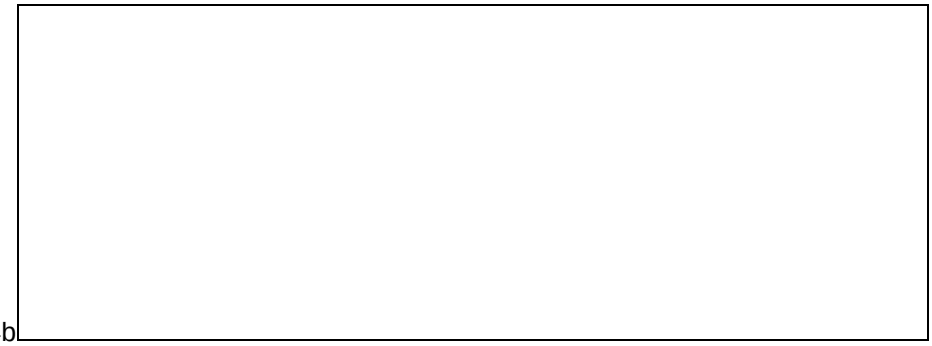
feature recursive communication between objects of a class. Two of the more easily understandable examples are shown in the book ‘object-oriented and business data’.)

17.2 Three patterns for variable-depth recursion

Variable-depth recursion is more common in enterprise applications than fixed-depth recursion. This section shows the typical entity model shapes.

17.3 Linear variable-depth recursion

Fig. 4b shows a List, an ordered sequence of Entries. Entries may be added at the top, or the bottom, or the middle.

Fig. 4b 

An entry in the middle has a one-to-one relationship to the entries on either side of it, the prior and next entries.These two relationships can be collapsed into a single relationship.

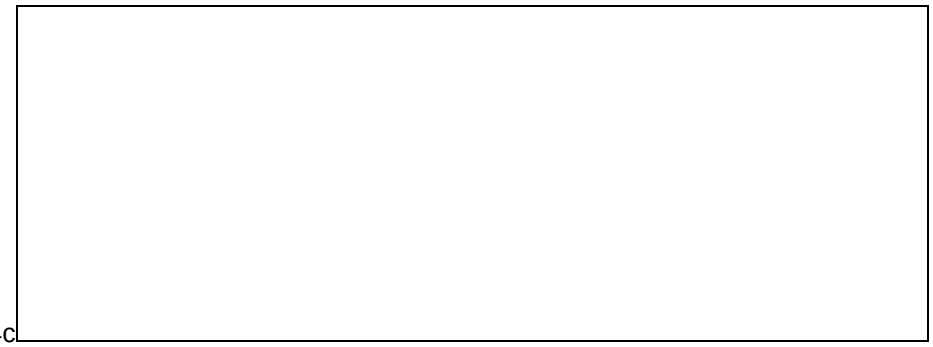
The relationship is optional at both ends because there must be a first and a last entry. And because if this is an enterprise application that only records lists rather than controls them, one might record an entry in the middle of the list before either of its neighbours.

One might store in each entry the key of both next and prior entries, but the convention in relational data analysis is to this is optimise this - to store either one or the other as a foreign key but not both. You can distinguish the primary key and the foreign key with role names:

Entry Number (this)	mandatory primary key
Entry Number (next)	optional foreign key

17.4 Hierarchic variable-depth recursion

Fig. 4c shows a management hierarchy. Each manager may manage several other managers, down to the bottom level where a manager manages employees rather than other managers (but we don’t record employees here).

Fig. 4c 

Again, the recursive relationship is optional at both ends, and following relational data analysis, the detail class will store the identity of the master class as a foreign key. Again, you can distinguish the primary key and the foreign key with role names:

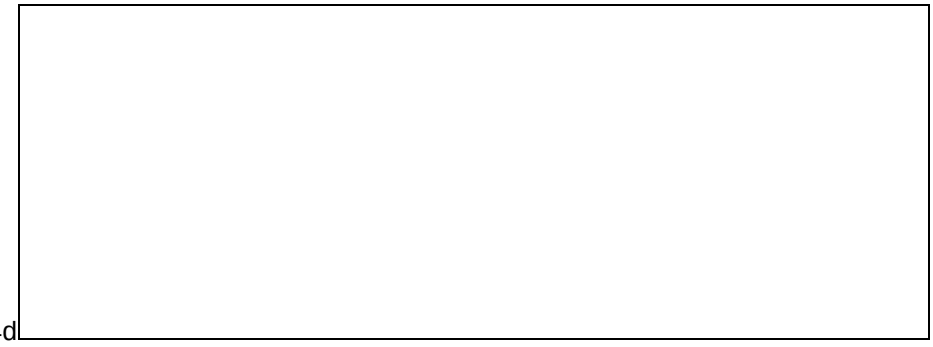
Manager Name (self)	mandatory primary key
Manager Name (manager)	optional foreign key

17.5 Network variable-depth recursion

Fig. 4d shows a 'parts explosion' problem. All but the lowest-level Components in a warehouse may be composed of several lower-level Components. All but the very largest Components may be used in composing several different larger Components. It is important to be able to find out which Components something is made of, and which Components something can be used to make.

Some Components are not composed of other Components and are not used in composing other Components. However, you cannot predict what Components a given Component may be composed from, or used to make, in the future.

Developing the previous pattern, you might draw the model shown in Fig. 4d:

Fig. 4d 

But the many-to-many relationship in Fig. 4d is itself a generative pattern. Fig. 4e shows that resolving this in the normal way leads to a classic V structure, except that the two masters of the V are collapsed into one.

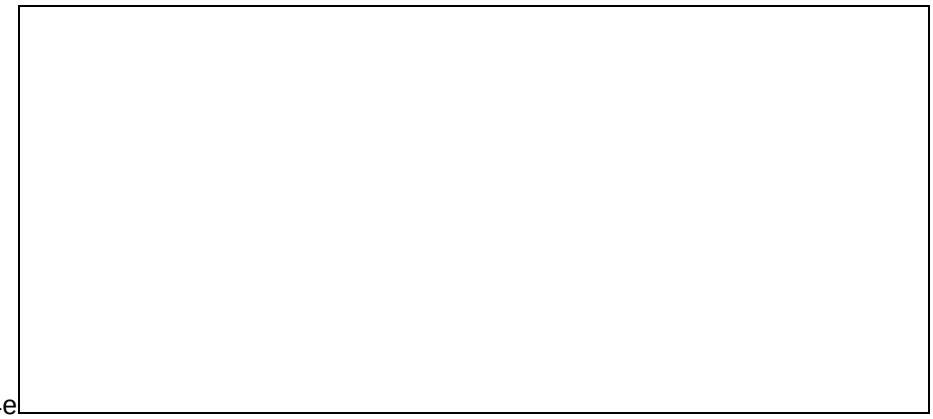


Fig. 4e

An object of the detail class forms a cross-reference between two different objects of the master class. In carrying out relational data analysis, the detail class holds the primary keys of the two masters. You can distinguish these keys by role names.

Component Number (made Component) mandatory foreign key

Component Number (used Component) mandatory foreign key

18. Things to look for in recursive structures

18.1 Constraints on the two masters of the link class

A constraint that applies most many-to-many link objects is that they cannot link one master object to itself. The normal way to specify this constraint is as a statement applied to the foreign key attributes of the link class. For example:

Manager Name (manager) not = Manager Name (self)

You should be aware that this constraint is not sufficient to guarantee an entity cannot relate to itself more indirectly. How to stop the situation where Manager A manages Manager B, who manages Manager C, who manages Manager A? You need to impose a more elaborate rule that might be expressed as 'Do not create an object as belonging to master object that that it already owns as a detail object.'

Most business database designers ignore the problem. They would say that if the user is silly enough to create objects in such a structure, they do so at their own risk. In cases where it is needed, the rule is best specified in specifying the logic of the event that creates a new object (the event must follow the recursive relationships and fail on finding an invalid self-reference).

18.2 Key-only and concrete links

In most of the examples people use to illustrate network recursion, the link class is specifically designed to relate two objects of the master class. It is primarily a cross-reference. Often it is a key-only class.

In other examples, the link class is a concrete entity in its own right, and the recursive connection between objects of the master class is almost incidental. Fig. 4f shows an example on the right.

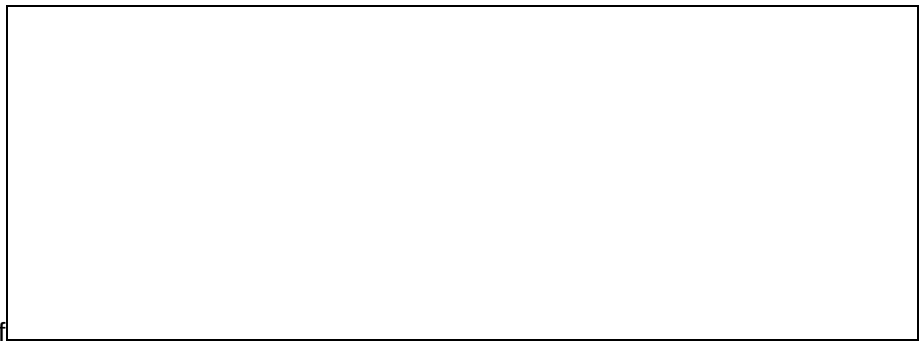


Fig. 4f

A Person who plays the role of committee chair on one or more committees is related to one or more other Persons playing the role of reserve committee chair (and the other way around of course). But this recursive relationship is almost an incidental feature of the situation in which a Committee has two chairpeople - two relationships to Person.

It seems highly doubtful that anybody will every enquire of a Person - which chairpeople are

they related to via Committees? However, if they did enquire, you could design a simple enquiry process to retrieve the information for them.

18.3 Symmetrical and asymmetrical links

Fig. 4g divides recursive structures in a more curious way, into symmetrical and asymmetrical cases.

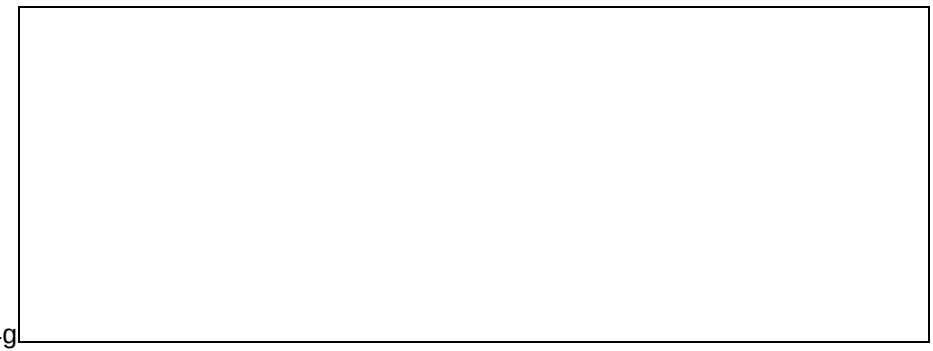
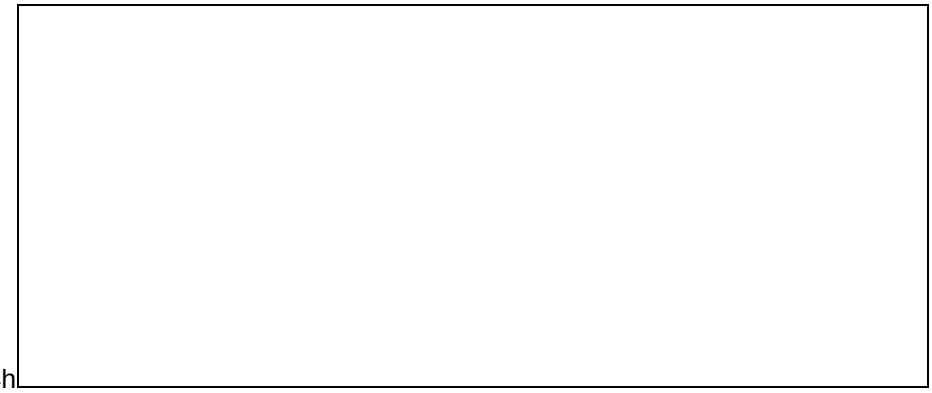
Fig. 4g 

Fig. 4h shows some asymmetrical examples, where the two masters of the detail are different, and can be named differently.

Fig. 4h 

Given a Component, you use one relationship to list all the Components is it made of, and the other to list all the Components it goes to make.

Given a Person, you use one relationship list all their husbands, and the other to list all the wives (depending on their sex).

Fig. 4i shows some symmetrical examples, where every detail has two masters, but there is no way to differentiate one from the other.

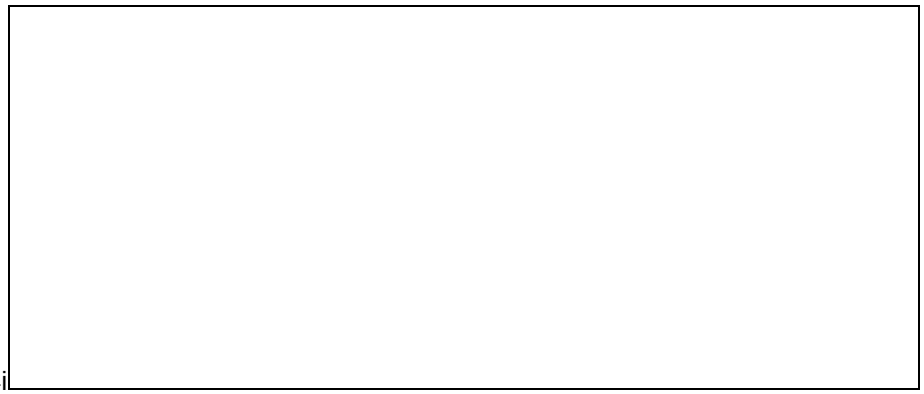


Fig. 4i

The examples help to reveal a practical difficulty in drawing an enquiry access path for symmetrical cases.

Given a Town, you cannot use one relationship to list all the Towns it is connected to by a Route, because it is arbitrary whether that Town is recorded as End 1 or End 2 in the Route.

Given a Person, you cannot use one relationship to list all their Friends, because it is arbitrary whether that Person is recorded as Friend 1 or Friend 2 in the Friendship.

To list all of a person's friends, or all of a town's neighbouring towns, you are obliged to make any enquiry down both relationships (or else record each link entity twice, reversing the sequence of the foreign keys in the second).

What is going on here?

18.4 Turning sets into lists

We set out to model things in the real world and classify them into sets. There is no order to the things in a real-world set. There is no precedence between the friends in a friendship, or the towns connected by a route.

But during software design we turn sets into lists. A list is ordered, one thing comes after another. When we represent an unordered set by an ordered list, we are obliged to impose a sequence on it. We are obliged to list one friend before the other, one town before the other.

So there is an inevitable mismatch between the real world and the implementation. This is the reason why you cannot devise a simple enquiry process to list all of a person's friends, or all of a town's neighbouring towns, without duplicating all the link data twice.

18.5 Modelling events that maintain recursive structures

The entity models for recursive problems can look deceptively simple. Specifying processes to upadte recursive data strucures can be relatively difficult.

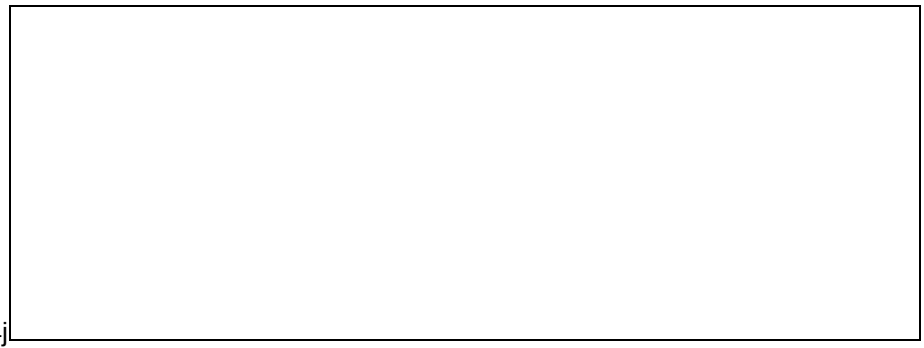
Consider the construction and deletion events of an object in a recursive structure. One event can appear in the state machine of the master class as having both 'gain' and 'loss' effects on different objects of the class. You have to distinguish the event effects with role names, much as you distinguish the attributes in relational data analysis.

Consider the construction event of an object in a network variable-depth recursive structure. When does it happen? You have to decide whether Composition Relationships are only created and destroyed on the birth and death events of a Component, or whether the users need additional events to create and destroy a Composition Relationship at any time between existing Components.

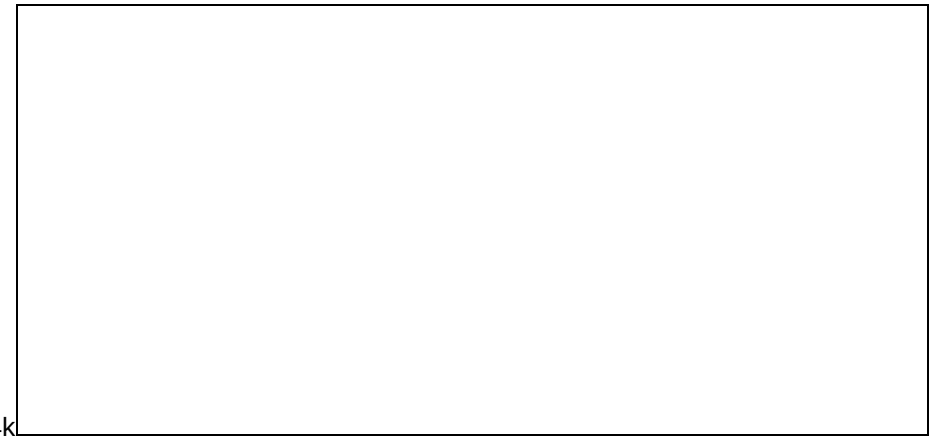
I find that methodical event modelling (supported by methodical object behaviour analysis) does lead economical solutions of the kind that are intuitively correct. The most difficult part is naming the event effects, since several events will affect different object instances of the same class, and you need to be careful to give these event effects a role name.

18.6 Fixing one end of the recursive structure

Fig. 4j shows a royal succession that starts with the declaration of a new king or queen. From that point on, each monarch inherits the throne from the previous monarch. Each succession is a one-to-one relationship between two monarchs.

Fig. 4j 

But given that the first monarch has a fixed and unique place in the lineage, perhaps you should be more specific. Fig. 4k shows you might introduces subclasses and split the original one-to-one relationship into two halves - the next and the prior aspects of the relationship.

Fig. 4k 

We'll explore the use of subclasses in the next section.

19. Two patterns for fixed-depth recursion

Fixed-depth recursion is perhaps more common in academic and software engineering problems, but examples do occur enterprise applications.

19.1 Hierarchic fixed-depth recursion

Consider for example a map spotted with sites of interest. Each site may be divided into areas; each area may be further divided into sub areas; each sub area may be further divided; and so on down to the level of ‘elementary areas’ that are not sub-dividable.

A site is an area that cannot ever be part of a larger area. A site may contain many buildings. Each building on a site must be wholly contained in one ‘elementary area’ (though this may be the site itself).

The first attempt in Fig. 4l is somewhat ambiguous. The optionality of the recursive relationship allows all areas not to be part of a wider area.

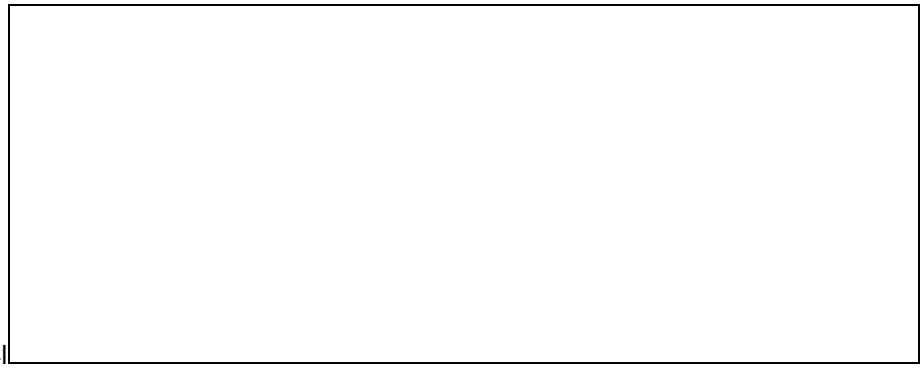


Fig. 4l

The second attempt in Fig. 4m is more explicit, but there is still some ambiguity surrounding the optionality of the recursive relationship.

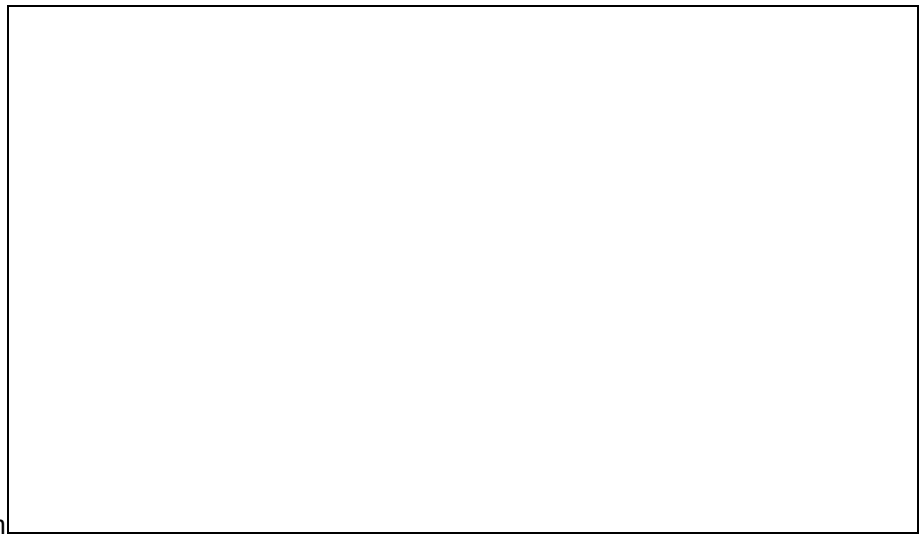


Fig.4m

Fig. 4n removes the ambiguity, making all constraints explicit in the data structure.

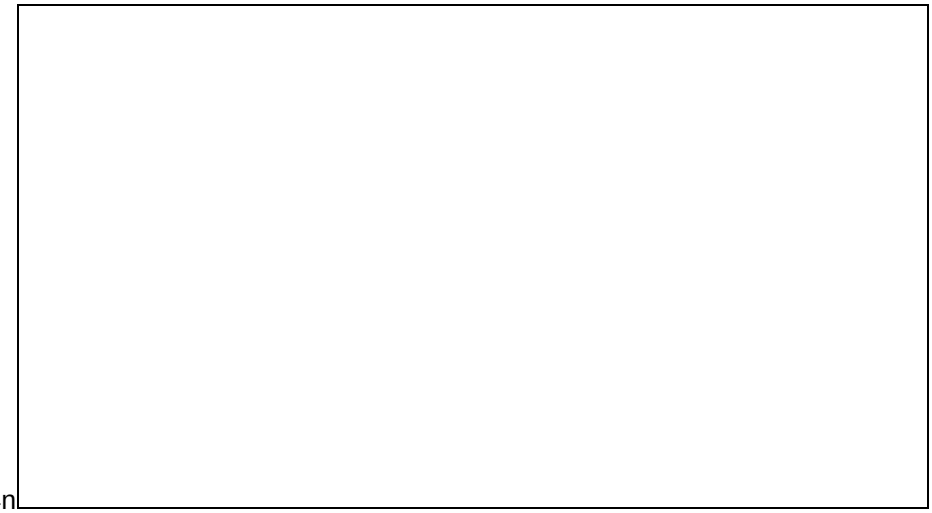


Fig. 4n

Fig. 4n is interesting as a specification of constraints, but in developing event models to specify the processing of examples like this, you will come to a different view of what the distinct classes are.

Event modelling view of the same data structure

Fig. 4q shows the general pattern for modelling a fixed-depth hierarchic recursive class. It involves only three state machines - a basic class and two parallel aspect classes that specify what is different about the special instances of the first or top object and a last or bottom object.

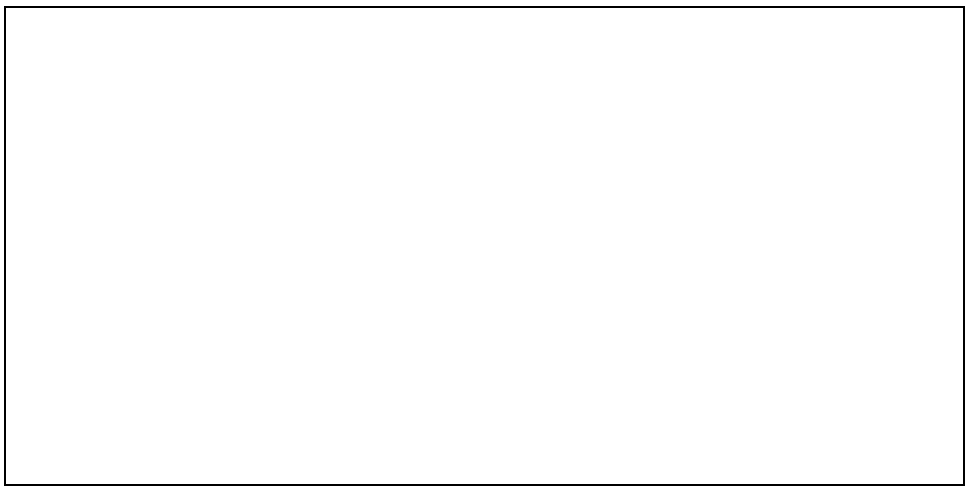


Fig. 4o

Notice how subclasses roll up into their superclass. The book ‘object-oriented and business data’ describes how and why in more detail

Don’t forget we’re talking about the specification of the Business services layer. In Data services layer, you might roll all the classes up into one database table. The attributes of the subclasses may be stored in mutually exclusive columns of the table for the basic class.

19.2 Network fixed-depth recursion

Consider for example a variation of the ‘parts explosion’ problem, with the extra constraint that you know when recording a Component whether it is composed of other Components or not, and whether it is used in making other Components or not.

Fig. 4p shows the ‘typing’ of a Component can never be changed; it remains ‘elementary’ or ‘composed’, and ‘composing’ or ‘non-composing’, for all its life.

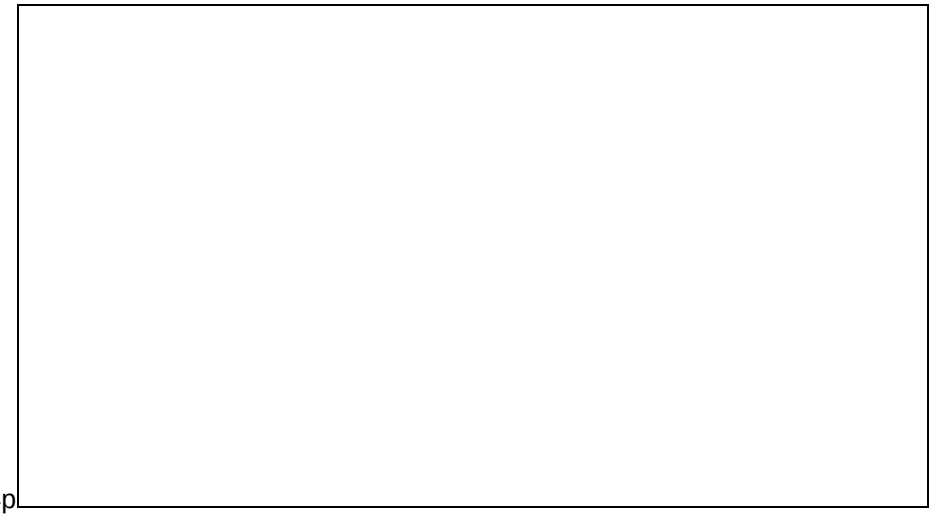


Fig. 4p

Again, Fig. 4p is interesting as a specification of constraints, but in developing event models to specify the processing of examples like this, you will come to a different view of what the distinct classes are.

Event modelling view of the same data structure

Fig. 4q shows the general pattern for modelling a fixed-depth network recursive class as state machines. It involves four state machines - a basic class with two parallel aspect classes, and a link class.

Fig. 4q

The two parallel aspect classes specify what is different about the special instances of a first or top object and a last or bottom object.

19.3 Transient types as state variables

Often, the top and bottom objects in a recursive structure may change. In other words, the 'topness' and 'bottomness' types of an object are not fixed.

Where a type can be changed over time, the distinction between the 'type' and the 'state' of an object is blurred. It is rarely a good idea to model changeable types as subclasses in a data structure. The subclasses are better specified as options in event models and state machines than as subclasses in an entity model.

The standard entity models for fixed-depth recursion show the top and bottom types as subclasses that roll up into their superclass. The state machine of the superclass specifies only that behaviour that differs between subclasses, as mutually exclusive options in the state machine. The state variable of this state machine is in effect a type variable; it tells you what subclass applies.

What happens if you try to extend the standard fixed-depth model to handle a variable-depth structure? Suppose the top node is fixed, but the bottom node may be replaced. The

bottomness of a node is now a temporary state rather than a fixed type. You could introduce a class called ‘period of time as a type’, between the master class and its subclasses.

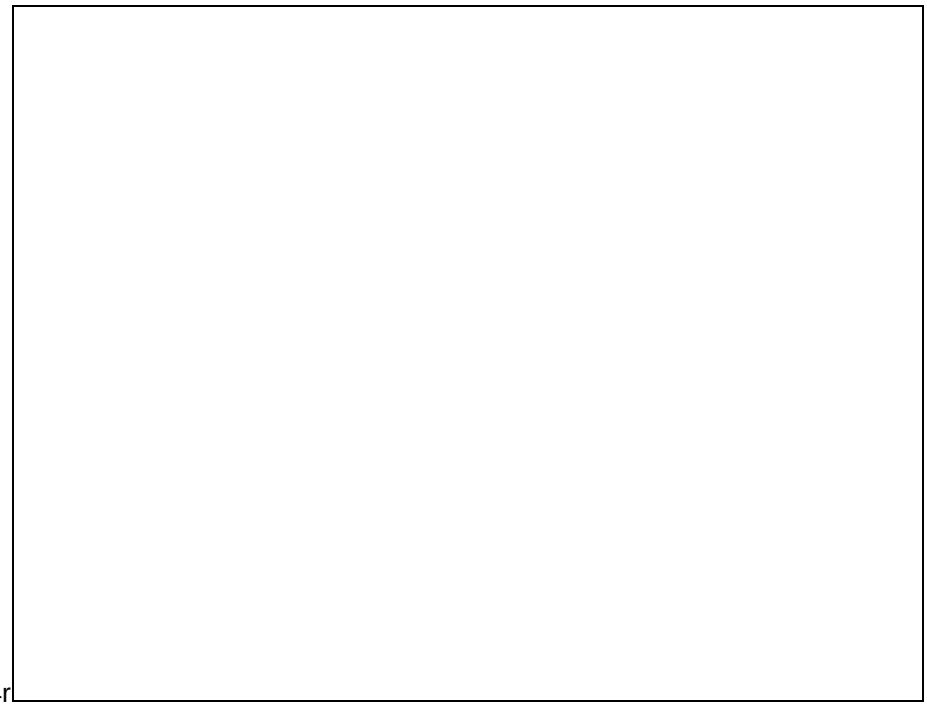


Fig. 4r

You might well specify a period of time as a distinct class in this way if you need to keep a history of each period that each node was at the bottom of the structure.

Normally, past periods are forgotten, and the period of time is represented only by an iterated cycle inside the state machine of a higher-level persistent class. So all the lower-level classes disappear into the state machine of their master class.

If the top node were also variable, and you follow this line of reasoning through, you end up returning to the much simpler entity model I presented for a variable-depth hierarchic structure earlier.

20. Examples of constrained linear recursion

From the design point of view, a relaxed recursive structure is not so interesting as a constrained recursive structure. The most interesting of all is a fixed-depth linear recursive structure.

I have worked through four case studies of linear recursion where one or both ends is fixed, from entity model to processing code. This section presents three of the resulting entity models.

Note how the standard pattern for fixed-depth recursion remains visible in the implemented solutions.

20.1 The Eight Queens problem

Dijkstra (1972) codes the Eight Queens problem as a peculiar kind of first-in-first-out stack. The main task is to model then implement the behaviour of a queen as she steps along a row of a chessboard looking for a safe square to sit in. Fig. 4s shows the entity model reverse-engineered from our coded solution.

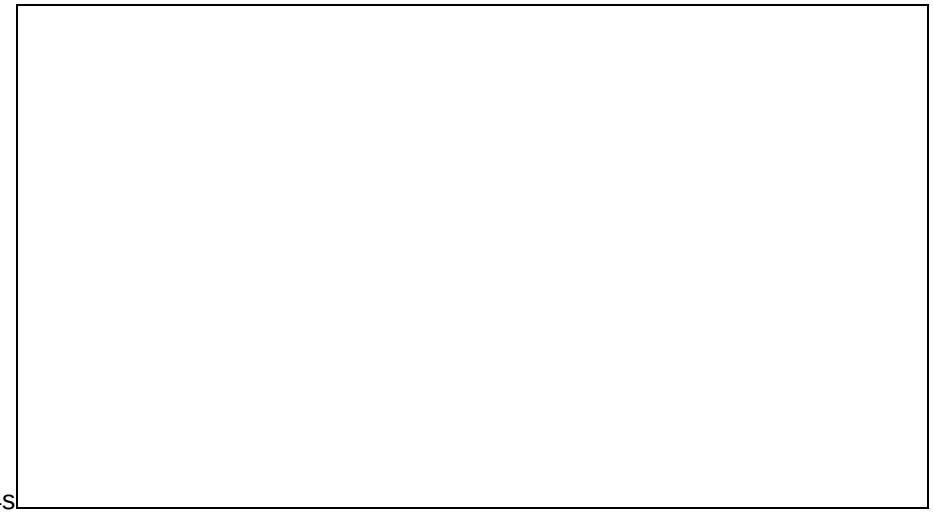


Fig. 4s

As the patterns in this chapter have suggested, it turns out that what you need to model as state machines are the parallel classes QueenBasic, QueenFirstness and QueenLastness. The subclasses appear only as mutually exclusive options within these state machines, not as distinct machines.

20.2 First-in-first-out shelf

The main task in the next example is to model then implement the events that store and remove Items from a first-in-first-out Shelf. Fig. 4t shows the entity model reverse-engineered from our coded solution.

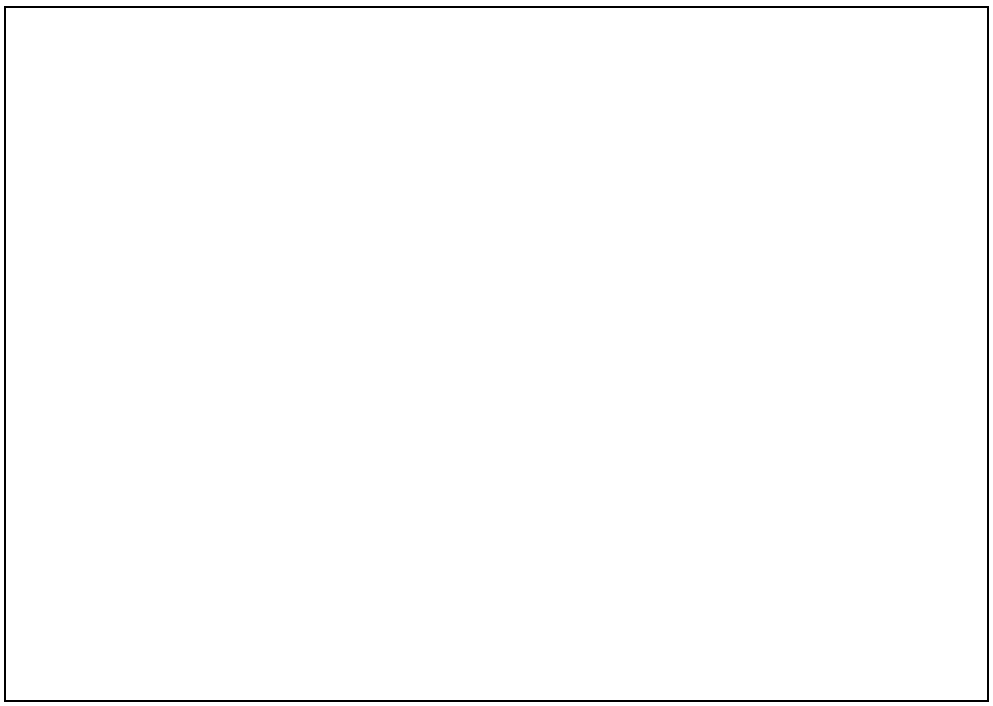


Fig. 4t

I have shown the ‘classes’ that disappear during object behaviour analysis into the state machine of their master class, as boxes without solid boundaries.

It turns out that what you need to model as state machines are the parallel classes (SlotBasic, SlotBottomness and SlotTopness); the subclasses and periods of time beneath them appear only as components within these state machines, not as distinct state machines.

20.3 Chained list

The main task in the final example is to model then implement the insert and delete events that manipulate Items in a chained list. Fig. 4u shows the entity model reverse-engineered from our coded solution.

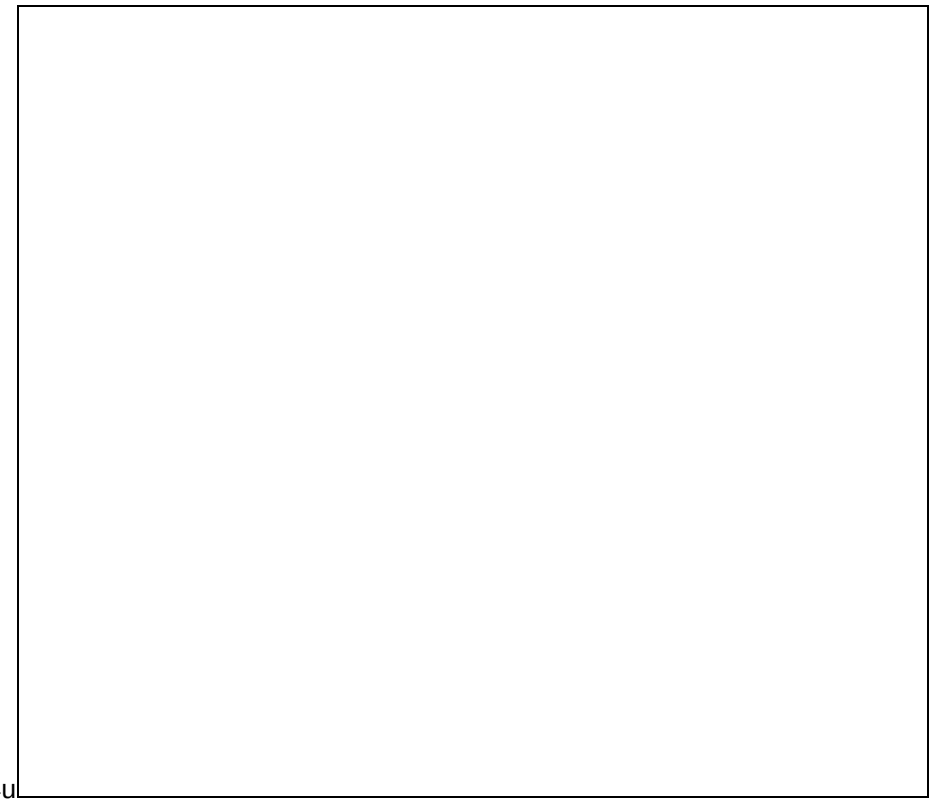


Fig. 4u

Again I have shown the ‘classes’ that disappear during object behaviour analysis into the state machine of their master class, as boxes without solid boundaries.

It turns out that what you need to model as state machines are the parallel classes (ItemBasic, ItemFirstness and ItemLastness); the subclasses and periods of time beneath them appear only as components within these state machines, not as distinct machines.

20.4 A fourth example

The fourth case study I looked at was a very simple last-in-first-out stack. I did follow the standard design pattern, but the design was so trivial and so far optimisable by following various optimisation tricks, that very little of the pattern remained by the time I had finished with it. The design was transformed into less than a dozen lines of code.

(The last-in-first-out stack turned out to be interesting from a different point of view. It is better for revealing optimisation tricks than for illustrating a standard pattern for recursive design.)

21. Conclusions drawn from the case studies in recursion

Some have challenged us with: 'You cannot reconcile the modelling of state machines with the object-oriented modelling of class hierarchies' or 'There are rules you cannot specify as tests on state variables.' Some throw in arguments recalled from their computer science course such as 'You cannot specify a stack using a regular expression'.

The detailed workings of the case studies (not shown here) support my claim that you can meet all these challenges, some by contradicting them, some by accommodating them.

You can reconcile inheritance with state machines

The case studies demonstrate the principle from our paper in the Computer Journal (1994) that you can accommodate inheritance in state machines by specifying the subclasses of a class as mutually exclusive options within one state machine. So a type variable becomes nothing more or less than a state variable.

You can equate classes with state machines

The case studies demonstrate the idea from chapter 5 that you can equate a class with a state machine. There are some cases where it seems a single variable is best maintained by more than one state machine (so 'encapsulation' is broken), but these are very rare and the principle is easily bent to accommodate them when they happen.

The full workings of the case studies exemplify some of the tricks that you need to know about how to model classes in the form of state machines, such dividing a class into parallel aspects. And importantly, the case studies illustrate how to coordinate state machines via event models, and how to turn these event models into working code, object-oriented or not.

You can apply constraints by testing cardinal numbers

The last-in-first out stack case study (not shown here) shows you can derive a simple solution involving precondition tests on the value of a cardinal number from a more complex solution involving precondition tests on the keys and state variables of objects, by well-defined transformation steps. Given that the transformations are understandable and reasonable, you can jump to designing the simple solution without any academic concern about the validity of the approach.

You can solve academic problems using our design techniques

The case studies show that the structured design techniques developed for enterprise applications are equally well suited to computer-world systems. Our notations and techniques are based on those in the UK government's Structured Analysis and Design Method (NCC Blackwell, 1995). Some treat SSADM as only a fuzzy method for information analysis and miss the surprisingly formal method for computer science that it contains.

The case studies show you can completely 'solve' even obscure design problems like Dijkstra's Eight Queens problem using an entity model, object behaviour analysis and Event Modelling.

Other case studies suggest the same analysis techniques apply equally well to embedded systems such as real-time process control systems.

All computer systems have at their heart a set of communicating state machines. We build these as a model of real-world objects and events in the Business services layer.

Prototyping the user interface may help to reveal the objects and events but it cannot completely specify them. Quite distinct effort is needed for analysing, specifying and implementing the Business services layer. This distinct effort involves thinking in ways made explicit by our three-way conceptual modelling techniques.

22. PART FOUR: CLASS HIERARCHIES

23. Bottom-up class hierarchies

Use and abuse of generalization in an entity model

Given some features are shared by several objects, you can abstract these features into a class specification, and identify the objects as members of that class.

Given some features are shared by several *classes,* you can declare them as subclasses of a generalised superclass which owns the shared features. A *class hierarchy* shows entity types divided into subtypes.

It is all too easy to get carried away by generalisation. Questions to be addressed in this paper include:

- How much generalisation is enough? Required? Desirable?
- What is the difference between an enterprise model, a business rules model and a database structure?
- What is the difference between a business class and a generic class?
- When should we introduce generic classes into an entity model used for system specification?

Some of the conclusions may surprise you, because they are contrary to practices that are widely employed by data architects in the IT departments of large enterprises. The big conclusion is this:

> Guideline: The same high-level abstractions or generic classes that are so useful to the enterprise entity modeler and analyst, often turn out, when it comes to the detailed design of a specific system, to be only minor optimizations.

23.1 Case study

The Financial Regulation Agency (FRA) requires a system to monitor how far Independent Financial Advisors influence the formation of contractual agreements between Financial Institutions and their Customers. The main business rules are:

An Account is a contractual agreement between a Financial Institution and a Customer. An Advice Contract is a contractual agreement between an Advisor and a Customer.

Some Accounts result from an Advice Contract. An Advisor may lead to a Customer to create several Accounts, under the relationship defined by one Advice Contract.

The database must record the name and address of each relevant Financial Institution, Advisor and Customer, and the start and end dates of relationships between them.

23.1.1 Finding business classes

It is good idea to start by naming the things the users want to monitor and perhaps control, and to use the terms the business people use. E.g.

- Financial Institution
- Account
- Customer
- Financial Advisor
- Advice Contract

You might slowly be able to move the business towards using a new, more generic language. But it is not a good idea to start there. And you will still need to analyse and understand the specific concepts.

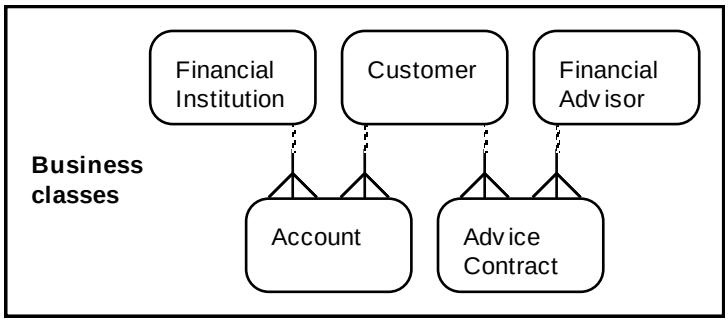
23.1.2 Finding business relationships

You go on to look for associations between pairs of classes and draw them as relationships. You can resolve any many-to-many relationship by naming the link class that is the reason for, or result of, the relationship. E.g.

- An Account is a cross-reference between one Financial Institution and one Customer.
- An Advice Contract is a cross-reference between one Independent Financial Advisor and one Customer.

"Interesting. I have often done this in explaining the idea of relationships to those unfamiliar with the idea." Michael Zimmer

So, a first attempt at building an entity model might be:



Notice that the relationships specify constraints on associations between objects. One obvious constraint is that the system cannot relate objects in two classes that are not connected by a relationship in the model.

23.2 What makes a good business rules model?

A good business rules model has two features that are worth noting here.

23.2.1 Objects should be recognisable by users

A business rules model is built at the level users can discuss. Users should recognise the difference between the classes you define. In addition, users must be able to differentiate between objects within one class; and for this, they need a uniqueness constraint to be placed on some or all of the data attributes of the objects.

I loosely distinguish between 'firm' and 'fuzzy' classes.

'Firm classes' are composed of objects so fundamental to this or another enterprise that the objects must be labelled with a unique identifier for the business to succeed.

E.g. a laundry business attaches a numbered label to each item in a customer's pile of laundry; so it can reassemble the pile after washing. The business simply cannot succeed without these numbered labels.

E.g. a clearing system cannot work without unique codes to identify banks, their accounts and transactions on those accounts.

The best object identifiers are those that exist not only in a computer system but also in the real world. So, when a computer system is introduced, the object identifiers will be available to users on datan entry.

'Fuzzy classes' are composed of objects that cannot easily be identified. This means it is easy to create duplicate objects by mistake. Of course, a computer system can generate a unique number for each object in a class, but such system-generated keys are no help unless or until they become used in the business world.

E.g., consider the five classes in our model so far. There are four firm classes and one fuzzy class. The FRA provides unique reference numbers for Financial Institutions and Advisors, who in turn provide unique reference numbers for Accounts and Advice Contracts. But how do users identify Customers?

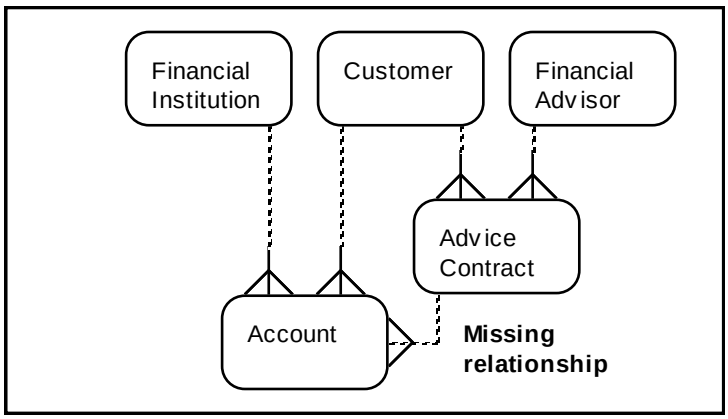
Fuzzy classes are a big headache. It is difficult to identify objects and prevent duplicates. But let us ignore the pain for now and move on.

> “Even with what you call firm classes, it really is hard to identify objects and prevent duplicates. We have a whole operational unit to deal with clients of the government healthcare system, and there are still lots of data quality problems centred around identification of individuals.” Michael Zimmer

23.2.2 Enquiry requirements should be supported in an obvious way

Back to the requirements. How do designers list for a given Advisor, all the Accounts that have resulted

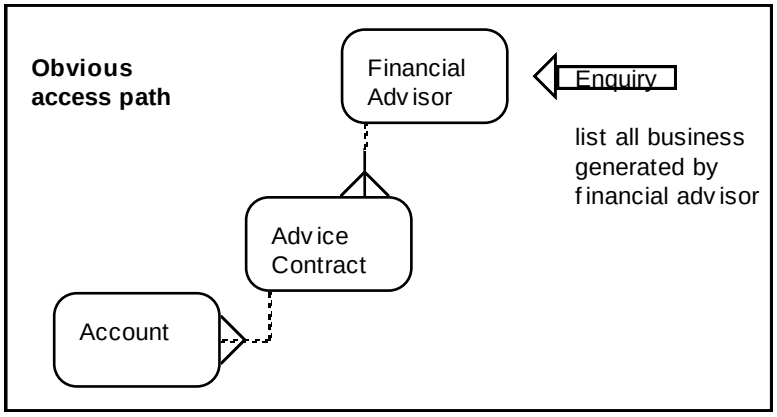
from their Advice Contracts? A logical relationship is missing from the model. Adding the missing relationship, I arrive at this model:



Note that putting business rules into the database structure

- constrains system update processing as it should be, and
- simplifies enquiry processing, but
- may make it harder to accommodate subsequent changes.

The access path for the enquiry is shown below.



"I think relationally, so I don't think of navigation so much as joins. For a join, there is no particular starting point for the path." Michael Zimmer

Ah! Not thinking about the navigation routes or access paths of processes is one of the main reasons why people fail to specify the 'right' relationships in entity models.

23.3 Generic classes and relationships

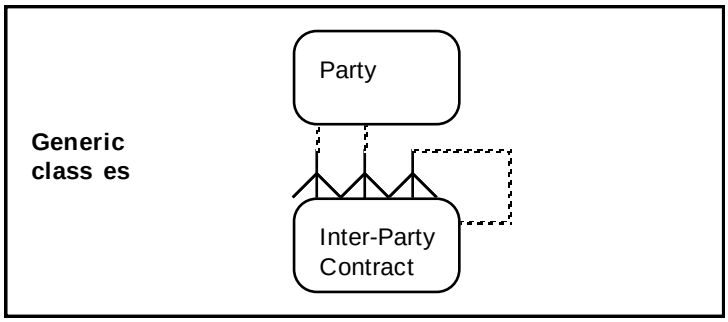
The main topic of this paper is the introduction of generic classes and relationships into a model that is to be used for processing.

What is the difference between a business class and a generic class?

- A business class should correspond to a set of objects that business people recognise as sharing similar features.
- A generic class is a generalisation of two or more business classes, at a level higher than that talked about by business people.

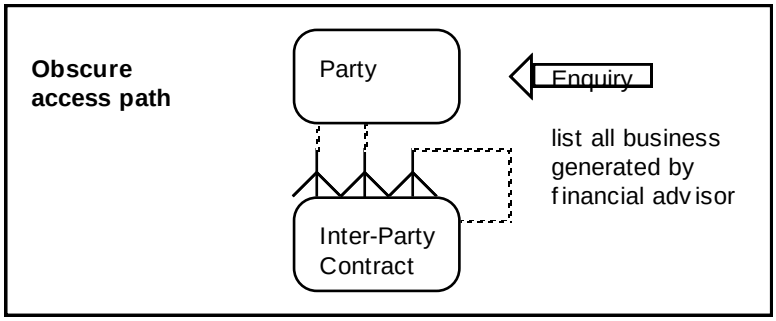
Consider generalisation in the case study. The Financial Institution, Independent Financial Advisor and Customer classes share features, a name and address attributes. Similarly, the Account and Advice Contract classes share features, start and end date attributes.

Therefore, you might abstract two superclasses. You have to invent terms to name them, since you have moved above the level that users talk about. The names I choose are: Party and Inter-Party Contract. Now it becomes possible to build a much more abstract model of the business requirements.



The paper later in this series called <Enterprise entity models> suggests this may be OK as an enterprise model, or as a highly generalised database structure but:

- What are the keys of these classes? The keys must be artificial because these superclasses are abstractions, fuzzy classes, and not things the business people talk about or recognise.
- And how do we support enquiries? E.g., List for a given Advisor, all the Accounts that have resulted from their Advice Contracts? The required access path through the abstract model is obscure.



This is a poor specification of the enquiry. The problem is that the business rules are missing. If the rules are not in the diagram, then they must be in documentation behind the scenes.

A business rules model is not just a diagram of classes and relationships; it includes the detailed specification of each class and relationship, and all the business rules contained therein.

So where *do* you specify the various processing constraints on how Inter-Party Contracts are created between Parties? Just a few of the constraints to be specified are:

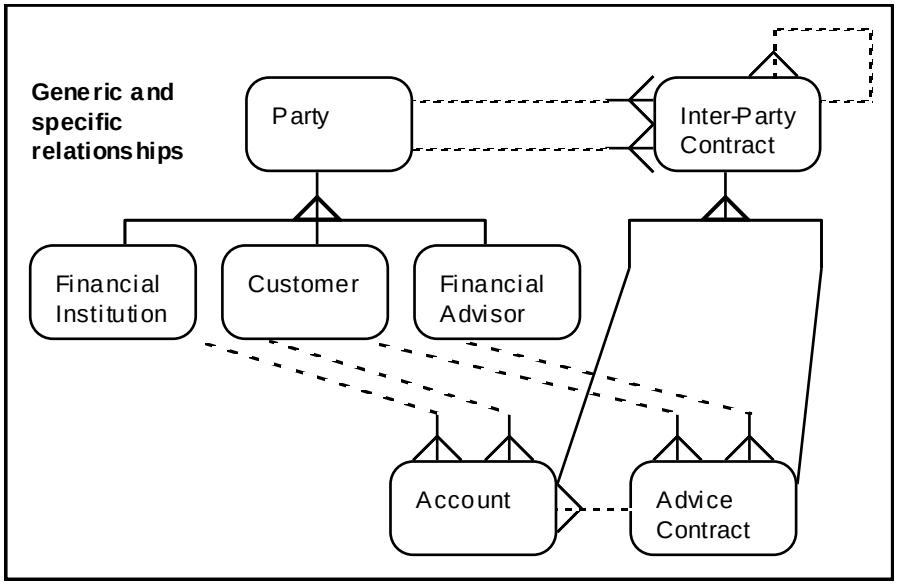
- an Inter-Party Contract cannot associate classes not declared as subclasses of Party,
- an Inter-Party Contract cannot associate Financial Institutions and Advisors,
- the only way an Inter-Party Contract can be related to another Inter-Party Contract (by means of the recursive relationship) is where the parent is an Advice Contract and the child is an Account.

The companion volume, The Event Modeler, introduce several ways to specify constraints. The focus here is on using an entity model to do this.

23.4 Showing generic and specific in one model

The main problem caused by drawing a class hierarchy of super and subclasses is not to do with the classes, but to do with the relationships between them. Do you need both super and subclass relationships? Three possible models might be drawn:

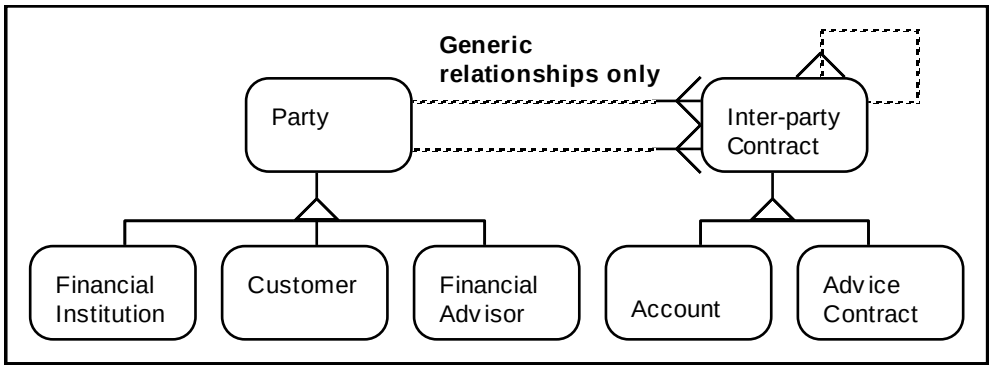
23.4.1 Model with generic & specific relationships



This is unacceptable, because there is redundancy. The generic relationships say nothing that is not

already said by the specific relationships.

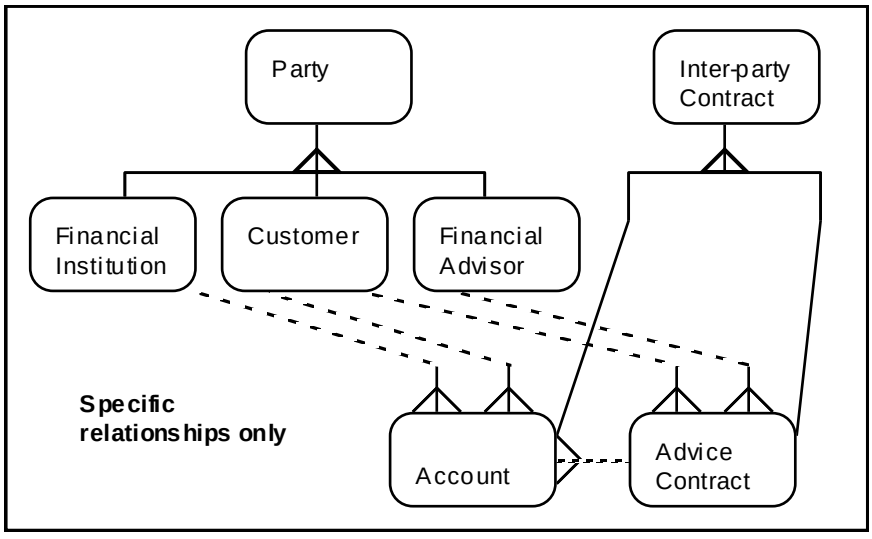
23.4.2 Model without specific relationships



Diagrams like this are often drawn. The papers on <Enterprise models> suggest they may pass as enterprise models, but they make poor system specifications, because too many missing business rules are missing.

23.4.3 Model without generic relationships

The third model below shows specific relationships but not generic relationships.



This seems the best model of the three in terms of specifying business requirements. This brings us to the second conclusion that may surprise you. If you draw specific subclasses, you ought to draw specific relationships. This is not absolute rule, just a guideline.

Guideline: Where we take the trouble to specify subclasses in a business rules model, then we ought to specify the relationships down to the same level of specialisation.

MICHAEL: I think, in practice, that is what I would do.

DAVID: In your example, you divide Party into three subtypes: Financial Institution, Customer and Financial Advisor.

It is arguable whether a Financial Institution is ***inherently*** a Party, but Charlie the Customer and Sally the Financial Advisor certainly are not Parties. They were not born as a Customer or a Financial Advisor. They were Persons first and that is the nature of them as entities. Only later did they enter into relationships that provided them with those roles.

GRAHAM: Sure, but yours is a model of the real world as you see it. Mine is a model of a specific problem domain. My customer didn't know Charlie or Sally when they were born. And frankly doesn't care a jot about their role as human beings.

23.5 Specification and programming costs of generalisation

During specification, a generic class may cost more, by adding complexity to an entity model, than it saves by removing replicated class features. Not always, but often enough to be a concern.

During programming, a generic class usually makes it harder for designers to write programs.

Generic classes are often promoted as a means to save effort at amendment time. Even here the picture is far from clear cut. Let us see by way of example.

23.5.1 Adding a new attribute

Suppose users want to record post codes separately from addresses. The Party superclass provides the Single Point of Specification for this amendment, but this only works if the enhancement applies to *all* subclasses of Party.

Moreover, inserting a post code attribute into three classes is not a big deal; changes to relationships are more of a problem.

23.5.2 Adding a new class and relationships

Suppose the term 'Independent' means only that Financial Advisors are free to take commission from several Financial Institutions. Users now want us to record the contractual agreements (Commission Contracts) formed between Financial Institutions and Financial Advisors.

A Commission Contract is a cross-reference between a Financial Institution and an Independent Financial Advisor. A Commission Contract may be regarded as a third subclass of Inter-Party Contract.

You can easily extend the model with the new class, and draw relationships to it from the existing classes.

>> diagram <<

Does the presence or absence of the abstract superclass (Inter Party Contract) make much difference to the amendment effort? The generic class saves us specifying two attributes (Start and End Dates) for a Commission Contract and makes us specify an extra generalisation relationship. In practice, these savings and costs are too small to be of much concern.

The critical part of the amendment is the specification of new relationships from Institution and Advisor to Commission Contract, which must done whether the superclass exists or not. The amendment might mean redrawing a diagram or updating text documentation behind the scenes. It might mean altering a specification of data items or processes, classes or operations. Whatever you do, it is likely to take about the same amount of effort.

23.5.3 Database costs

Amendment of a specification, or a program, is one thing. Amendment of a live database structure is another. Amendments that force a database schema change normally cost more than those that involve only rewriting and recompiling programs.

The business entity model is a specification; it may become the structure to which business services are coded, but neither of these means that it must become the physical database structure. Of course the database must be designed to contain the business classes and relationships, but it may do using a different structure.

To reduce the costs of restructuring the database, the database designer may roll subclasses into generic database tables. Then, given the kind of software architecture described in the group papers on <Architecture definition>:

- the user Interface layer presents specific subclasses, and
- the business services layer programs deal with specific subclasses, but
- the data services layer stores generic database tables.

The database designer must weigh the undeniable advantage of reducing database schema evolution costs, against difficulty of maintaining more complex data abstraction processes, and the performance costs below.

23.5.4 Performance costs

Implementing a generic class as a database table may damage the overall performance of a system in several ways. It can both increase database storage requirements and slow down transaction processing.

- Implementing a generic superclass as a database table is likely to mean the system must generate and store an additional artificial key. This is not a trivial matter in large systems.
- System-generated keys slow down distributed datan entry tasks, because every time an

object is added to an abstract superclass, the distributed process must call one central server to activate the component that generates an artificial key for the new object. This component becomes a bottleneck.

- Transactions will be slower if they have to instantiate an object by joining super and subclass tables, since this means accessing two tables instead of one table.
- Transactions can be slower if they retrieve objects of a specific subclass via a generic relationship, because they must access many irrelevant subclass objects to find the ones they want.
- Old-fashioned database technologies can slow multi-user access even further by locking the whole of a superclass database table when one row is accessed.

If several distributed locations are creating objects of a single class, then how to guarantee they do not create duplicate primary keys?

- Either they share one key value generator - they call into a central server when they need to create an object.
- Or each location has its own primary key range and its own key value generator. This means objects are location-dependent, in effect, they are members of distinct classes.

23.6 Conclusions

23.6.1 The places for generic classes

Generic classes have various roles. They can be used by analysts to prompt business people to think about their business. They are helpful in enterprise-level models, where they are needed to suppress detail, and prevent the enterprise model from growing impracticably large.

> Guideline: where generic and specific classes both appear in an enterprise model, then it is advisable to show the generic relationships and suppress the specific relationships.

Database designers often roll subclasses into a generic class (or rather abstract table), to reduce the number of 'joins' and reduce the cost of database restructuring on amendment, even though highly generic tables do tend to make database programming harder.

Generic classes tend not to be so helpful in system specification-level models where the business rules must be made explicit rather than hidden from view. Given the goals are to specify the business rules at a level of generalisation users can understand, and to specify the processing constraints that designers must apply to datan entry, then it is advisable to show the specific classes.

> Guideline: where both generic and specific classes both appear in a business rules model, then it is advisable to show the specific relationships, and perhaps to suppress the generic relationships.

Note that this discussion of generalisation has said nothing about granularity, that is the size of components, classes and operations. Granularity is discussed in the Chapters <Aggregate entities> and <The ubiquitous business object>.

23.6.2 Rules of thumb

Finally, later chapters identify some relatively mechanical principles that help to restrict the pointless proliferation of class hierarchies in business rules models. A superclass must:

- be non-trivial, more than a common attribute or two
- have generic behavioral features (operations) not just structural features (attributes and relationships).

Subclasses must:

- inherit ALL their superclass's features
- extend their superclass with extra features
- be mutually exclusive, not additive.

These points are explored in more detail in later chapters.

24. Top-down class hierarchies

It is easy to define very high-level abstractions, and tempting to define deep class hierarchies. I believe it is almost always a mistake.

If you abstract a generic class that business people do not naturally talk about (e.g. Party), then this should not be regarded as a business class. It is more accurately regarded as a minor optimisation designed by us to avoid a minor redundancy in the specification (e.g. replication of name and address attributes in Customer and Employee classes).

Yes, a generic class can give greater adaptability to the resulting system, if carried through to design, but at some cost. If you cannot confidently predict the benefit will be realised, then don't pay the cost!

> Guideline: Very few abstract superclasses deserve a prominent place in a business rules model we use to code business services.

David Hay tells me his models always feature entity types that are general enough to apply to all businesses, but are also recognizable to all (see the earlier chapter).

24.1 A conversation with David Hay

GRAHAM: I notice in your book that Party is subtyped into Person and Organization. May I point out that until the UK tax laws changed, one-man companies were common?

DAVID: This is an interesting point. US tax law sees three kinds of businesses: corporations, partnerships, and sole proprietorships. A sole proprietorship is a company that happens to consist of only one person. The proprietorship is an organization and the owner is a person, with the two related to each other.

The model could be drawn as below, where ORGANIZATION = corporation, sole proprietorships, etc. and PARTY RELATIONSHIP TYPE = employee, owner, spouse, member etc.

PARTY <-- PERSON
PARTY <-- ORGANIZATION
PARTY (1) --< PARTY RELATIONSHIP >-- PARTY (2)
PARTY RELATIONSHIP TYPE --< PARTY RELATIONSHIP

Notations used here
TYPE <-- SUBTYPE
ONE --< MANY >-- ONE

GRAHAM: One tangible real-world person can appear in your model as two objects, one of class PERSON and one of class ORGANISATION. I wonder, would object-oriented designers complain about this? And it does suggest to me you are modeling roles rather than entities.

DAVID: In a sense you are right, but it's a different kind of role. I don't have a problem with a person appearing twice because I believe that the person that has attributes "birthdate", "social security number", etc. is a different thing than the sole proprietorship that has attributes "establishment date", "annual revenues", etc. In this case, by looking at the thing from a different perspective, you are looking at different ***things***.

The same might be said for "customer" and "vendor", but in this case, the role is defined not by a different view of the thing itself but rather by the relationships the thing participates in. I can define a "customer" as a Party that is in a "buyer of" relationship in an order. You can't define a sole proprietorship as a person with a relationship.

OK, maybe you can, but I am going to pretend that you can't. So there! <g>

MICHAEL: I have had a lot of experience with generalisation over the last decade, starting before I became familiar with David Hay's approach. My most recent thinking has been that you should start with a literal business perspective, and then explain the benefits of the more generic perspective. You make the generic perspective the new business perspective.

GRAHAM: That implies a long-term commitment to managing users and models, a commitment that is beyond most of the environments I have worked in. My rule is - if you cannot confidently predict the benefit will be realised, don't pay the cost.

How much generalisation is enough? Required? Desirable? Generalisation is easily overdone. E.g. it takes me only a few moments to construct the hierarchy below.

Level 1	Level 2	Level 3	Level 4
Partnership			
	N-way Partnership		
	3-way Partnership		
	2-way Partnership		
		Friendship	
		Contract	
			Account
			Advice Contract
			Commission Contract

This deep class hierarchy looks OK, and that it what is worrying, because it is not OK. Creating a class hierarchy to describe a world I imagine is not a profitable exercise.

The most useful classes and superclasses come not out of a software model builder's mind, but out of business people's understanding of their business. The best superclasses and subclasses have distinct features of interest to people running the business.

DAVID: There is a lot of bad hierarchy specification that I agree should be dispensed with. But I have ample examples of fairly deep structures that were both meaningful and well understood by my clients. For an oil company, I had to divide Real Spatial Element as shown below.

Level 1	Level 2	Level 3	Level 4	Notes
Real Spatial Element				
	Earth Volume			
	Linear Object			
	Location Point			
		Control Point		very important to an oil company (and anyone doing geographic enterprise applications), because the boundary of any piece of land is defined as a set of points. They want to be able to define that space.
		Road Landmark		
	Geographic Area			
		Geopolitical Area		
			Country	
			State	
			County	
			City	
			Postal Area	
		Management Area		defined by organization, say marketing region.
		Surveyed Area		as in the U.S. surveying system.
			Township	
			Section	
			Range	
		Natural Area		
			Lake Boundary	
			Habitat	
			(Oil) Reservoir	
			Projection	

GRAHAM: I understand you can build such a hierarchy. But why? And how stable is it? Challenges to mutual exclusivity might include:

- lake = habitat? (house boats, houses on stilts)
- city = county? (happens in the UK)

- postal area = city?
- management area = county?

DAVID: Why do it? Because it accurately describes the world. Should the structure be implemented with just a few tables? Probably. But in the analysis model it was extremely helpful to be able to explore categories and sub-categories.

GRAHAM: I am sure it describes the geographical features of the problem domains your customers have been interested in so far - at least at a general level.

DAVID: And these are fundamental classifications.

GRAHAM: I guess you mean fundamental in the sense that you don't have to change the top-levels of the class hierarchy much when you move from problem domain to problem domain. But there will be considerable redundancy in the classification for some customers. Few care about CONTROL POINTS and ROAD LANDMARKS.

DAVID: It isn't that there is redundancy. It is true that some of the elements of the model are not of interest to some clients. That's ok. They don't get included in that client's model.

GRAHAM: If nothing stops you extending your class hierarchies for new customers, then I suspect your hypothesis about the classification's validity cannot be disproved, and is therefore not a scientific claim.

DAVID: To the contrary. My models are a set of facts that are specifically constructed so that they can be wrong. That is the advantage of my method of labeling relationships. It is a fact that a geographic area is not the same thing as a point or an earth volume. They sound simplistic, but they are an important basis for what is built on this model.

GRAHAM: Again, one real-world lake can appear in your model as two objects, one of class LAKE and one of class HABITAT.

DAVID: The definition of a habitat is different from the definition of a lake boundary. REAL SPATIAL RELATIONSHIP (of a REAL SPATIAL RELATIONSHIP TYPE "habitat location") links them together.

It is a fact that a city is a kind of geopolitical area and not a management area. Denver County is one political entity, while the City of Denver is another. The fact that they are conguent is an additional fact.

One of the amusing things about this model is that people are inclined to "hard code" the relationships between cities and states, states and countries, etc. In the United States, you can assert specifically that each city must be in one state (Kansas City, Kansas is a different city from Kansas City, Missouri), and that each county must be in one state. But people who haven't travelled much want to assert that each city must be in one county. They don't know that New York City consists of five counties and Atlanta has something like three. So, the workhorse is REAL SPATIAL ELEMENT RELATIONSHIP.

While the boundaries of a GEOPOLITICAL AREA may change, of course, the fact that one constitutes an occurrence of an entity type that is classified as I describe does not. A city will always be a geopolitical area, not a management area or a natural area.

GRAHAM: Why not? The management team headed by the mayor who won the last city election will

surely define the city's geopolitical area as a management area!

DAVID: The definition of "city" is that it is a bounded area whose boundaries are defined by law, and which has the characteristics of a municipality. It is governed by an organization for that purpose. (The government is a different thing, by the way--an ORGANIZATION.) If Lever Brothers decides that New York City is a "marketing area", the boundaries of that area would be the same as the boundaries of the City of New York. But it is a different thing. If the City is determined to be a habitat for a rare breed of pigeon, then the boundaries of the habitat may also be the same, but a habitat is also a different thing.

Again, the question is: what is the thing (object?) you are talking about. You allude to the right problem. I suspect that object-oriented people are too focused on tangible things. Too much Aristotle. Not enough Plato.

To see two different areas occupying the same space is not to describe the space redundantly. It is to say that two different areas are intimately related to each other.

GRAHAM: Hmm... You are interested in drawing entity models to understand and explain what a business is about. I am interested drawing entity models that work well as the database structures of enterprise applications to support known data processing requirements.

24.2 Conclusions

I have challenged David Hay about the extent to which class hierarchies dominate his entity models. David's class hierarchies are surely very useful as analysis tool, but won't somebody have to transform them into my kind of entity model before they build an enterprise application?

This modeling practice took hold of some entity modelers when object-oriented design first became the vogue. But often, their class hierarchies were/are ill advised. And often, the hand over to design is problematic. Some analysts never realise the entity models they draw for discussion with users have to be rebuilt by the database designers. Anything that results in a needless structure clash between the entity model and the database, or indeed between the object-oriented code structure and the database seems bad news to me.

This brings me to a surprising conclusion, surprising because it is contrary to practices that are widely employed by enterprise data architects in large IT enterprises.

> Guideline: Very few deep class hierarchies deserve a prominent place in a business rules model we use to code business services.

Abstraction by generalisation is a tool to be used with caution.

> The same high-level abstractions or generic classes that are so useful to the enterprise entity modeler and analyst, often turn out, when it comes to the detailed design of a specific system, to be only minor optimizations of a very small part of the application to be built.

24.3 Philosophical postscript: do we model entities or roles?

DAVID: In your criteria for a good class hierarchy you should add: Both the super-class and the sub-class must be true entity types, without roles included in their definitions.

GRAHAM: Your examples tend to persuade me that you model roles all the time (how things appear to an observer), rather than real-world things, or “true entity types” (how things are).

DAVID: In a sense you are right, but it's a different kind of role. This is a philosophical question, of course. The entities I have in my book seem pretty solid to me.

GRAHAM: I am thinking a little like a philosopher; I find phenomenology more relevant that ontology. But I thinking also of the sciences. In a psychology classroom, we learn the world is less solid than our egos let us believe. (Vanity, vanity, all is vanity.) In physics, cosmology and biology classrooms we learn that things evolve over time. Where is my grandfather's axe after my father replaced the handle and I replaced the blade?

DAVID: We are dealing with definitions of terms here. Of course these change over long time, but in my experience, the classifications I use are pretty solid. The issue of what constitutes an occurrence is a different one. Although I am not sure that as modelers we can deal with that one. An occurrence is whatever the people putting the data in say it is.

GRAHAM: Interesting class v instance thing here. You talk of the classifications being solid, but the occurrences being a different issue. Many think the reverse: real-world objects (instances) are tangible, but our classifications of them are fragile.

DAVID: So, you are Aristotelian and I am Platonic. To you physical things are the most real. To me the "idea of the thing" is the most real.

GRAHAM: Well, I do believe the real world exists. But I believe there are many equally valid views of the things in it, so there is no one “true” classification of those things. I haven't yet managed to reconcile quantum physics with cosmology and e = mc2.

25. Class hierarchies in practice

Pointers to where class hierarchies are and are not helpful.

There is a wide range of software design in which programmers find the inheritance mechanism provided by an object-oriented programming language helps them to economise, by reusing code.

This has encouraged software system designers to specify class hierarchies in the system specification, looking to maximise the use of inheritance. Some system analysts have now come to believe that they should be specifying class hierarchies wherever they can.

This chapter discusses some reasons why it is a mistake to try to impose class hierarchies on the persistent entities in the business services layer of an enterprise application.

Modelling business-world entities is different from modelling computer-world entities. It is easy to specify useful class hierarchies where the objects are records, transactions, windows, menus and command buttons. It is harder to specify strict class hierarchies where the entities represent persistent 'real-world' entities.

Fig. a shows our notations.

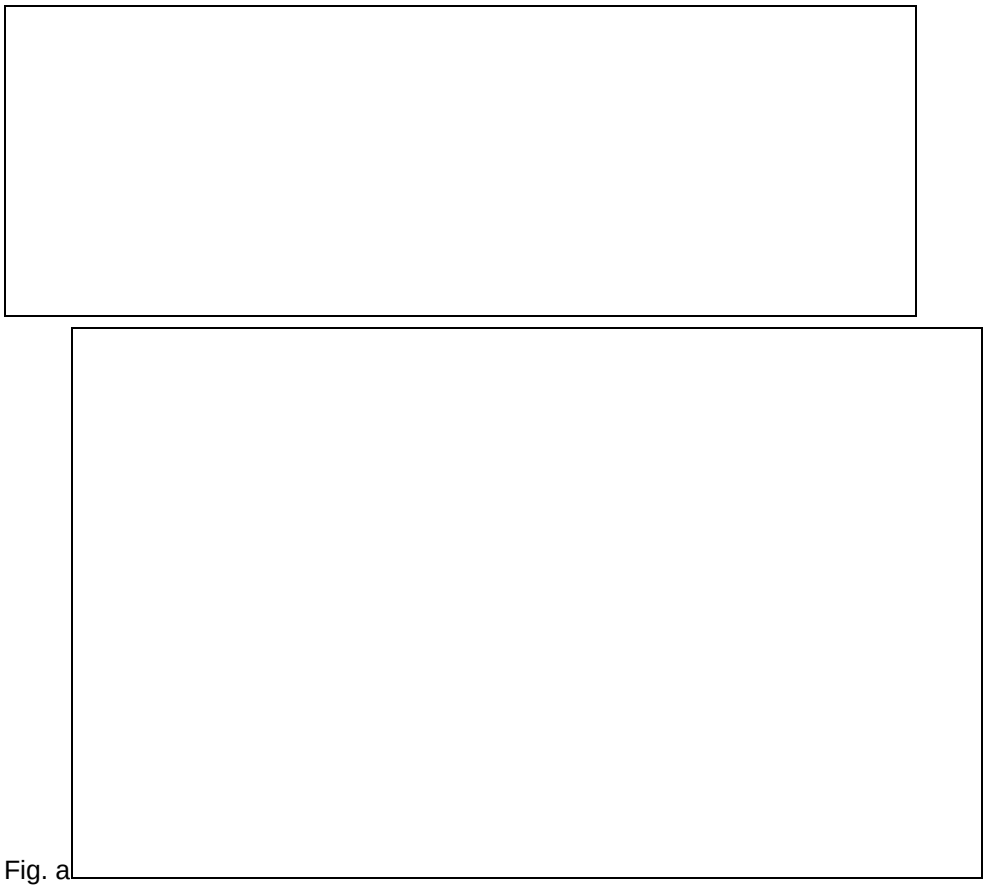


Fig. a

I will show why what appears to be a class hierarchy of persistent entities may better specified

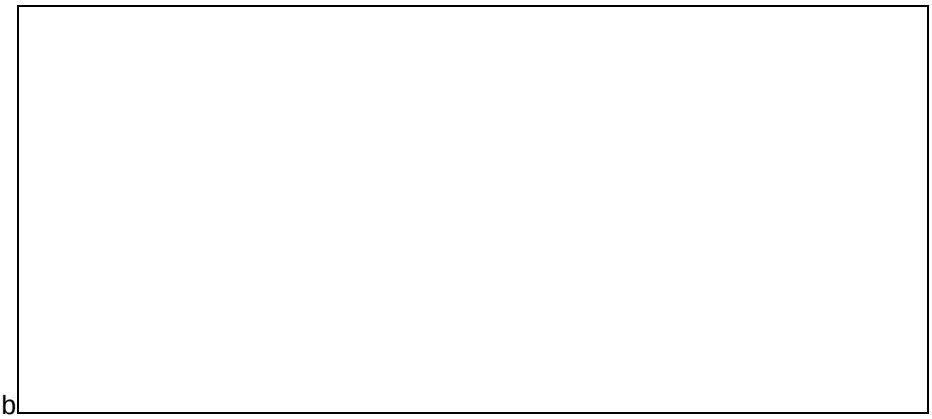
in a different way, often as an aggregate.

I will look in turn at: crude top-down classification, disputable divisions between subclasses, parallel rather than mutually exclusive classes, designer generalisation, and instances that change subclass.

Two themes run through all the sections below. First, the boundary between the classes of real-world objects or business entities becomes fuzzier the longer you look at them. Second, longevity turns types into states.

25.1 Crude top-down classification

Some people have proposed using stepwise refinement to develop not program structures (as Dijkstra proposed in 1972) but data structures. Their idea is to start with a few generic classes like Location, Person and Resource and then develop elaborate class hierarchies beneath them. Fig. b shows the start of this process.

Fig. b 

There are many difficulties with this top-down classification approach. One is to do with how fragile the notion of mutual exclusion is. What if a Person is both a Customer and an Employee? I have already considered this in chapter 4 and we'll come back to it later. I want here to challenge the broad 'top-down' approach rather than discuss semantic details.

The Chain of Christmas Trees pattern

The chief database designer a project in the US, after a brief introduction to object-orientation, led the analysis of the system by building four or five class hierarchies and connecting the top-level super-type objects by many-to-many relationships.

Fig. c gives an idea of what was done, except that there were about a hundred sub-types in each class hierarchy. I call this pattern the 'Chain of Christmas Trees'. It doesn't get you very far in software specification. This kind of model is so general that it has very little information in it.

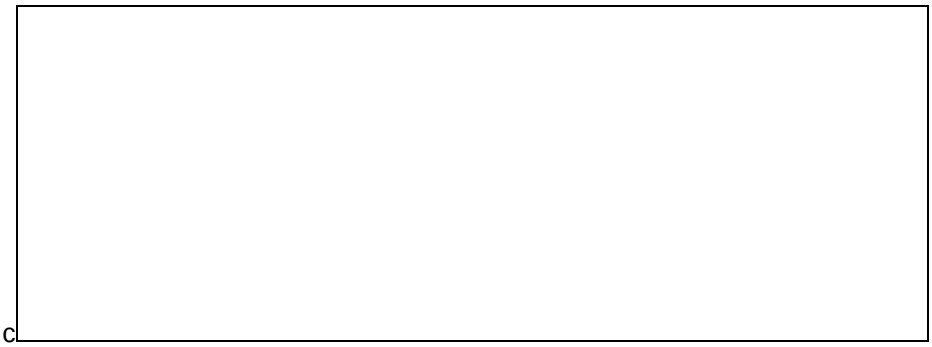


Fig. c

Where are the facts and constraints?

The specification above does not say how or why an Employee should be connected to an Office. What if there are really two meaningful relationships: ‘works at’ and ‘works for’? Fig. d shows you can do better by adding specifically named link entities between the specifically named subtype objects.

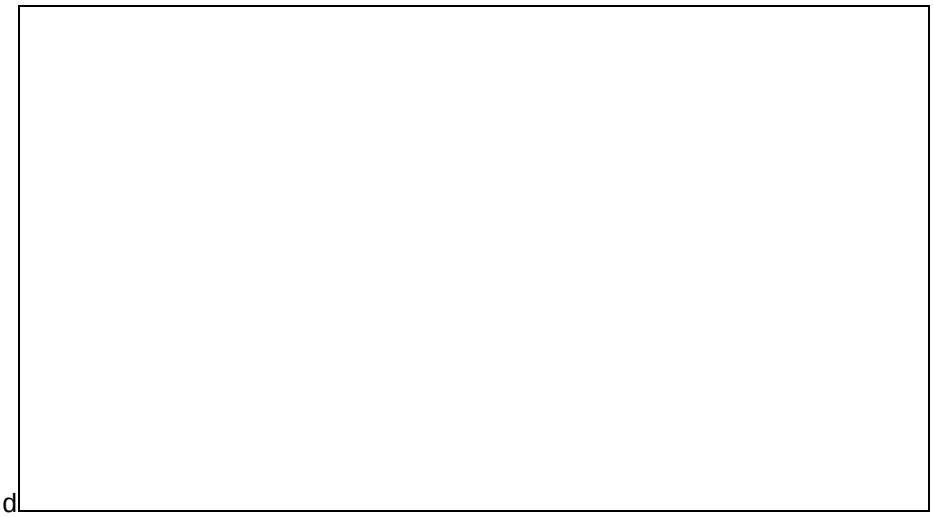


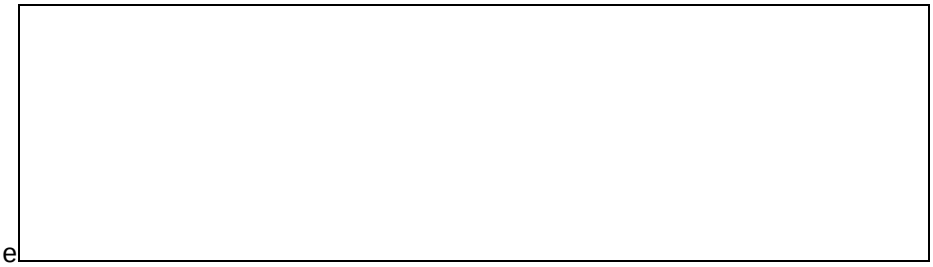
Fig. d

The two cross-references in the diagram above reveal two entirely different sets of Locations for a Person, and Persons for a Location. Once you realise this kind of analysis is necessary, the top-down development falls apart. You have to start again with a more traditional entity relationship modelling approach.

25.2 Disputable divisions between subclasses

In the world of mathematics, a circle and a square are different by virtue of their definition by mathematicians. In the world of computers, a file and a record are different by virtue of their definition by software designers. But in the natural world, things are not so sharply defined.

At least one author has claimed that class hierarchies in software design reflect the nature of the real world. Fig. e shows a favourite example, the biologists’ hierarchical classification of species.

Fig. e 

In fact, there is probably no such thing as a hierarchy in nature. The eminent biologist Richard Dawkins has pointed out that the higher levels of this classification are artificial, a man-made construction imposed on the fuzziness of nature.

You might hope there is certainty at the bottom level. Yet one species gradually (by tiny incremental changes) divides into two or more species, or evolves to become another. Obviously, the classification is transient over time. Less obviously, it follows that the edges of the classes are uncertain at any moment in time.

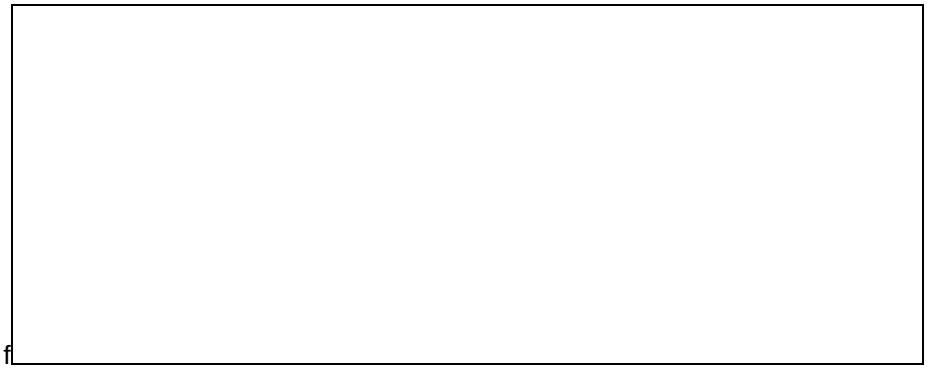
Even Darwin regarded the term ‘species’ as a ‘mere useless abstraction’ and ‘arbitrarily given, for the sake of convenience’.

A class hierarchy with uncertain boundaries between subclasses is a difficult thing to manage. People can find it hard to assign objects to a class. People will want to revise the class hierarchy in the light of new thinking and the discovery of new objects. Revising a class hierarchy is made more difficult when objects must persist from one version of the class hierarchy to the next.

25.3 Parallel rather than mutually exclusive classes

It is tempting to specify a class hierarchy where all but a very few objects are either one subclass or another. However, if the mutual exclusion is not an absolute rule, merely a *tendency,* then you will prevent the system from working properly.

Fig. f shows an example I came across in real system design. The Security class was specified as either Stock (earning interest) or Share (attracting a dividend).

Fig. f 

I soon discovered a few Securities that are both Stocks and Shares. (I might have guessed this sooner, on finding that the business gives every Security a unique number drawn from a single range, rather than drawing on separate ranges of numbers for Stocks and Shares.)

Users would be irritated by a system that made them record a Security as either Stock or Share. Some would have to enter the exceptional Securities twice. Some would fail to find all the information they want about such a Security because it has been split into two. Some would find that statistical reports are inaccurate.

Mutual exclusion rules that don't last

It is tempting to specify a class hierarchy if the mutual exclusion rule holds at specification time. However, the rule may break down; what seem to be disjoint subclasses when first analysed become parallel aspects soon after implementation.

A business might know that all its Vehicles are either Cars or Trucks, and all its People are either Man or Woman. But a thoughtful analyst should predict that one day the system will have to cope with a Vehicle that has the properties of both Car and Truck, or a Person who exhibits the properties of both Man and Woman.

Again, if you don't drop the mutual exclusion rule, the exceptional cases will have to be recorded twice in the system, or else processed outside the system. Fig. g shows the two data structures redrawn as aggregates to accommodate exceptions.

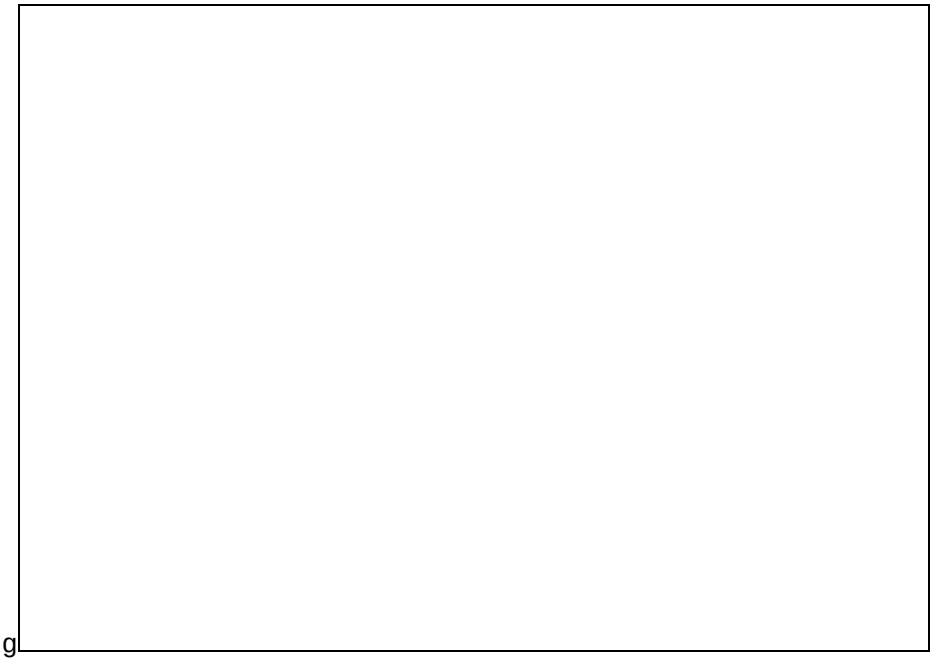


Fig. g

25.4 Designer generalisation

A real-world entity can play many roles at once. A business contact can be a customer and a supplier. A person can be a doctor and a patient.

The fact that these 'class hierarchies' are non-disjoint because some organisations are both customers and suppliers, and some people are doctors and patients is not the issue here.

The point here is that many businesses, much of the time, are perfectly content to record the distinct roles as distinct entities. The real-world entity is simply not important or valuable as a business entity.

Generalisation to track the real-world entities behind their roles

Designers sometimes create a class hierarchy in an attempt to keep track of the real-world entities that lie behind several business entities.

Suppose Healthcare administrators want their system to avoid printing out two Christmas cards to the same Person in the role of Patient and Doctor. The designer might specify Patient and Doctor as subclasses of Person, thinking this will help.

In general, such requirements are difficult to meet unless the users have a business identifier by which they can recognise instances of the real-world entity. The difficulty is that the business deals with the roles played by real world entities, not the entities themselves.

If the users have no way of identifying the real-world entity that lies behind several roles in the system, defining a class hierarchy is not going to be helpful. Fig. h shows you could instead specify the real-world entity as a kind of link class, related to its roles as business entities by associative relationships.

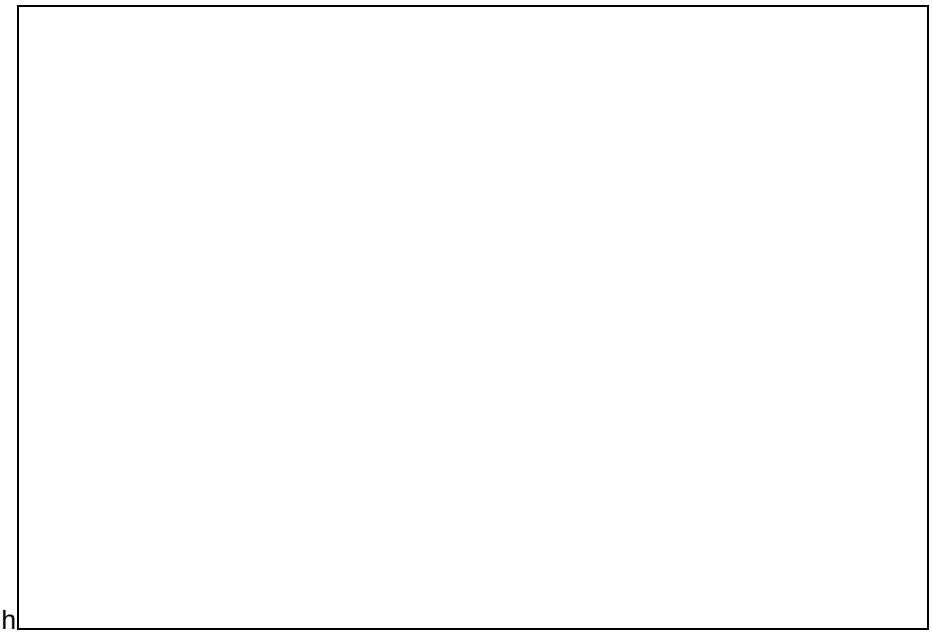


Fig. h

If the objects of the link class are visible to users, distinct from their roles, the link class should be given a meaningful key, even if this is a compound of all its attributes.

Generalisation to optimise design

Designers sometimes specify a class hierarchy to avoid duplication of specification and code. Fig. i shows an example where designers specified a class called Organisation to hold address details and a telephone number.

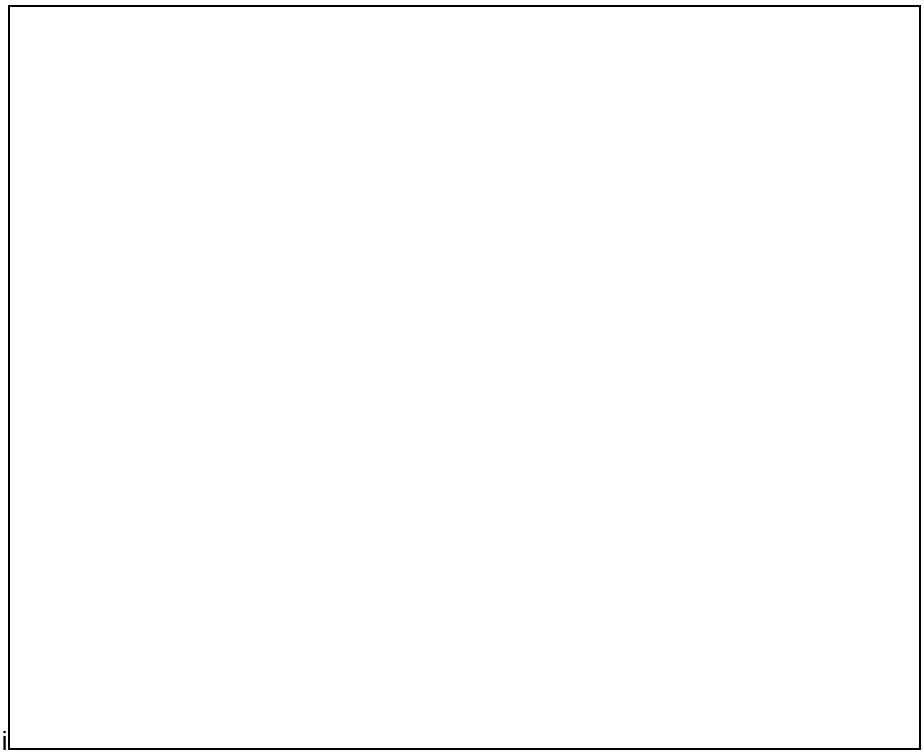


Fig. i

But the business deals with Customers and Suppliers, not Organisations. Users are quite happy to record details twice for the small number of Organisations that are both Customer and Supplier.

Indeed, users may want to record different addresses and telephone numbers for each role. They may even want to record several addresses for a Customer or a Supplier.

Where design optimisation is the motivation, you might respecify the superclass as a link class, related to the business entities by associative relationships.

25.5 Instances that change subclass

Persistent objects can hang around for a long time. In many of the class hierarchies people try to impose on the real-world, an object instance may change from one subclass to another. Fig. j shows two mistakes I have come across in real system design.

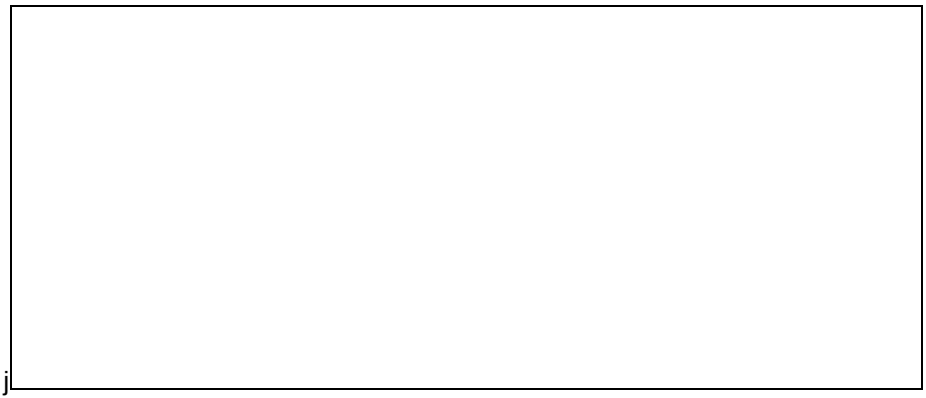


Fig. j

You might put the model right by specifying the superclass as a master, related to ‘period of time’ as a detail, and then subdivide ‘period of time’ into the different classes.

However, in practice, this is one of many cases where the subtyping is better specified as a cycle of states within a class than in the entity model. State machines are a better specification vehicle than entity models for capturing business rules of this kind. See chapter 8 for further discussion.

25.6 Where class hierarchies are useful

Class hierarchy are most useful where the hierarchy:

• is firm not fuzzy, has strictly disjoint subclasses

• persists as a definition as long as the lives of objects it defines.

Where in enterprise applications are these true? Fig. k is a picture that divides an enterprise application into three layers and helps us to show where class hierarchies are most useful.

Fig. k

Generally speaking, it is easier and safer to define class hierarchies of computer-world objects than of business entities. So class hierarchies are more useful in the technological layers of

design than in the business services layer of the three-tier architecture.

This is not a popular view with those object-oriented designers who want to prove the value of inheritance in the business services layer. But it is a fact of life.

Computer-world objects in the UI and data services layers

When you are constructing UI layer components, you have the power to order and classify them as you like.

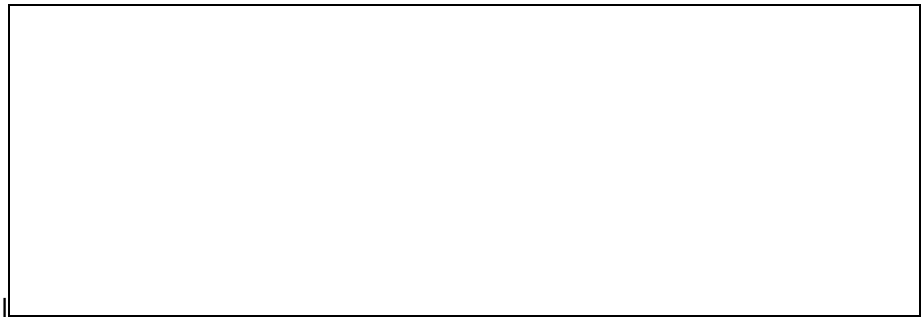


Fig. l

To a lesser extent the same is true of the data services layer. It might make sense to look for inheritance trees in the data services layer or the Data abstraction layer. (You can restructure the classes in the data services layer whenever you like, if you can bear the costs of data migration.)

Transient event classes in the business services layer

Since class hierarchies are much more common where the objects are short-lived, they are more likely to be found among transient event classes than persistent object classes.

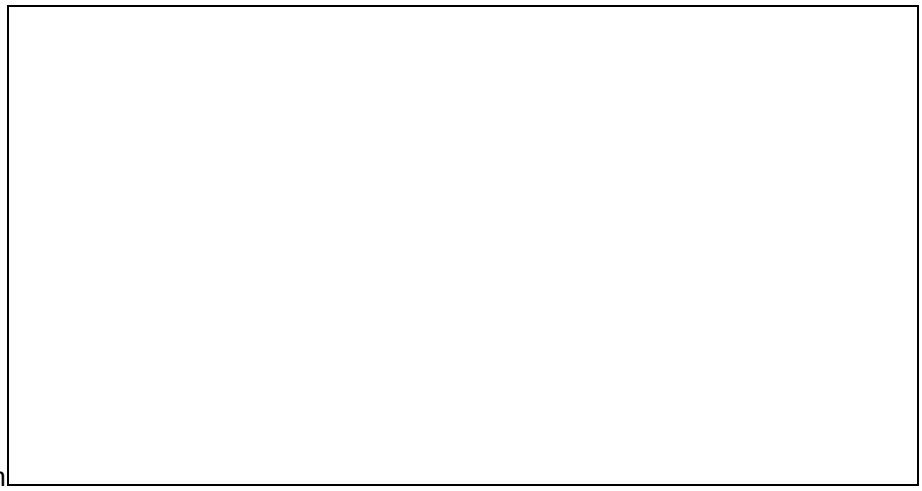
Robinson and Berrisford (1994) suggested that in enterprise application development you can normally make greater reuse of superevents (transient business objects) than superentities (persistent business objects).

Model transformation for schema integration

Schema integration is a one-off exercise. If you plan to merge two schemas, you do need to make the current rules (terms, facts and constraints) fully explicit. You can be confident that the range of a type, the instances of a class, the rules of the business, will not change while you are working. You might well convert any range of subtypes into a class hierarchy, just for the purpose of comparing schemas, as discussed in the volume ‘Introduction to rules and patterns’.

Data item definition

A class hierarchy may be used as a device for structuring the contents of a data dictionary, where all that is required is to allocate the various data items as attributes of entities. Fig. m arranges the data items involved in a hospital system using the notion of a class hierarchy.

Fig.6m 

This model is not good enough for building an enterprise application, since the relationships between entities are specified incorrectly. For system design, the model must be redrawn after considering:

• a person may be both a doctor and a patient

• a doctor may swap between working in a hospital and family practice

• a person can be a patient on many occasions.

Specifying the classes in the form of a class hierarchy fails to capture any of these rules.

It is much better to specify the 'subclasses' as 'parallel aspects' connected to a 'basic aspect' by associative relationships. Not only will this reduce schema evolution problems if the system ever has to record the history of a person over time, *it is a more accurate specification of the real world even if this day never arrives.*

Generic domain classes

As soon as a type of information is identified as generally reusable in lots of businesses, technology vendors can make a profit by selling it to business system developers.

There is some scope for defining class hierarchies among universal domain classes such as text, number and date.

It is hard to imagine how a business can gain a competitive advantage by focusing its efforts on specifying hierarchies of classes that it shares with other businesses. Technology vendors can supply generic class definitions. Packages should take over here. System developers will always be left to define the business-specific classes, the ones that cannot be reused so widely.

Generic business entities?

While class hierarchies are useful specifying transient objects in the more technology-bound layers of the three-tier software architecture, it is hard to find useful class hierarchies where the objects represent persistent external real-world entities,

If you have a large business information database that has more than a small class hierarchy in a corner of it - please show it to us.

The only convincing examples I have seen to date are drawn from financial systems. The

superclasses are things like Account, Transaction and Deal. These are frequently presented in books and talks.

These are highly generic entities - not specific to one business. They are also highly abstract - not as nearly as concrete as classes like Employee, Building, Ship and Engine.

25.7 Summary

Modelling real-world objects is different from modelling computer-world objects. It is easy to specify useful class hierarchies where the objects are computer-world entities such as records, transactions, windows and command buttons. But I am convinced that class hierarchies are of very limited value in the business services layer of database transaction processing systems.

Grady Booch has said to us 'It is a mistake to search too aggressively for class hierarchies and inheritance in the entity model of a business. Sometimes its there and it pops out at you. More often there aren't many class hierarchies to be found. Many things that look like mutually-exclusive subclasses turn out to be what you call parallel aspects. Where you do define a superclass, it must have common behaviour as well as common data attributes.'

It turns out that in specifying the essential processing requirements and constraints in the business services layer of an enterprise application, more reuse can be achieved in other ways

There is more reuse to be found by *associative relationships* than by inheritance relationships. To discover and specify this kind of reuse, object-oriented analysts need to think in terms of parallel aspects rather than class hierarchies of mutually exclusive subclasses.

There is also more reuse to be found between transient *events* than between persistent objects. Transient event class hierarchies are more useful than persistent object class hierarchies. To discover and specify event class hierarchies, object-oriented analysts need to add an event-oriented perspective to their existing object-oriented perspective.

In short

object-oriented technologies that support inheritance do help designers working in the technology-bound UI and data services layers, but the same principles do not provide analysts with the 'leap forward' that people have hoped for in the business services layer.

The persistence of objects and the fuzziness of the real-world makes it harder to specify strict class hierarchies where the objects are models of external real-world entities. More reuse can be achieved in other ways. object-oriented theorists need to take these other ways on board. A complete object-oriented systems development approach must help us to recognise and model the events as well as the objects.

26. Interpreting polymorphism as event effects

Polymorphism is a powerful programming tool; but it is easy to cheat, to create an abstract class where there isn't a natural class hierarchy.

> 'uncontrolled polymorphism would be incompatible with the concept of type'. Meyer (1988) says

It is possible that some object-oriented designers apply the ideas of type and polymorphism outside of the proper context, because they lack the concept of an event. We need a design method and a language for talking about events as well as objects. A helpful definition is:

An event is a minimum unit of consistent change to a system, a set of effects on one or more objects which must succeed or fail as a whole.

Given this definition, an event may affect several different objects of different classes. It is a good idea to record all these effects in an event model (object interaction diagram or use case) for the event.

When type should be state

The State design pattern might be used to make the effects of one event look like a polymorphic method. Say the Death event has different effects on a Person depending on whether the Person is employed or not. The designer might describe these as polymorphic methods depending on the type of an object. To the analyst they are **optional effects** of an event, depending on the *state* of the object.

When type should be role

Say the Death event has different effects on the Person (dead) and the Employer. Again, these effects are not best described as polymorphic methods depending on the type of an object. They are effects of an event on objects of one class playing **concurrent roles** with the respect to the event, depending on the *identity* of the object.

When abstract class should be event manager

An event model implies a weak kind of polymorphism. You can (we may say should) name all the methods in an event model after the event that fires them. The methods share the same name, but not the same effect, and not necessarily the same interface. In some cases you may be able to define a common interface for all methods fired by an event, perhaps:

input: event name and parameters

output: OK or Error code.

You might then create an abstract class for the event type, which is a supertype of all the classes that appear in the event model. Designers do indeed define such event classes in processing transient objects in the UI layer. But processing objects in the business services and database layers is different.

Classes of transient events appear in enterprise applications in the guise of 'event managers'. An event manager is a transient process that handles one event instance. It is a convenient home for things like transaction management and error handling. However, analysts do not normally include event managers in the entity model that specifies the data structure of

persistent objects.

We need a richer theory, one that accommodates type and state, objects and events, inheritance and polymorphism one the one hand, event managers and multiple event effects on the other.

27. PART FIVE: AGGREGATE ENTITIES

28. Aggregate/components entities/classes

Abstract

"Aggregation and Composition are one of my biggest bete noirs" page 80 of 'UML Distilled' Martin Fowler.

Should we specify a coarse-grained entity/class such that it contains several finer-grained entities/classes? Or specify only the finer-grained entities/classes? (Leaving composition to be done by processes when needed).

Should we specify an entity/class to have a multi-valued attribute? Or specify classes to have only single-valued attributes?

These are questions over which object-oriented designers and entity modelers might come to blows.

I have corresponded with an object-oriented guru, James Rumbaugh, about aggregate entities and the related issue of multi-valued attributes.

Some of our correspondence is included in the paper below. It tends to the conclusion that aggregation is used to optimise a physical design, but adds little or nothing to a purely analytical business rule model.

28.1 Data structures in enterprise application software development

Enterprise applications typically maintain a persistent data structure, which is describable as a set of related entities. E.g.

Customer ---< Order ---< Order Item >---- Product

Enterprise applications typically consume and produce input/output flows that are aggregate data structures.

A use case/session is usually supported or enabled by a particular data structure displayed at the user interface. E.g.

1. In the "order capture" use case, the customer enters or views an Order-topped data structure: Order ---< Order Item.
2. In the "study product demand" use case, the product manager views a Product-topped data structure: Product ---< Order Item.

The Model-View-Controller design pattern is used by OO programmers to structure the software that supports a use case/session:

The Views represent the HTML pages displayed during the session.

The Controller handles the events and enquiries entered during the session, which either update data in the Model or fetch data to populate a View.

The Model is the data structure relevant to that session. E.g.

1. In Model 1, the Order Item appears as an element in an Order aggregate data structure.
2. In Model 2, the Order Item appears as an element in a Product aggregate data structure.

A user interface naturally represents an I/O data flow in a hierarchy and sequence that aggregates elements in one direction rather than another.

This paper is about whether the underlying Business Model should aggregate the Order Items in any particular direction.

28.2 Aggregate entities

Specifying a business rules model involves a certain amount of unavoidable work.

If your model is detailed down to the level of fine-grained normalised entities, then all one-to-many relationships are made explicit in the structure.

If not, then these relationships must still be documented, not in the entity model diagram, but behind it.

People seem to prefer abstracting from a fully normalised entity model, to specify coarse-grained classes.

Some simply want suppress detail from the diagram, perhaps to facilitate discussion with users.

Others are looking to define fewer, coarser-grained classes for the purpose of defining fewer, coarser-grained operations.

How is this done?

In practice, the grouping of entities into aggregate entities is normally done in one of the following two ways.

28.2.1 Aggregation based on kernel entities

A **kernel entity** is one with a simple primary key recognised by users (say Order).

A **characteristic entity** is one whose existence depends on a kernel entity (say Order Item).

You can define an aggregate entity (also called Order) composed of the kernel entity with its characteristic entities

Any entity who existence depends on the existence of a kernel entity, might grouped into an aggregate with the kernel.

Typically, a dependent class can be recognised by the fact that its primary key is an extension of the kernel entity's key.

For example, in this model:

Customer ---< Order ---< Order Item >---- Product

Order ---< Payment

Suppose the primary keys of the Order Item and Payment classes are formed by extending the Order Number with a further identifier.

Then you might define an aggregate composed of Order, Order Item and Payment.

Coarser-grained classes mean coarser-grained operations.

28.2.2 Aggregation based on aggregation/composition relationships

UML includes aggregation and composition relations, which may be drawn between two classes

Aggregation connects a component class to the class that “contains” it in some sense.

Aggregation is a vague concept that can be given more substance by thinking of the associated objects’ state machines.

Do the associated objects share the same lifetime? even the same state transition diagram?

“An aggregation relationship implies that the object and its owner have the same lifetimes” Gamma etc. Design Patterns' Prentice Hall 1995

Consider two classes – parent and child – in which the identity of the child entity is constructed by extending the identity of the parent entity.

- The parent entity - Order - has the primary key Order Number.
- The child entity - Order Item - has the primary key Order Number + Item Number.

Are these two classes? Or one aggregate class that contains the other?

James Rumbaugh: “In this case I would treat them as separate objects.

You want to manipulate them separately and treat the order item as something you can change, with the changes reflected to the order.”

Graham: One might ask: Do you ever invoke an operation on a child object without invoking an operation on its parent?

If yes, then they are better specified as two distinct classes.

E.g. Do you ever enquire on, update or delete an Order Item without accessing its Order?

If yes, it is better to specify Order Item as a distinct class.

Composition is a tighter form of aggregation.

James: Note the concept of "composition" that we have added to the UML.

“A composition is a strong aggregation in which the composite is the sole owner of the part and is responsible for its creation and destruction.

The part cannot exceed the lifetime of the composite.” James Rumbaugh 1998

Graham: Is the concept of composition helpful here?

Does it add much to the relational database principle of referential integrity between child and parent.

The principle that the ‘child cannot exceed the lifetime of the parent’ applies to any child-parent association where

- the child’s primary key includes the parent’s key, or
- the relationship is mandatory and fixed at the child end.

James: “I’m not sure you can distinguish composition and association so well in the real world, or in an analysis model.

But at the design level I think it is clear.

The composite has responsibility, and sole responsibility, for the memory management of the part.

There will be no conflict over the reference and no danger of dangling pointers if the owner deletes one.

It is a guarantee that there will be no garbage collection problems.

Therefore physical embedding is composition, because the part is allocated and de-allocated with the whole.

But you can use a pointer to memory off the heap, but it may not become independent.

That is the meaning of composition in a practical sense: it doesn't matter if the part is physically part of the whole or stored in a separate block, it is handled the same way.”

Graham: In any case, the UML definition of composition isn’t applicable to this case study.

Because Order Item is owned not only by Order but also by Product, it is a cross-reference between them.

An Order Item cannot exceed the lifetime of its owner Order, but nor can it exceed the lifetime of its owner Product.

(Nor any other fixed mandatory parent, including indirect parents such as the Customer who placed the Order).

There is one more analysis question to be asked of the case study example.

Can an Order include two Order Items for the same Product?

If not, then you might do better to redefine the primary key of the Order Item as a compound of Order Number and Product Type.

And so prevent several items on one Order from requesting the same Product.

28.2.3 The weakness of the aggregation concept

Martin Fowler again: “Aggregation is easy to explain glibly.

The trouble is, there is no single accepted definition of the difference between aggregation and association.

In fact, very few [methodologists] use any kind of formal definition.”

The aggregation concept seems natural where object-oriented programming was first successful, that is, in the handling of graphical user interface objects.

Consider a dialogue box that can be moved across the screen.

All the objects (buttons, fields, whatever) within that dialogue box have no existence before, after or outside the box, and must move with it.

There are three reasons why aggregation relationships between business information classes far less natural.

1) There is a scale of associations from weak to strong

“Aggregation and acquaintance relationships are easily confused” Gamma et al. ‘Design Patterns' Prentice Hall 1995

It is easy to waste time arguing about the distinction.

It seems better to regard association relationships as being placeable on a sliding scale from tight aggregation relationships to loose acquaintance relationships, without any firm dividing line between them.

2) Aggregation and association relationships are normally implemented the same way

They cannot be distinguished in the compile-time structure or the implementation language.

Martin Fowler again “The cascade delete is often considered to be a defining part of aggregation, but is [clearly applicable to structures that are not aggregations].”

3) Persistence undermines aggregation.

Time destroys compositions.

Time reveals apparently strong aggregation relationships to be weak associations or acquaintance relationships.

Events break up aggregates (the Soviet Union, Yugoslavia and the United Kingdom come to mind).

In short, aggregation is a fragile concept.

Aggregation relationships seem natural in a static unchanging data structure, in a short-lived data structure, in a transient input/output view (like a graphical data structure perhaps).

But where entities persist and change over time, the concept of aggregation, composition or containment is a weak one.

28.3 Multi-valued attributes

There is a debate about whether an attribute can have plural values.

According to most people’s interpretation of relational theory, an attribute that is plural (e. g. Telephone Numbers) ought to be specified as an independent class of objects.

Applying this ‘first normal form’ principle sometimes leads to a different model from object-oriented design, which allows what might be called an aggregate entity that contains multi-valued attributes.

Customer entity: attribute list
Customer Number
Name
Country
Telephone Number
Telephone Number
Telephone Number

James: I wouldn't rule out attributes with plural multiplicity, in the situation that the set of values has no independent identity.

Graham: Asking about uniqueness constraints might help.

Do users care enough to prevent an attribute value appearing twice in the list? If they don't care, then the multi-valued attribute is reasonable.

If they do care, this implies that each attribute value uniquely identifies something (a telephone), and suggests you should normalise it to become a distinct class.

James: I might put it differently.

If the reference is to a telephone as an object, you could change the number and all of the users of the telephone would see the new number.

If it is just a string in each list, then it has no identity.

Graham: There is also a more practical design question. How long is a list of attributes?

A database designer might not like either a long fixed-length list (empty in most cases) or a variable length list.

James: As an implementation issue you would likely make it (Telephone) a class, but that's not the modeling issue.

Graham: Strictly, the designer or implementer's decision should not influence the builder of the business rules model.

But it is often tempting to align the entity model with the database model.

And in the case of multi-valued attributes, it normally seems convenient and harmless to do so.

28.4 Conclusions

It is easy to draw an aggregate entity box around a parent entity and some or all of its children.

But does such composition actually help?

28.4.1 Composition in the UI layer

User interface designers find composition useful.

A window or dialogue box can be regarded as an aggregate entity.

The user may trigger operations that act on the whole aggregation, rather than elements of it.

28.4.2 Composition in the data layer

Database designers find composition useful.

Group entities into an aggregate entity may imply the clustering of tables into a ‘block’, so as to speed up processes that access data from closely related entities.

28.4.3 Composition in the business layer

Composition does seem a good way to document the relatively uncommon situation where a class has multi-valued attributes.

Such as the list of telephone numbers illustrated earlier.

And composition might also be used in circumstances constrained by the following two rules.

Include only single-parent characteristic entities with the parent in the aggregate entity.

A single-parent characteristic entity is a child that has no relationship other than to its parent.

Do not include any characteristic entity upon which operations can be invoked separately from the parent entity (which implies there is a distinct identifier for the characteristic entity).

But many people are insensitive to these constraints.

They often include link entities, and sometimes even parent entities, in aggregate entities.

They may do this for many reasons, but their reasons are always to do with design rather than analysis.

There is a limit to how much I can conclude from an email correspondence.

But I have the impression that James Rumbaugh also regards most composite classes as artefacts of physical design rather than analysis.

The business rules model is not supposed to be influenced by the design decisions of user interface designers and database designers.

There are only a very few cases where the concept of an aggregate entity seems helpful in a business rule model.

Beyond these cases, specifying composite classes in a business rules model seems inappropriate and unhelpful.

It forces the analyst to make a design decision that need not be made, a decision that often has to be

undone later.

28.4.4 Composition to suppress detail

Some people need to suppress detail from the entity model diagram, perhaps to facilitate discussion with users.

This is a harmless enough idea.

If the aim is merely to draw a higher-level of entity model that suppresses detail from the full one, then a simple procedure will do the job:

Examine every entity that has only one relationship.

If this entity is not of primary importance to the users, then hide it inside its neighbouring entity.

Examine every many-to-many link entity.

If this entity has no attributes or operations of its own, then draw the entity as a relationship line rather than a box.

28.4.5 Composition for CBD, microservices or micro apps

Some people are looking to define fewer, coarse-grained classes for the purpose of defining fewer, coarse-grained operations.

However, composite classes or aggregate entities are simply too small for component-based development.

This is the reason why people complain that the granularity of object-oriented design is too small.

The practical way forward is to move towards a component-based development approach where many classes are grouped into a coarse-grained business component, and operations are defined at the level of the business component.

The component-level operations, or business services, correspond to what most pre-object-oriented methods called transactions.

29. PART SIX: DEEPER THOUGHTS ABOUT ENTITY MODELS

30. Five kinds of entity model

What is a class? What does a box in an entity model represent? Should either subclasses or parallel aspects of a class be specified as distinct classes? You can't have it all ways in one entity model.

This chapter shows five kinds of entity model. A methodology or technology can accommodate clashing views of what a class is, by allowing several versions data structures to coexist, each entity model being directed at a different purpose.

30.1 The inevitability of structure clashes

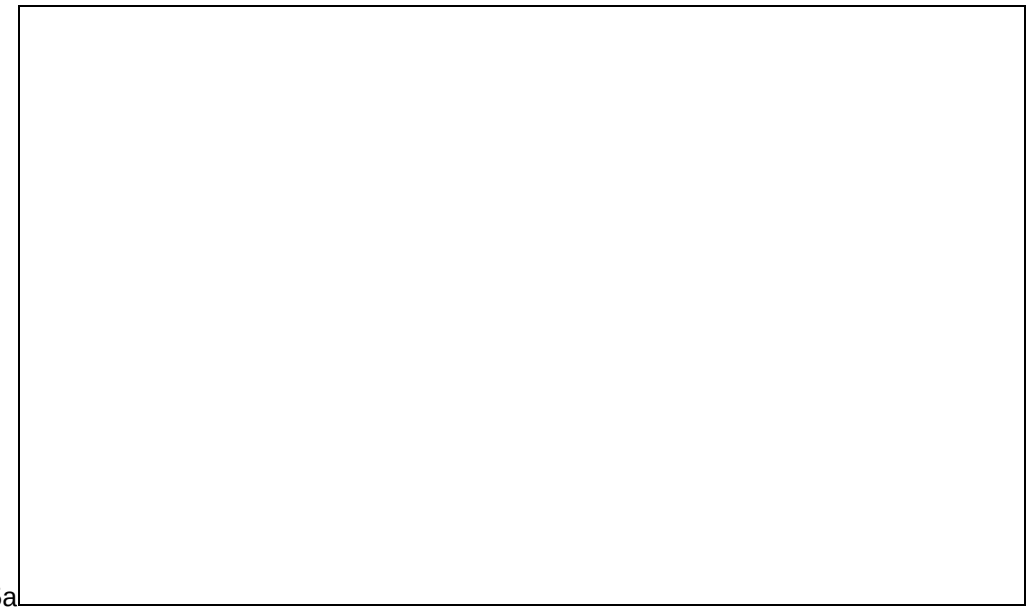
Booch (1994) probably speaks for most computer scientists when he says 'mapping an object-oriented view of the world onto a relational one is conceptually straightforward, although in practice it involves a lot of tedious details'. We cannot deny the 'tedious details', which means that even trivial examples of enterprise applications take up a lot of space, but this chapter offers a challenge to the 'conceptually straightforward'.

Although the individual data items of system specification are very important, we are more troubled in enterprise applications by how to specify the rules and constraints governing larger objects, higher-level relations or aggregates of data items. Aggregation becomes an issue.

Whereas *inheritance* implies the generalisation of mutually exclusive subclasses into a higher-level class that may be smaller than its subclasses, *aggregation* implies the summation of parts into a higher-level class that must be larger than its parts.

There are clashing views of how to aggregate properties to form an object class. Different ways to group data items into classes lead to different entity models, that is different structures specified over the top of the data items. We need to understand the different possible views and their implications.

A 3-tier software architecture can reconcile various views by handling each view in a separate layer. Fig. 5a gives a rough idea of what we mean. The chapter will explain.

Fig. 5a 

30.1.1 Structure clashes

Aggregation is the means by which simple elementary components are added together to form larger and more complex structures. You can build one set of elementary components into an infinite number of different, clashing, higher-level structures.

For example, a school geography book presents several maps of the same area, showing different divisions of the earth's surface into:

• land masses bounded by oceans

• countries bounded by political administration

• territories bounded by climate or vegetation.

It would be foolish to say that any one of these is the 'right' view. And there are further clashing aggregations of people: by race, religion and native language. Fig. 5b shows part of the world where aggregations have been disputed for thousands of years.

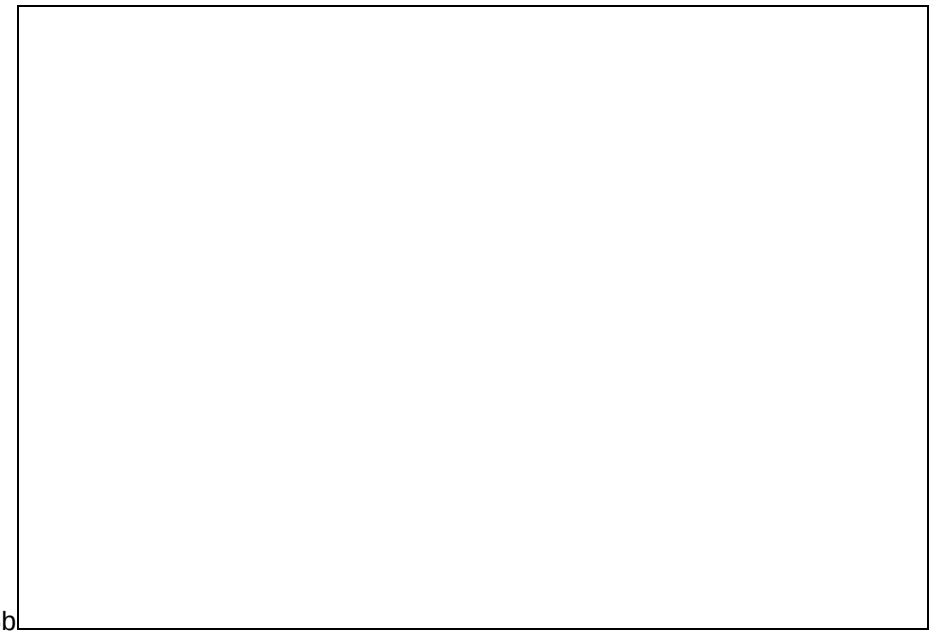


Fig. 5b

The ‘correct’ aggregation of components in this example has long been a matter of dispute. Land borders have been redrawn several times. Peoples have moved en masse between Scotland and Ireland (and from both to the USA). Eire, the ‘Irish Free State’, separated from the United Kingdom as recently as 1921.

Components in this example are members of higher-level aggregations, Europe, NATO, the United Nations, and so on. There are further structure clashes between these higher-level aggregations. Again, it would be foolish to say that any one view is ‘right’.

30.2 Classes as elementary data types

Starting at the bottom level of system specification, the atomic particles are the data items or variables. Each variable has a data type that defines its valid range of values. The volume ‘Introduction to Rules and Patterns’ showed how you may declare the data type as what may be called domain class.

Generic domain classes such as Text and Integer are widely reusable. You may define more application-specific domain classes such as Telephone Area Code and Country Name for reuse in a local context.

Where several variables (say, Price and Telephone Area Code) share the same domain (say Integer), the variables are subclasses that inherit the properties of the one domain. You can arrange domain classes themselves into a complex inheritance tree.

Early object-oriented authors such as Meyer (1988) were much concerned with designing production software operating at the level of domains. They sought reuse of code via inheritance between domain classes. Some wanted object-oriented to provide a once-and-for-all inheritance tree of domain classes, and so save them from have code from scratch each new programming language, operating system and CASE tool.

But for enterprise application specification a simple two-level structure of domain classes is

usually enough: generic domain class (say, text) and application-specific domain (say Country Name). In practice, Analysts spend little time on specifying domains. They spend most of their time thinking about the rules operating on larger relations, each an aggregates of variables about a business entity.

Specification of classes at the level of relations is different from specification of classes at the level of domains. Among database-oriented authors, Chris Date (1994) makes the remarkable statement: 'Object classes are domains or data types. Questions about inheritance therefore apply to domains, not to relations.' Is this true? What is a class anyway?

30.3 Classes as aggregations

To make an aggregate, you simply add components together. There are an infinite variety of way to group data items into aggregate classes. It is hard to make any general statement about what an aggregate class is, until we separate the three layers of the three-tier architecture.

Layer	Objects like	In domain of
UI	menus, windows, buttons etc.	user interface technology
Business services	business entities and rules	business users
Data services	tables, records, indexes, etc.	database technology

This three-way separation of concerns recurs throughout our work in information analysis. It helps us to separate different kinds of problem, and retain this separation from analysis and specification through to coding. It enables us to change the data storage or UI layer with minimal disruption to the essential business components.

30.3.1 Classes in the data storage structure

Aggregate classes appear in the form of persistent database tables or record types. Database programs treat each table as a distinct object. Some database management software expects you to specify larger aggregate classes such as database blocks or pages.

The size and scope of each aggregate class in the Data services layer is physical. It is guided by considerations of efficiency and limited by the database technology. Each internal class may roll up data from several entity classes, or store only part of one entity class.

30.3.2 Classes in the UI layer

Aggregate classes appear in the form of transient input messages, output messages, windows and dialogue boxes. The GUI management software will treat the data structure in a window as a single object when, for example, it moves it around the screen.

The size and scope of each aggregate class in the UI layer is physical. It is guided by considerations of usability and limited by the user interface technology. Each UI layer class may roll up data from several entity classes, or display only part of one entity class.

30.3.3 Classes in the business services layer

The Business services layer includes both persistent aggregate classes (objects) and transient aggregate classes (events and enquiries).

Events and enquiries are naturally limited in size and scope by the rule that a transient event is a *minimum* unit of consistent change. But what rules limit the size and scope of the persistent object classes? In other words: How should you group the persistent variables into aggregate classes in the entity model?

The size and scope of each aggregate class in the business services layer has nothing to do with technology, or physical objects such as database tables or GUI windows, it is a matter of logic.

Three logical views

You might define a class as an aggregate of variables around three different centres:

aggregate centred on	**means the class is**
a key	a third normal form relation
a type	a type within a class hierarchy
a state variable	a state machine (or 'parallel aspect' of a relation)

So far, authors have tended to gloss over the possibility of structure clashes between these different views of a class. In trivial examples, where there is no structure clash, the different definitions lead to the same set of classes. You will draw the same data structure whichever definition you pick.

But for non-trivial enterprise applications, we can no longer pretend that the different definitions give the same answer. The more complex the system, the more the logical views diverge from each other. The data pictures you draw will depend on the definition you pick.

There are clashes between logical views of a class, and then between logical and physical classes. You may design the physical database tables and GUI windows to match either the relations or the state machine, but you can't match both. We need a richer theory of system specification, a software architecture that results in separates components handling each logical and physical view.

30.4 A more formal view of entity models

Entity model notation is not the issue here. The notation we use is only one of several possible notations. The same questions and choices arise whatever notation you use.

Different ways to group variables into classes lead to different entity models, that is different structures specified over the top of the variables. Analysts often draw entity models that are mixture of different styles without realising it. They should understand the different possible approaches and their implications.

30.4.1 The informal entity model

At the earliest stage of system specification, you should be free to draw any informal picture that helps you. For the sake of giving this kind of data structure a name, we'll call it an 'informal

entity model'. This is an informal picture of the data in which an 'entity' is whatever level or size of data group you want it to be.

30.4.2 The relational entity model

Given an enterprise application consumes data and produces information, you may uncover the classes of interest by relational data analysis of the data in forms, reports, screens and files. The result of such analysis is a relational entity model.

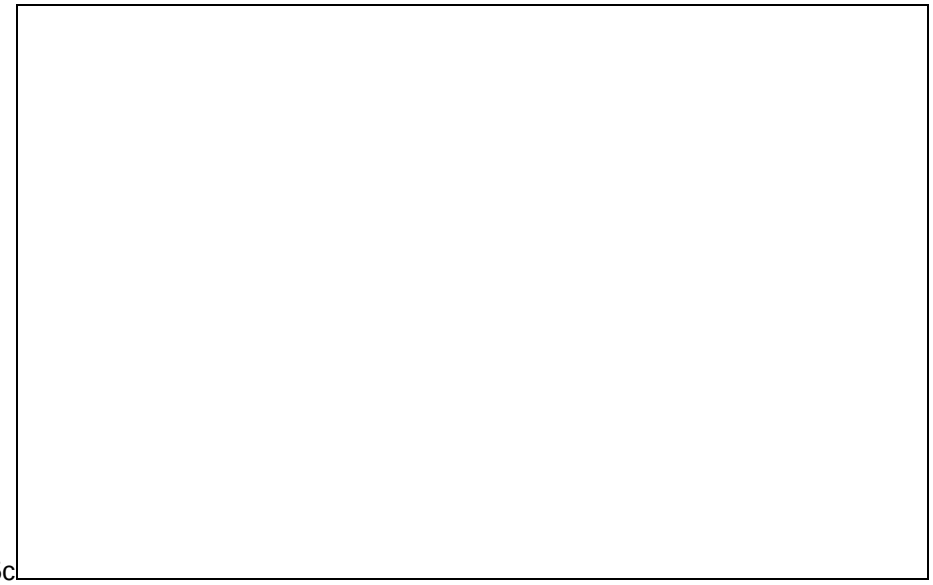


Fig. 5c

An early step in relational data analysis is to spot the business identifiers or keys. Given an object, the value of each of its attributes is uniquely determined by the value of its key. In what is called a 'third normal form' relation, the value of each attribute is determined by first the key, second the whole key, and third nothing but the key.

In the example above, each box is a relation, its key is underlined, its attributes are listed, and its associations to other relations are shown as lines connecting the boxes. The meaning of the different styles of line doesn't matter here.

There is little freedom of choice about what the relations are, given that you know the users' information requirements and you follow the idea that object instances are uniquely identifiable from each other by a key. But this is not the only logical view.

30.4.3 The entity state machine model

Another common logical view is that an object is something that progresses through a defined series of states, from a beginning to an end. In this state-transition view, a class is a state machine or behaviour, governed by a state variable (SV in the illustrations below).

People who design systems with little or no persistent data, typically embedded or process control systems, often view classes as state machines. They discover the classes by analysing states that classes pass through and the events that trigger state-changes.

You can use similar techniques for enterprise applications. You can transform a relational

entity model via behaviour analysis techniques into a specification of event models that maintain the data. These event models include the required rules in the form of preconditions. It turns out that behaviour analysis may lead you to decompose a relation into parallel aspects. In the event models, each parallel aspect of a relation appears as a distinct class.

People often assume you can draw one state machine for each relation. However, you need to draw separate state machine for the parallel aspects of a relation.

Fig. 5d shows the entity model with one box per state machine. This differs from the relational view in that parallel aspects of a relation have been divided.

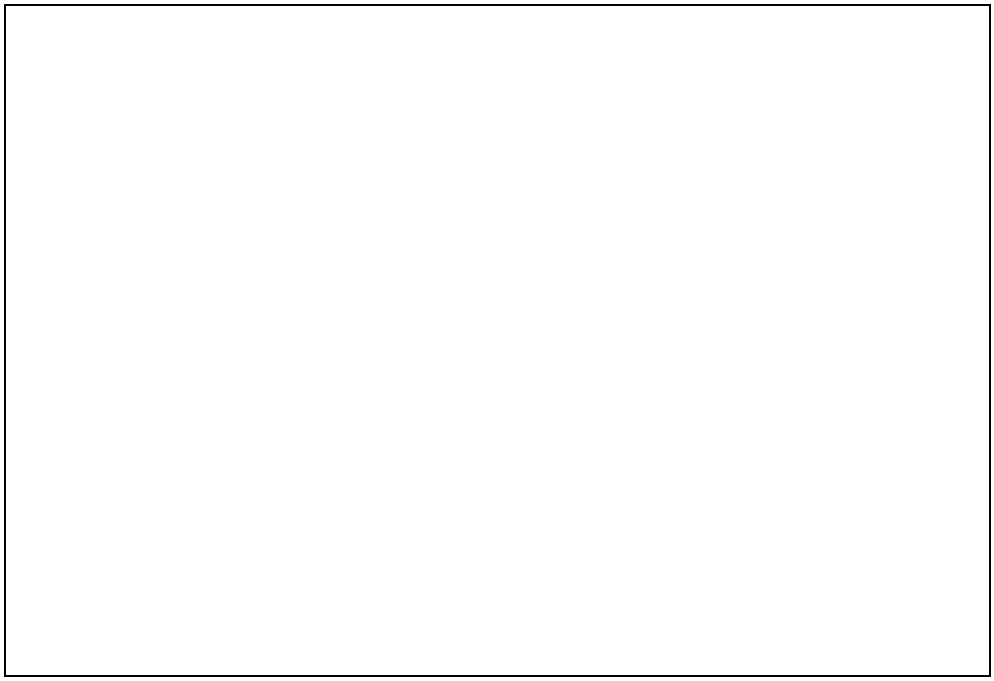


Fig. 5d

There is little freedom of choice about what the classes are, given that you know the rules and constraints and follow the idea that each class is a state machine controlled by one state variable (or none if it can be optimised away). Note that the behaviour analysis shows that most of the classes in this example have one-state lives, and so require no state-variable.

See the footnotes for some remarks on classes as state machines.

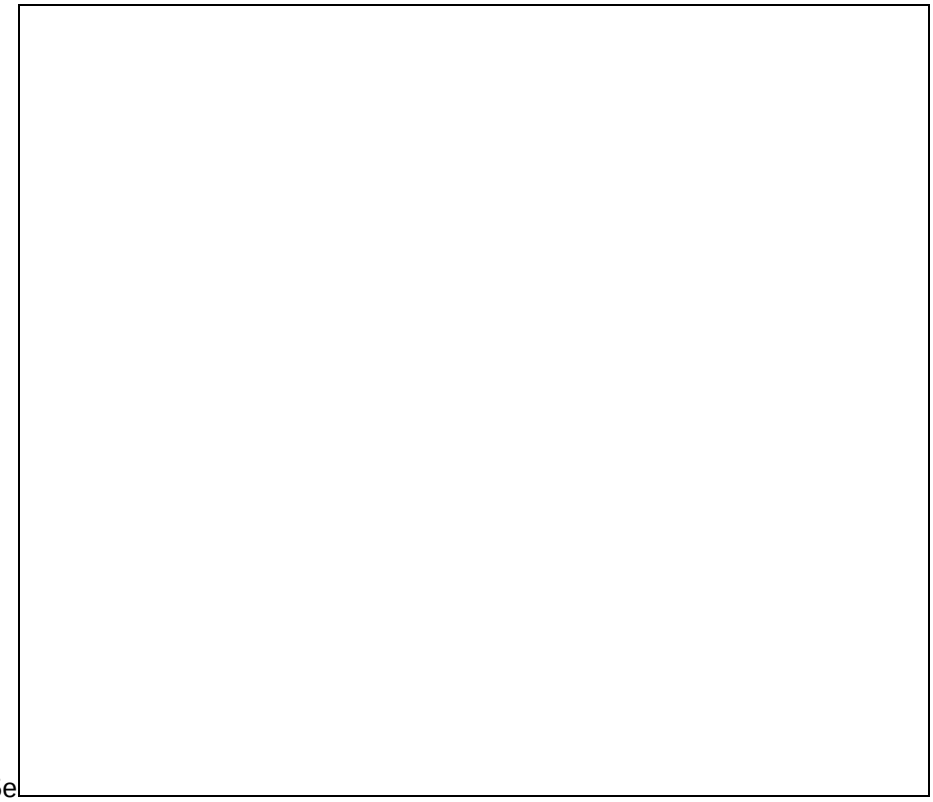
30.4.4 The data storage structure

Given a system that maintains persistent data, you must design the record types or tables into which the database will be divided. The physical database designer's view is that a class is a record type or database table, that is the unit of input/output accessed by programs.

Database designers usually transform a relational entity model first into a data storage structure (technology-independent) and then into a physical database structure (technology-dependent). The data storage structure records decisions about the physical database structure, with one box for each table or record type.

There is much freedom of choice here. You might map logical to physical by designing a separate table for each relation, or each subclass in a class hierarchy, or each state machine. In complex systems you cannot do all of these at once.

Fig. 5e avoids the structure clash between different logical views by rolling up the logical subclasses and parallel aspects of Vehicle into one table.

Fig. 5e 

30.4.5 Object-oriented purist's entity model

There is one more logical view, considered in the next section. The inheritance-oriented view is that a class is something uniquely identifiable by a type or subclass. In our example, a Vehicle may be either a Car or a Truck. Should we show the subclasses as distinct classes in an entity model?

So by way of summary, you can distinguish four or five different ways to define what a class is, and so draw different data structures:

Kind of entity model	The entity or class is
The informal entity model	whatever data group you like
The relational entity model	a normalised relation
The entity state machine model	a state machine
The data storage structure	a database table or record
Object-oriented purist's entity model	a type in a type hierarchy

The main point of this chapter is that all five of these views are reasonable and useful. What we need is an architecture that enables us to consider each view during analysis, and maintain clashing views separately in the modular construction of our software.

31. Subclasses and parallel aspects

This and the following sections detail variations in ways to draw entity models where a class might be divided into subclasses or into parallel aspects.

- For subclasses the options are called: aggregation, pseudo-inheritance and delegation
- For parallel aspects, the options are called: aggregation, partition and delegation.

31.1 Subclasses

Given a class hierarchy containing a superclass (say Vehicle) with two subclasses (say Car and Truck), people have proposed three different ways to specify the subclasses: aggregation, pseudo-inheritance and delegation.

31.1.1 Aggregation of subclasses into one class

This means specifying the entire class hierarchy as one class. You may draw an exclusion arc across relationships specific to different subclasses.

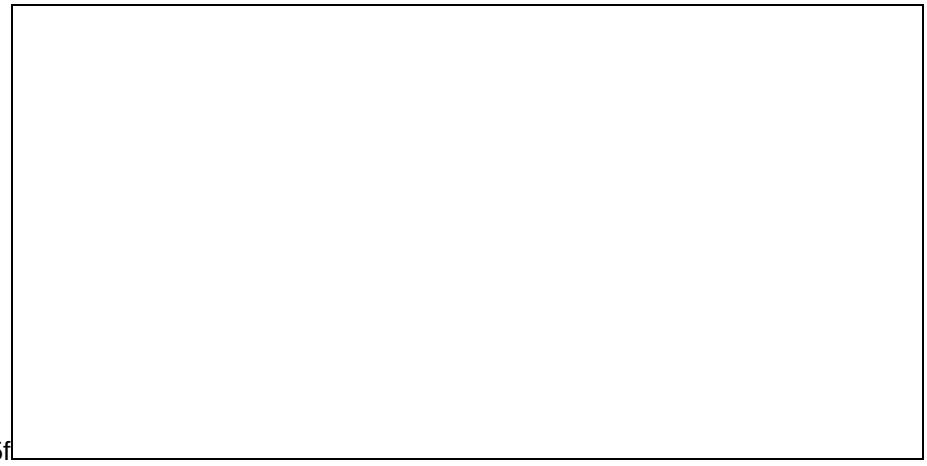


Fig. 5f

31.1.2 Pseudo-inheritance of super class into subclasses

This means specifying only the subclasses as classes, copying the properties of the superclass into each of them.

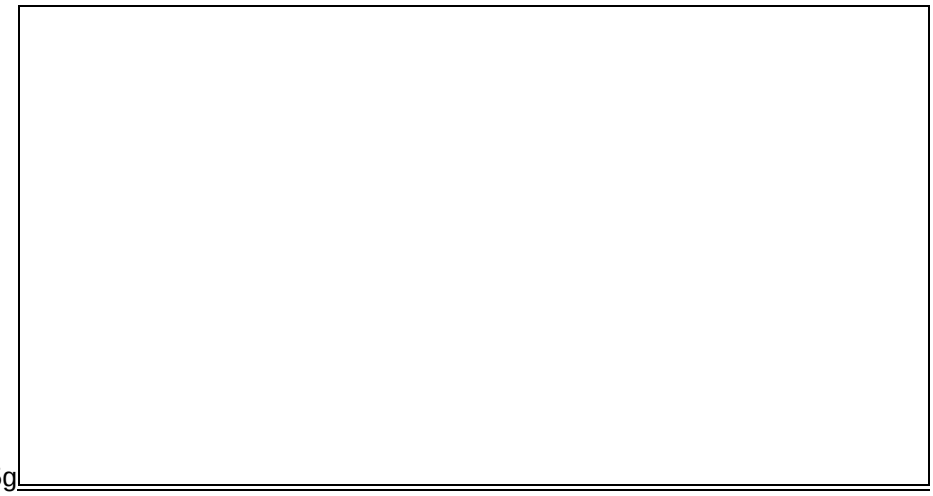


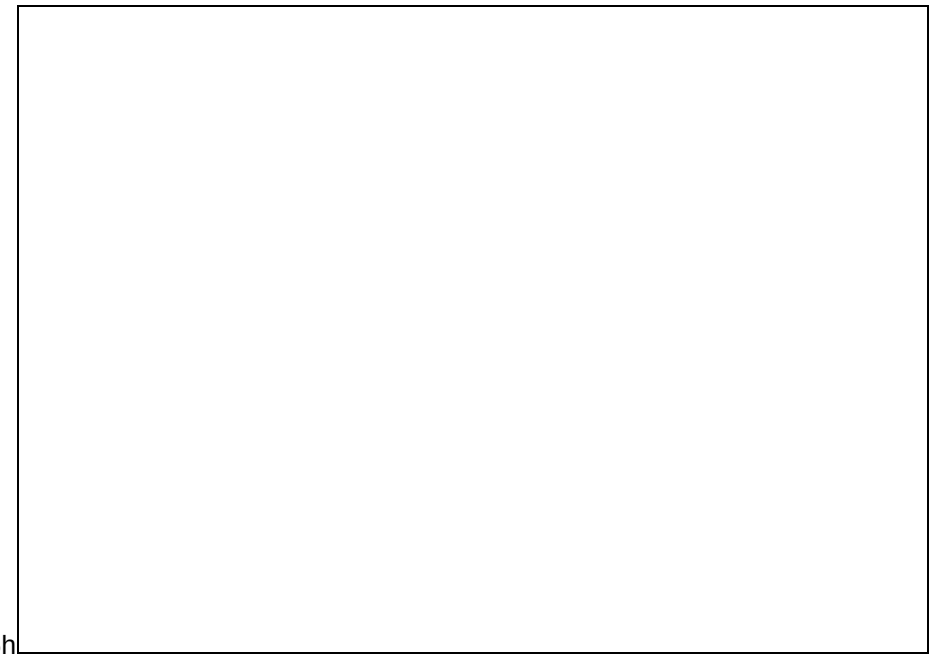
Fig. 5g

The pseudo-inheritance approach means repeating data specification (above, the relationships to Owner and Model) in an anti-reuse, anti-maintenance, kind of way. Nevertheless, at the expense of some duplication, a handy rule-of-thumb is to do this if the users employ a different range of identifiers for each subclass.

Since we have assumed that all kinds of Vehicle are identified by the same primary key, Reg Num, we would not divide in this example. By the way, if we did divide, then any process given only Reg Num as input data must perform a preliminary enquiry to find out which subclass to access.

31.1.3 Delegation of subclasses as detail classes of a master class

This means specifying the superclass and each subclass as distinct classes, connected by ‘is a’ relationships. In this entity model the boxes are things that are uniquely identifiable one from another by a combination of key and ‘type’.

Fig. 5h 

Representing the subclasses of a class

Given four kinds of entity model and three ways to specify a class hierarchy as classes (aggregation, pseudo-inheritance and delegation), which way suits which kind?

Drawing class hierarchies in an informal entity model

Since the informal entity model is only an informal picture, it doesn't matter which way you choose to draw it. However, delegation is common. People like to draw the subclasses as distinct boxes in an informal entity model because it helps them analyse the problem domain, even if they later decide to aggregate the subclasses, or use pseudo-inheritance.

Drawing class hierarchies in a relational entity model

Rule-of-thumb: if the users recognise objects of two subclasses by the same primary key use aggregation; if not use pseudo-inheritance. We aggregate in our example, because all kinds of Vehicle are identified by the same primary key, Reg Num.

Drawing class hierarchies in an entity state machine model

The structure clash between subclasses and parallel aspects needs further exploration. The difficulty is outlined here.

Before you can draw a box in an entity model for each state machine, you have to understand the right way to build state machines. You must somehow document the constraint that the state machines of the subclasses are mutually exclusive. The way to do this is by drawing a high-level selection between options, where each option represents a subclass.

Aggregation is wrong. It means drawing only one state machine for the class hierarchy, including the behaviour of all subclasses in it, duplicating the superclass behaviour under each subclass option.

Pseudo-inheritance is wrong. It means drawing state machines only for the subclasses, duplicating the behaviour of the superclass in each. This means there is no specification in the state machines of the mutual exclusion between subclasses. You must prefix some events by an enquiry to work out which class they affect. Given an event (Scrap Vehicle) that carries only the primary key of the object (not its subclass) there is no way of knowing to which class the event must be directed. The corollary is you cannot rely on generating all the event preconditions from the state machine documentation.

Drawing class hierarchies in a data storage structure

Aggregation of subclasses into one relation is normal. The subclass's data attributes are contenders for the same data storage space. Aggregation also sits happily with the view taken in behaviour analysis.

31.2 Parallel aspects

A parallel aspect of a class (or as some object-oriented authors say 'non-disjoint subclass') is an independent group of attributes and relationships, whose behaviour is governed by a single state variable.

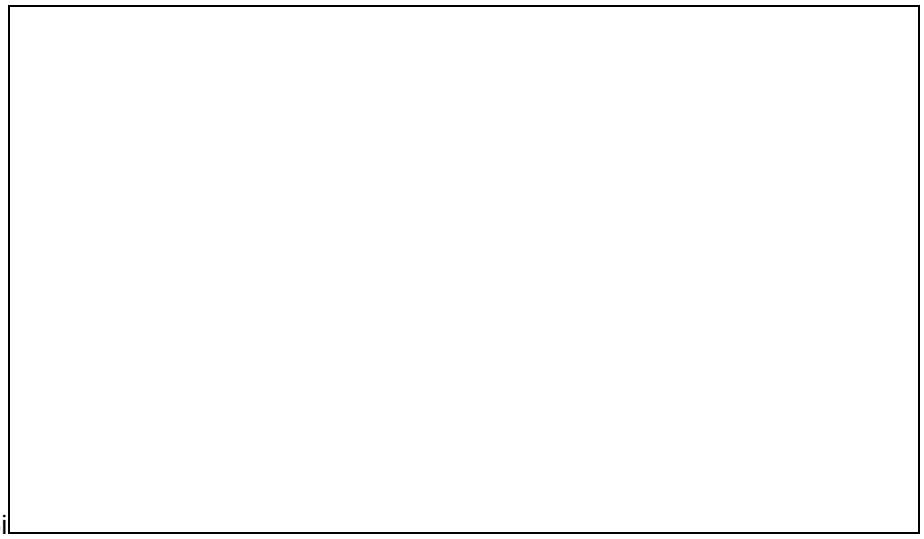
Behaviour analysis (and structure clash resolution) tends to lead you to divide an aggregate relation into its component parallel aspects. You may go as far as to decompose the behaviour of a relation into one parallel state machine for each attribute and relationship. However, a group of attributes and relationships that share the same state-transitions are normally lumped together in one state machine, making it a very low-level aggregate class.

31.2.1 Aggregation of parallel aspects into one class

Since aggregation rolls up all parallel aspects into one class, the picture of the case study here would be the same as that for aggregation of a class hierarchy shown earlier.

31.2.2 Partition of parallel aspects into several classes

The picture here is different from partition of a class hierarchy shown earlier. There is a structure clash between subclasses and parallel aspects. Dividing parallel aspects leads to this entity model.

Fig. 5i 

31.2.3 Delegation of parallel aspects as detail classes of a master class

There is only a subtle difference between delegation and partition of parallel aspects.

In delegation, one of the parallel aspects is appointed as the 'basic aspect'. E.g. the basic aspects in the example are Owner-basic and Vehicle-general. The 'basic' aspect can be thought as being at a higher level and owning all the others. This means we can connect all the other aspects to the basic aspect by association relationships.

31.2.4 Representing the parallel aspects of a class

Given four kinds of entity model (informal, relational, application and data storage) and three ways to specify parallel aspects as classes (aggregation, partition and delegation), which way suits which kind?

Drawing parallel aspects in an informal entity model

Aggregation is normal. People rarely draw one box for each parallel aspect in drawing an informal entity model. As far as code specification is concerned, it doesn't matter which way you choose to draw the informal entity model.

Drawing parallel aspects in a relational entity model

Aggregation is normal. Relational theory doesn't really account for parallel aspects, but one might say it assumes aggregation of parallel aspects into one relation.

Drawing parallel aspects in an entity state machine model

Partition seems the natural thing, since each box is supported by a distinct state machine. It enables The entity state machine model to be used as a map or graphical menu for navigating to the state machines.

Strictly, delegation is the right approach. One of the parallel aspects must be appointed as the 'basic' aspect, which can be thought as being at a higher level and owning all the others. The state machine of this basic aspect will normally be responsible for all different varieties of birth and death events, but may pass these on in the form of a 'superevent' to the other parallel aspects.

After aggregating the class hierarchy and delegating the parallel aspects in our example, the result is an entity state machine model that matches the event models. The boxes in it are state machine. The boxes are the classes for which state machines are constructed and that appear as distinct components in event models.

Drawing parallel aspects in a data storage structure

Partition is simplest. This sits happily with the view taken in behaviour analysis and facilitates the distribution of parallel aspects to different physical locations.

Aggregation of parallel aspects into one relation should reduce access times, but requires a little extra work in the data abstraction layer (components that retrieve logical application objects from the Data services layer, and restore them).

31.3 The need for a richer analysis methodology

What is an entity or a class? What does a box in an entity model represent? Different answers lead to different entity models (ignoring differences between notations). The picture is further complicated by different ways (aggregation, partition and delegation) to specify the 'subclasses' and 'parallel aspects' of classes.

Given there are many ways to draw an entity model, a methodology should help us to decide which way suits which purpose. Which way of drawing an entity model best suits database design? Which way suits object-oriented modelling? We need a methodology which disentangles the various questions involved and provide some answers.

The methodology implied by clashing entity models can be organised into a handful of major activities, where a fair amount of parallel activity is possible. Fig. 5j gives an idea of what we mean.

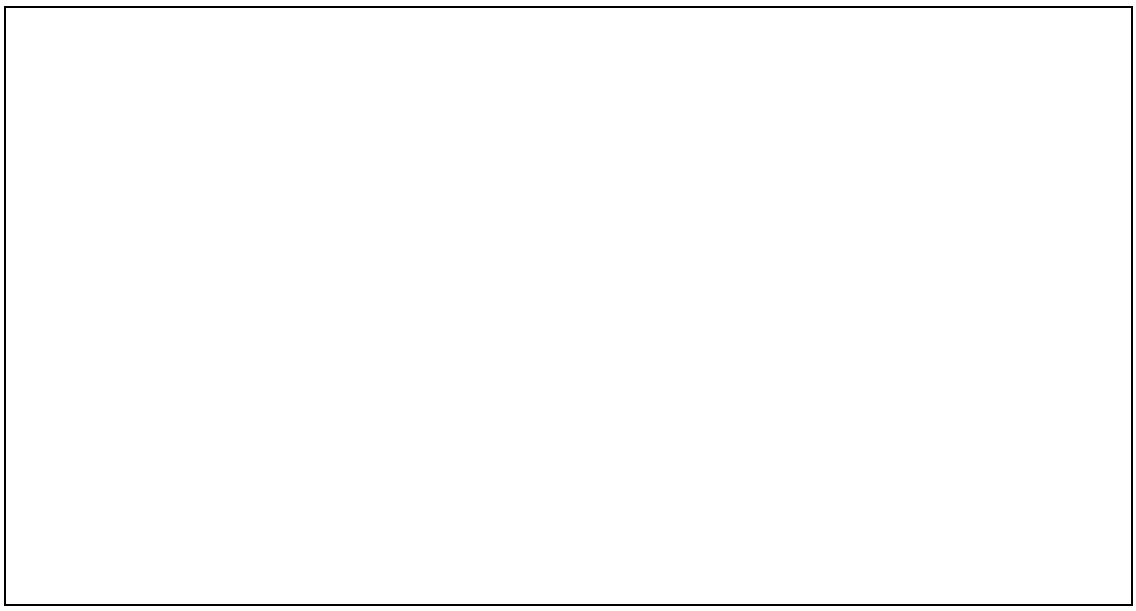


Fig. 5j

There is a progression from informal to formal, only two models appear in the implemented code - The entity state machine model and the data storage structure. In simple systems these will be the same. In very simple systems they will both be the same as the relational entity model. See 'The OPEN book' for further discussion of the methodology in Fig. 5j.

Coordinating software architecture

In non-trivial enterprise applications, rather than select one or other logical view as the basis of encapsulation, we want to have it all ways. Current object-oriented ideas are inadequate; we need a richer theory of system specification. We need a software architecture that results in separates components handling each logical view, and separate components addressing the physical concern of designing efficient database tables. Our 3-tier software architecture is designed for this purpose.

You can specify the data attributes of any entity model box using an underlying data dictionary. You can specify one-to-one correspondence between models by names. A CASE tool can help as discussed below. The earlier sections of this chapter show how you can make things easier by taking design decisions that align different versions of the entity model.

31.4 Technology implications

Using object-oriented ideas to specify the Business services layer does not mean you have to use object-oriented software tools implement the application. We do need some kind of:

- GUI management software to implement the UI layer
- Common programming language to implement the business services layer
- Database management system to implement the data services layer

We need a database management system that supports the notion of a commit unit, handles

the back-out of updates to objects within a commit unit, and helps with locking and logging of transactions. It must have an efficient way of reading and writing the instance variables of not only of single object but of aggregates of related objects (say, all the details of a master).

CASE tool implications

An upper CASE tool could allow us to draw four models:

Kind of data structure	Each box connected to a
informal entity model	-
relational entity model	data group
entity state machine model	state machine and data group
data storage structure	physical database table.

Copy and paste functions will enable us to copy all or Component of one model into another. Ideally we'd like to draw only one 'master' relational model, then draw parallel versions only of the Components that differ in the 'subordinate' entity state machine model (where parallel aspects are separated) or data storage structure (say, where subclasses might be divided). Working out a pleasing way to cross-refer between the master and subordinate diagrams would be a challenge.

Copy and paste

The tool should allow you to draw multiple versions of a data structure. It should help you to duplicate whole entity models and copy and paste partial entity models between them.

Cross-reference by name

The tool should connect any entity model box to related documentation items by recognising the names (or possibly synonyms) of corresponding classes. The name of a box in an entity state machine model must match the name of a state machine. The name of a box in the data storage structure must match the name of a database table in the lower CASE tool. The name of an attribute must match the name of a domain in the dictionary.

Domain dictionary

The tool should help you to specify the data group behind any box in any entity model as a list of domains drawn from an underlying domain dictionary. This central repository should be independent of the internal data storage structure or the external user interface. Most current implementation tools tie variable definition too closely to one or the other.

Cross-layer class specification

A single class may have properties in each layer, a data storage format, a presentation or display format, and some application rules. Suppose the user asks for a class from one system to be added to another. We'd like to reuse the class without defining new data storage and presentation formats.

We need a way of defining an entity class and attaching to it the baggage of its data storage and presentation views, so that these can travel with the entity class. We can envisage this working at the atomic or data item level of system specification. How to handle larger classes is an open issue.

31.5 Footnotes

On domains in enterprise applications

James Odell (1994) says: ‘Existing entity modelling techniques have deficiencies for object-oriented analysis. To assist the object-oriented designer, the object-oriented analyst must specify *all* object types and associations clearly.’ By ‘all’, he means to nag the analysts into adding the basic data types and domains as classes into their entity models. But they don’t want to do this!

Analysts don’t usually have too worry much about domains. True, specifying the individual variables in a system specification is important, often difficult and always time-consuming. But specifying the *domain* of each variable is not that difficult. It can be postponed until near the end of design.

Every database builder knows they must define the domain of each variable. Most are happy to leave it until the analyst has completed something like a relational entity model. You need to get an overview of the system structure before defining its details.

When it comes to specifying the domain classes in enterprise application, a simple two-level structure, generic and applicaion-specific, is usually enough.

Generic domain classes

At the bottom level of specification you may use a few generic domain classes, such as text, number and date formats. These are only a minor concern; you probably need only the half a dozen or so data types provided by your chosen implementation technology.

Application-specific domains

At the next higher level above generic domain classes, there may be tens or hundreds of application-specific domains, such as ‘Phone-Num’ and ‘Country’. There are two ways to specify these system specific domains, at the bottom and at the top of the specification.

You may specify the domain in a dictionary, to be reused in defining the attributes of classes. You might define a dictionary entry Phone-Num as being of the generic domain class Number and always beginning with 0, then specify the classes Supplier and Customers as having attributes called Supplier-Telephone and Customer-Telephone, both with the properties of the application-specific domain called Phone Num.

Note that naming conventions must be agreed. If you name two or more attributes directly after their domain, then wherever the context is ambiguous, say in an input message, you will need to declare the context somehow, say Supplier/Phone-Num or perhaps Phone-Num (of Customer). For this reason, people tend to compose an attribute name by combining the class name and domain name.

Or else you can turn the specification on its head by declaring the domain as a high-level master class (say Country) in the entity model, and then specify different classes owning this attribute (say Customer and Supplier) as detail instances of the master class. Thus, you can invert *any* attribute variable, or rather its domain, to become a class whose key is one valid value of the domain.

An age-old question of database design is: should we do this? Should we show the application-specific domains as classes in an entity model? We addressed this question in ‘Introduction to rules and patterns’.

On classes as state machines

Jackson (1975) showed how to decompose a system into components by resolving 'structure clashes'. The resulting system is a set of co-operating state machines, where each process has an input data structure, a state vector and a state variable. Let us bring Jackson up to date with object-oriented by adding object-oriented terms in square brackets to Jackson's original remarks.

The input data structure of a class

Jackson considered each object as consuming the stream of events that update it. In object-oriented terms, this stream of events is a stream of method invocations. Each event invokes a method of the object. We name each method after the event that invokes it, or after a superevent where more than one event can trigger the method.

The state vector of a class

Jackson grouped the private variables of a class in its state-vector. He said: 'the contents of the state-vector are private to the [object]: they are truly "own variables" in the sense that no other [object] should be able to inspect or change them, or to take cognizance of their formats and values'. In object-oriented terms, this is encapsulation.

To resolve an 'interleaving clash', you have to separate the 'multi-threaded' process from its the state vector. You keep one copy of the process (one for the class), but many copies of the state-vector (one for each object). Thus, Jackson promoted the idea that a database is nothing more or less than a place to hold the state-vectors of concurrent objects. In object-oriented terms, this is resolving a concurrency problem.

The state variable of a class

Jackson said: 'the [object] has only one linear text and one location counter; the current place in the text must correspond to the current place in the data structure [stream of method invocations] being processed'.

In object-oriented terms, the importance of the object's location counter or state variable is that you can use it in evaluating rules. If an event finds an object in the wrong state, then that event must fail. Following our application modelling techniques, rules are coded in the form of event preconditions.

Object-based versus object-oriented

Jackson's view of object classes is sometimes called 'object-based' rather than object-oriented, meaning that it does not incorporate the idea of class hierarchies or inheritance. Berrisford and Burrows [1994] showed this to be untrue.

Class hierarchies and aggregates

Fig. 5k shows School as an aggregate in which the basic aspect of the class has parallel aspects that are mutually exclusive. The relationships from the basic aspect to the parallel aspects are crossed by an exclusion arc. The entity model then *looks* like a class hierarchy, but the lines between boxes are 'association' relationships rather than 'is a' relationships.

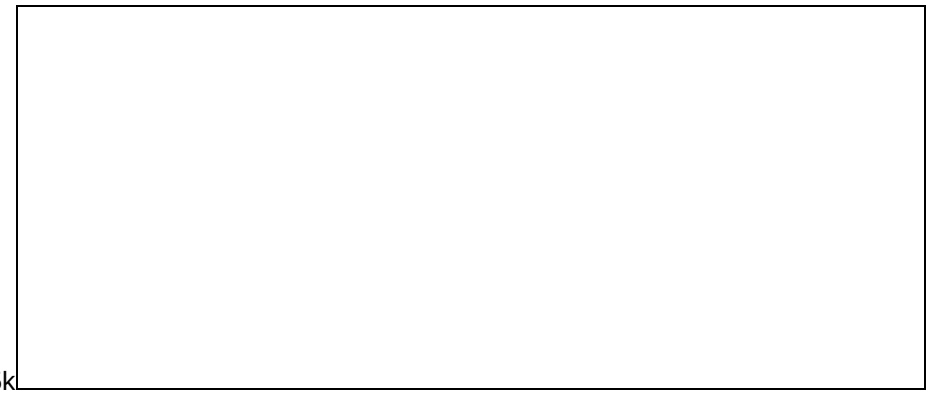


Fig. 5k

Head Teacher and Principal Governor are disjoint ‘aspects’ rather than disjoint ‘subclasses’. Is this merely sophistry? Does it makes a difference to the state machines?

Berrisford and Burrows (1994) showed that to express the disjointness of subclasses in state machines, you have to draw the state machine of each subclass as an option within a state machine of the superclass. So the life of a subclass may be completely rolled up into the life of its superclass.

At first sight (we haven’t explored enough examples yet), it seems like the same principle applies to an aggregate of disjoint aspects as to a class hierarchy of disjoint subclasses. If so, then we would say the distinction is sophistry.

Glossary

Aggregation: grouping the properties of subclasses or aspects into one high-level superclass or aggregate class.

Aspect: an independent role of a class, a group of attributes whose updating is constrained by one state variable, a class whose behaviour is representable as a single finite-state machine.

Class: a set of object instances that share the same properties.

Class hierarchy: a structure dividing a superclass into subclasses.

Delegation *: dividing the properties of a class between a high-level class and low-level classes connected to it.

Event: an atomic transaction, a minimum unit of consistent change, transient but leaving a mark on persistent objects.

Event model: a specification that shows how one or more objects are affected by a single event, and the constraints that must be tested.

Object: something that persists and must be remembered.

Partition *: dividing the properties of one class between smaller roles or aspects.

Pseudo-inheritance: copying the properties of a superclass into its subclasses.

* The difference between Partition of parallel aspects and Delegation of parallel aspects is not obvious. Both mean separating a class into parallel aspects. But Delegation implies one of the parallel aspects

is appointed as the ‘basic aspect’ that is at a higher level and owns all the others. You can think of the ‘basic aspect’ as the creator and destroyer of the object identity, simultaneously creating and destroying all related aspects.

32. Design issues

This chapter discusses design issues and tradeoffs. It shows how the separating the application and Data services layers of 3-tier architecture can help you to hide data replication and aggregation from the Business services layer of code, and minimise data migration difficulties.

32.1 Data replication and derivation

Redundant data makes one object dependent on another, so if you update one object, you are obliged to update another at the same time. But redundant data is not necessarily a bad thing. You have to consider a design tradeoff, and the kind of redundancy that is involved.

Tradeoff: enquiry process v. update process

The speed of a process is largely determined by the number of discrete objects it accesses. Reducing the objects accessed on update may increase the objects accessed on enquiry, and vice-versa, so you cannot optimise both updates and enquiries.

Similarly, the *simplicity* of a process is largely determined by the number of classes it accesses. Reducing the classes accessed on update may increase the classes accessed on enquiry, and vice-versa, so you cannot simplify both updates and enquiries.

An aim of relational data analysis is to simplify programming, to prevent programmers from writing unnecessary code. It achieves this by reducing data replication. E.g. you would normalise the Sale class on the left below, specify Stock as a separate class and assign Stock Description as an attribute of Stock rather than Sale.

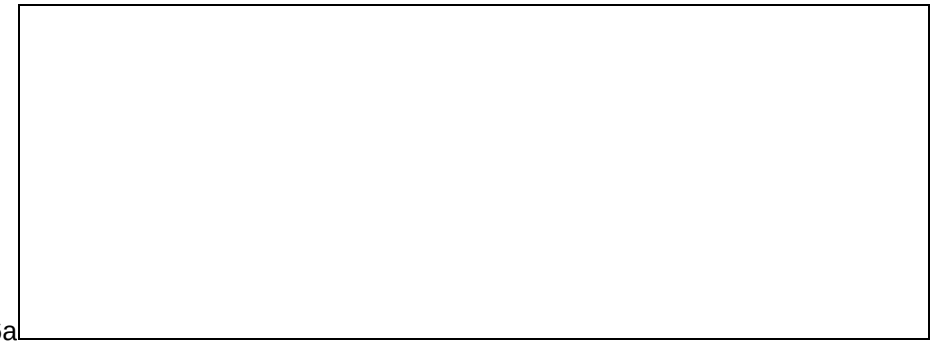


Fig. 6a

Thus, you prevent programmers from having to locate and update all the Sales of a Stock, in order to update a Stock Description. This has the added benefit of reducing the danger of inconsistent Stock Descriptions being stored, through the update process not being completed properly (whether this is a failure of the programmer or the technology).

People sometimes teach relational data analysis as though its aim is to remove all redundant data. First, this is a means not an end. Second, there are two kinds of redundant data - replicated data and derived data - and they have different implications.

Replicated data

Replicated data occurs where one piece of information is repeated. This is not necessarily a bad thing. You may choose to replicate data to speed up or simplify enquiry processes. Typically, you might repeat an attribute of a master object in every one of its detail objects.

E.g. you might store Stock Description as an attribute of Sale, replicated in all Sales of a Stock. This will speed up any enquiry on a Sale that would otherwise have to access the Stock object for the Stock description. And if the Stock object is stored in a different database from the Sale, it will increase the cohesiveness and robustness of local processing.

If you do replicate data, it is wise to maintain the original data as well as its copies. So you should maintain Stock as well as Sale. We'll come back to this under 'Distribution'.

Derived data

Derived data occurs where several pieces of information are summarised in one place as the result of a calculation or procedure. The usual example is a total stored in a master object of detail objects. E.g. you might store a summary total of Sales in a Stock object, to save adding up this total on each enquiry.

Another kind of derived data is a derivable sorting class. E.g. 'Customer Interest in Stock' is a derivable sorting class that clusters all the Sales for a given combination of Customer and Stock.

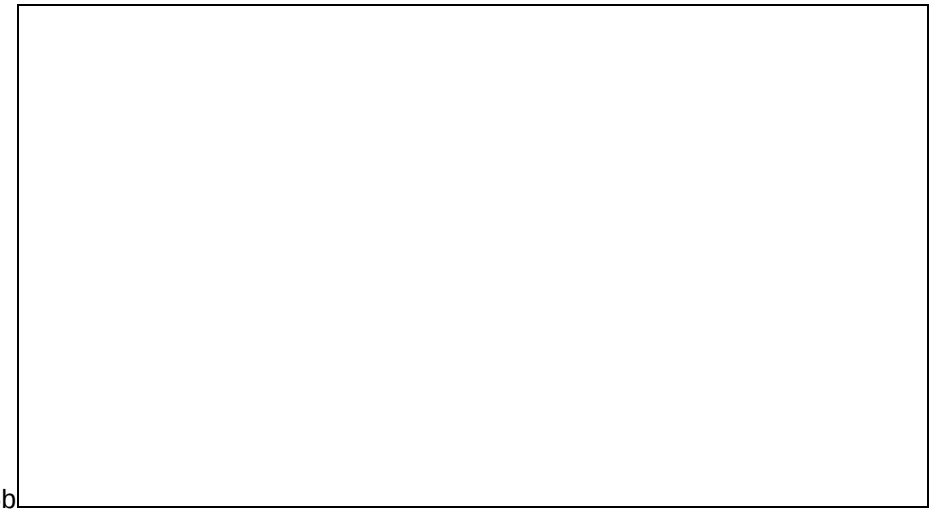


Fig. 6b

Removing derived data can frustrate the aim to simplify programming. Suppose you omit the sorting class from the data structure. Programmers will have to sort Sales by Customer within Stock, or Stock within Customer, every time they want to display them in a structured list. In effect, they manufacture a 'soft' instance of the sorting class every time they need one.

In general, analysts can make extra work for programmers. Missing derivable classes and relationships from the data structure can make programming unnecessarily complex. Programmers end up defining the missing classes and relationships in program code. And they may have to do this lots of times, in many different programs.

Given the tradeoff between defining a 'hard class' in the structure of the persistent data, and defining a 'soft class' in one or more transient processes that operate on the data structure, the balance lies in favour of the former. As a rule:

Specify classes and relationships in the data structure, rather than leave them to be constructed by programs.

Benefits: simpler enquiry programming and easier program maintenance. True, whenever a class is amended you will have to amend all the programs which refer to that the class, but this is the case whether the class is hard or soft. And there will simply be less program code to maintain if the class is a hard one.

Costs: some extra update processing, extra data structure maintenance and data migration costs. If you map every entity class onto a database table, then you will have to 'migrate' persistent data from one structure to another whenever a class is amended.

Benefits without costs?

How to get the benefit of an application-specific entity model that makes application programming easy, while at the same time using a data storage structure that speeds up enquiry processes, increases the robustness of distributed operations, and facilitates maintenance without data migration? The 3-tier architecture opens up the interesting possibility that you might define different data structures for:

• Business services layer - entity state machine model designed to simplify processing

• Data services layer - data storage structure designed for performance and flexibility.

Most database designers reproduce the 'logical' entity state machine model as closely as possible in the data storage structure. This is how most systems are built. But you might take a very different approach in designing a large enterprise application. You can write application programs to operate on The entity state machine model , while storing instance data in a differently-structured data storage structure.

Replicated data belongs in the Data services layer

We propose that replicated data belongs in the Data services layer, not in the Business services layer

The idea is that you can specify and code the Business services layer as though no data is replicated, hiding all replication in the Data services layer.

E.g. the application program that updates a Stock Description will assume it is stored only in a Stock object; it will call the data abstraction layer; this will find all the places where the Stock description has been replicated and update all of them. So the application program is entirely unaware of how far data has been replicated. The data abstraction layer handles the extra complexity. You may even be able to a buy a distributed database management system that does the job of the data abstraction layer for you.

Derived data belongs in the Business services layer

Perverse though it may seem, we propose that derived data belongs in the Business services layer, not in the Data services layer.

It is nonsensical to hide derived data in the Data services layer only. It would be foolish to code an enquiry in the Business services layer to report the total Sales of a Stock by adding up the total, if the total has already been calculated and stored in the database. Likewise, it would be foolish to code complex enquiry processes in the Business services layer as though a sorting class does not exist, if it does exist in the Data services layer.

Doing it the other way around is far more reasonable. You can specify and code simple enquiry processes in the Business services layer as though a derivable total or sorting object has been stored. You may then choose to store the derived object in the Data services layer,

or else the data abstraction layer can derive it and present it to the Business services layer whenever it is required.

Conclusions: Don't store redundant data until you have established a clear business case in terms of speeding up enquiries or increasing the robustness of local operation. Specify 'replicated data' in the Data services layer of code. Specify 'derived data' in the Business services layer of code. Yes, this does mean the conceptual model of the Business services layer is influenced by physical design considerations, but the alternative is ludicrous.

32.2 Data distribution

A single central database is the simplest option from the design point of view. The motivation for distributing subsets of a database around the nodes of a network is to enhance the performance or robustness of local processing at a node. This may involve replicating data at different locations.

If you define different data structures for the Business services layer and Data services layer, then you can hide all data distribution decisions and complications in the Data services layer. You may the annotate the data storage structure with distribution details, leaving The entity state machine model untouched.

When it comes to distributing objects, the classes in the data storage structure might be divided into three kinds.

Objects that sit naturally at location

Locations where a business wants to store data often appear in the model as classes (department, warehouse, local office, or whatever). The natural scheme is store an object of a such a class at its real-world business location. Some details fall naturally under these locations.

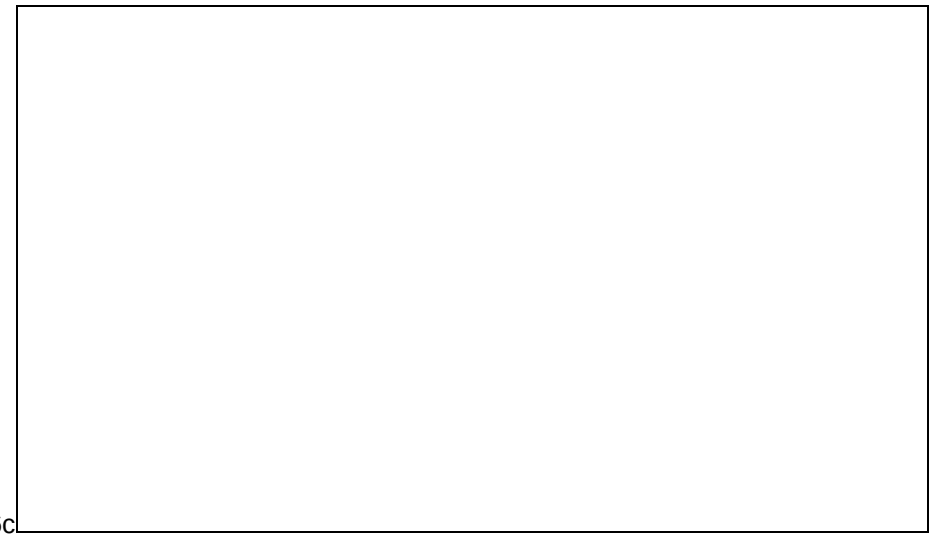
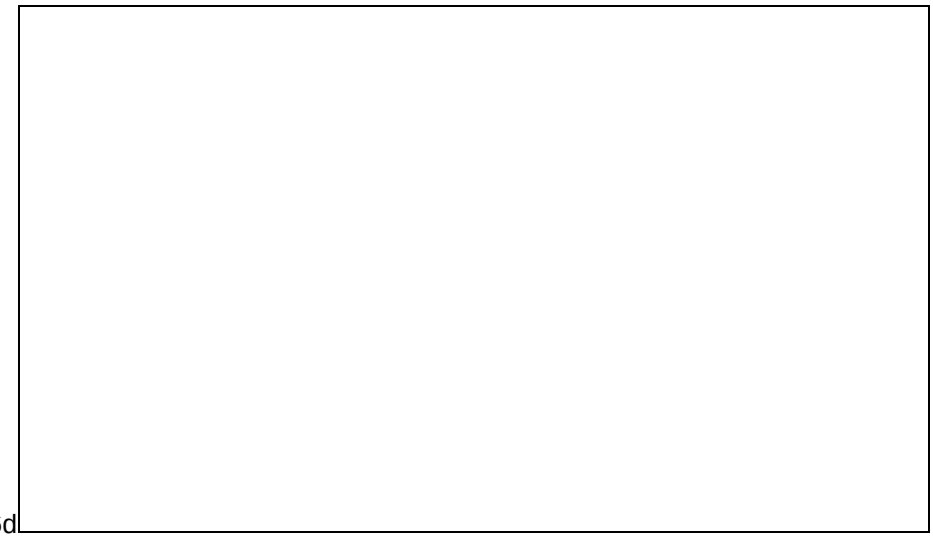


Fig. 6c

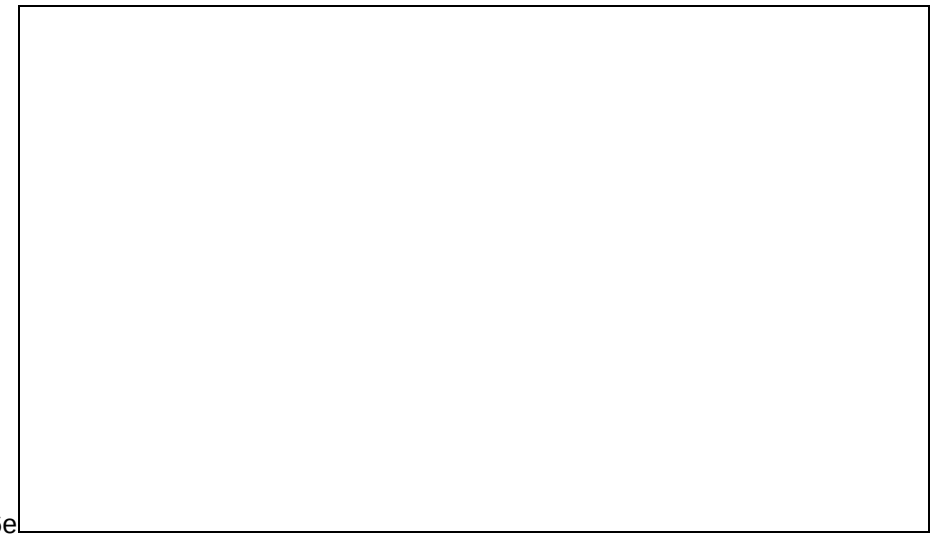
Not every object is naturally related to only one location. A customers may be the recipient of Sales from several Warehouses. You might begin by assuming that all multi-location objects objects are stored at a central server location.

Detail objects that link objects in different locations

You might choose to store a Sale at the location of either Customer or Stock. The notation in Fig. 6d suggests a Sale is stored with its Stock.

Fig. 6d 

Or you might choose to store Sales in a distinct storage location, separate from both Customer or Stock. Either way, distributed locations are connected along a one-to-many relationship. Managing a one-to-many association between distributed objects can be difficult. So you might instead choose to replicate a Sale in both locations, and connect the two Sales together.

Fig. 6e 

This has the advantage of connecting locations along a one-to-one relationship, which is simpler to manage. Of course, if there are further detail classes connected to a Sale, you now have to decide where they are stored, and perhaps duplicate them as well.

Master objects that are used in several business locations

Some master objects (like ‘Currency Conversion Rate’ or ‘Customer’ in our example) can appear at several business locations. You might choose to store these objects only once, in a

central server or head office storage location. The problem is that local processing may be too slow, or if the network goes down, people cannot carry on working on their local office database. A way around this is to unnormalise data and copy the master object into several locations, so user have all the information they need close at hand.

You might repeat the Customer name in every one of their Sales records. Or more openly, you might copy each Customer object in all business locations.

Fig. 6f

You should not eliminate the original master object. One object (in a master location or a distinct server location) has to keep track of all the places where the object has been copied, for the purpose of broadcasting updates.

So in short, distribution means you may have to:

- select one business location for objects that naturally relate to more than one business location
- define distinct storage locations other than natural business locations
- divide one class into two parallel aspects connected by a one-to-one relationship
- divide one object into one master object owning many copies.

Tradeoff: robustness v. inconsistency

Where a single database is partitioned and stored at several locations, the issue of robustness arises. If the network fails, you want to carry on working at one database location while not connected to the others.

To increase robustness, you will tend to replicate data at different locations. But this means that there is the danger of data in different locations getting out of step, whether due to sloppy design by or failure of the network technology. What if somebody updates, or worse deletes, a Customer object on one of the databases while the network is down? The various databases will get out of step.

Getting the databases back in step can take a great deal of effort. It is not just a question of running automatic update programs. While the network is down, you might accept Orders at a Warehouse for a Customer that has been deleted or black-listed at head office.

When you find out later that the Customer has been black-listed: Should you now reject these Orders? Or should you find some other Customer to take them? These are questions that the business analyst must address rather than the database designer.

32.3 Data migration

Data replication and data distribution are two good reasons to design a data structure for the Data services layer that is different from the data structure of the Business services layer. Data migration may be another reason.

Programs are transient. Data is persistent. So changing a data storage structure involves an extra step, called ‘data migration’, that changing a program does not. You have to reorganise already-stored data, shifting it from one version of the data storage structure to the next.

The more you specify application-specific classes and relationships in the data storage structure, the greater the data migration cost whenever these classes or relationships change.

This is not necessarily a bad thing. Remember, the rule to specify classes and relationships in the data structure rather than leave them to be constructed by programs. What the database designer misses out, the programmers will have to put in, tenfold. And if data migration is needed because you are correcting a poor data storage structure, inserting classes or relationships you overlooked, then you have only yourself to blame.

Conclusion: expect data migration and include it your plans.

Nevertheless, there are some very large databases where data migration is just too expensive. Is there an alternative design for maintenance strategy that will reduce or eliminate data migration?

Avoiding the cost of data migration

Can you have it both ways? Can you have both the specificity of The entity state machine model , and the flexibility of a data storage structure that does not require amendment when The entity state machine model is altered?

Again, yes you can. You can write application programs for the classes in The entity state machine model , and store instance data in different and more generic structure in the data storage structure.

Fig. 6g shows an extreme example. The structure on the right is generalised so far that no conceivable application amendment would require it to change.

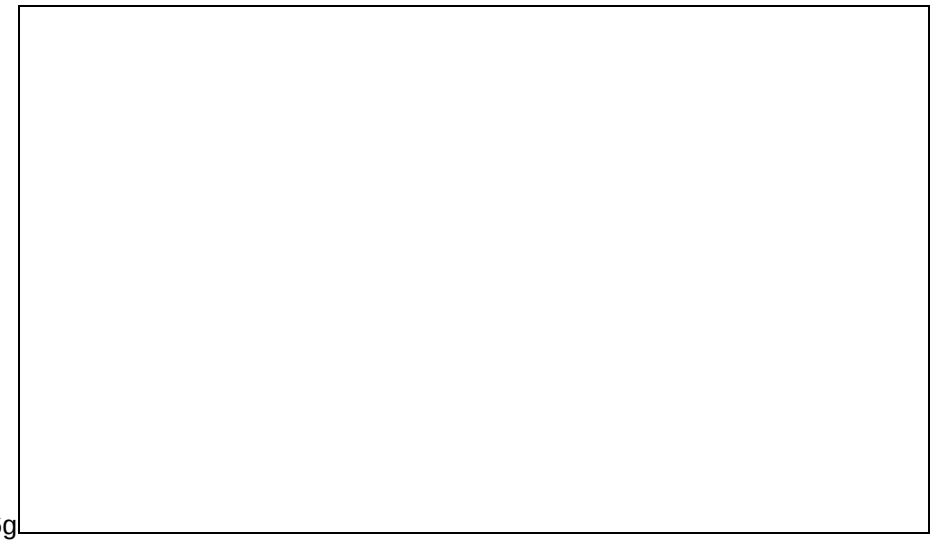


Fig. 6g

How does this work? You code the entity classes and relationships in the Business services layer. You code the data storage structure in the Data services layer. You design an data abstraction layer to translate between the entity classes and relationships and the data storage classes and relationships.

When your application program wants the instance data of a specific Customer, it does not read the data storage structure but calls the data abstraction layer. How the data abstraction layer assembles the Customer object from the data in the data storage structure is a matter only for the data abstraction layer.

When The entity state machine model is altered, you have to amend the application programs, you have to amend the data abstraction layer, but you do not have to restructure the data storage structure or carry out a data migration exercise.

Conclusion: where it is justified (by data migration or performance costs) introduce an data abstraction layer to separate The entity state machine model from the data storage structure (designed for flexibility and performance).

32.4 A few more tradeoffs

The art of system design is to find the best balance between conflicting objectives. Many authors have listed general objectives for system design. Some have suggested ways of measuring how far these objectives are achieved. Relatively few have focussed on the tradeoffs between objectives.

The optimum balance between conflicting objectives will differ from system to system. We have been making generalisations about tradeoffs in the kind of system we are most interested in - enterprise applications. Here are some more tradeoffs to finish with.

Efficiency: size v. speed

You might reduce the amount of code in a monolithic program by removing a repeated block of code into a reusable subroutine. But this will tend to slow the program down.

You might provide a faster alternative algorithm for a given process. For example, you might

design a faster text printing algorithm that produces only rough or draft quality print. But this will increase the amount of code in the system.

Object-oriented programmers often do provide alternative algorithms for a single process. The substitution of one algorithm by another is recognised by Gamma et al in the form of a design pattern called 'Template'. The substitution of one step in an algorithm by another is recognised by Gamma et al in the form of a design pattern called 'Strategy'.

Yet in the Business services layer of an enterprise application, you virtually never provide alternative algorithms for one process. In fact, it is not worth worrying about processing speed at all. The speed of an enterprise application is completely dominated by the time taken to store and retrieve data. Efficiency lies in the hands of the database designer.

In speeding up data access, a database designer will tend to increase the backing store needed to hold the database. The designer will allow more space for a data group to fit on the page of the database it is placed on, so it doesn't overflow that page. The designer will allow more space for storing relationships, space for extra pointers and extra indexes.

Conclusion: buy much more database space than you think you will need.

Database accessibility: crude locking v. concurrent usage

While it is running, a database update process has to lock the entities it is working on so that no other process can alter them. A crude locking mechanism will lock the whole database, or a large area of it. The ideal locking mechanism will lock only the objects actually updated by the process

If there are many concurrent users of the system, a crude locking mechanism can dramatically degrade the system's performance. To speed up the system, you will need a more sophisticated locking mechanism that works at a lower level of granularity.

Conclusion: refine the locking mechanism in proportion to the number of concurrent users.

Database enquiry speed: aggregation v. flexibility

To speed up a specific enquiry or display you may store all the data you want for that enquiry in one large object. The price you pay is inflexibility and disoptimisation from another enquiry perspective.

For example, if you store all of a Customer's Orders within the Customer object, then you can easily and swiftly assemble the list of Customer's Orders for display.

You might call this an aggregate entity state record, or an unnormalised object. Calling it a 'real-world' object is nonsense. An aggregate entity state record is no more a real-world object than a third normal form relation is a real-world object, it's just data storage that's optimised from one perspective, usually for output display.

Such optimisation makes the system less flexible, less suited to processing from another perspective. For example, you cannot so easily list all the Orders placed for a specific Stock Type.

(By the way, some of the things people say about how much better an object-oriented database is than a relational database are the same things network database designers have been doing for twenty years to optimise performance. To speed up access - store pointers to the detail objects along with the master object. To save space - roll up detail objects into one or other master object, making an aggregate entity state record. These are matters for the Data services layer, nothing to do with defining the Business services layer.)

Conclusion: don't unnormalise stored data into an aggregate entity state record until you have

established a clear business case in terms of enquiry speed, and define aggregate tables in the data storage structure rather than in the entity state machine model.

Cost of usage v. cost of design

Making the users work at the user interface easier takes more design effort. Conclusion: spend money on usability in proportion to the number of end-users who will benefit from your design efforts.

Breadth v. focus

Users want a system that does the job, no more, and operates efficiently. If you give users more than they ask for, you may end up obscuring the main functions behind features people never use, making the system harder to use, and slowing it down.

(Perhaps you discovered this from a user's perspective when you last upgraded your word processor to the latest version.)

Worse, features that are never used tend to fall into a state of disrepair and decay. Since nobody cares about them, you can be pretty sure that they won't work very well if somebody wants to use them in the future.

Conclusion: don't implement more features than you are asked to, but don't let this stop you thinking ahead and designing for maintenance.

Complexity: component size v. component interaction

Designing a large component or module takes a long time. A large component is harder to understand, test and maintain. Most people recommend you decompose a system into small self-contained components. Indeed, this is a mantra of object-orientation.

The trouble with replacing a large component by smaller ones is that they must talk to each other. There is more interaction between components than before. You have to concentrate more on the interfaces between components. Message-passing becomes more of a design issue. You replace one kind of complexity (proportional to component size), by another kind of complexity (proportional to component interactions).

(There is a more obscure difficulty with defining many small object-oriented components or classes. Where not all the effects of one event type appear in one class, you may have to add an extra 'gatekeeper' class to sit on the path of an event type, whose only job is to decide whether to let an event instance through to a related object or not.)

Conclusion: when you partition a system into smaller classes, expect to increase the effort you apply to Event Modelling.

32.5 Summary

We've discussed design issues and tradeoffs. We've shown how the 3-tier architecture can be used to minimise data migration costs, and hide data replication and aggregation from the Business services layer of code.

In summarising the conclusions of this chapter, we can list a dozen design or so principles for large systems.

• specify classes and relationships in the data structure rather than leave them to be constructed by programmers

• where it is justified (by data migration or performance costs) introduce an data abstraction layer to separate the entity model from the data storage structure (designed for flexibility and performance)

• don't store redundant data until you have established a clear business case in terms of speeding up enquiries or increasing the robustness of local operation

• specify 'replicated data' in the data services layer of code

• specify 'derived data' in the business services layer of code

• expect data migration and include it your plans

• buy much more database space than you think you will need

• refine the locking mechanism in proportion to the number of concurrent users

• don't unnormalise stored data into an aggregate table until you have established a clear business case in terms of enquiry speed

• define aggregate tables in the data storage structure rather than the entity model

• spend money on usability in proportion to the number of end-users who will benefit from your design efforts

• don't implement more features than you are asked to, but don't let this stop you thinking ahead and designing for maintenance

• when you partition a system into smaller classes, expect to increase the effort you apply to Event Modelling.

33. From design patterns to analysis patterns

Analysts need what might be called ‘analysis patterns’. These will be similar to design patterns for object-oriented programming in some ways, but different in other ways. This chapter focuses on a pattern they call State. The footnotes mention also Composition, Decorator, Facade, Adapter, Bridge and Proxy.

33.1 What analysts need from patterns

Design patterns have been developed by and for object-oriented programmers. The usual reference is ‘Design Patterns: Elements of Reusable Object-Oriented Software’ by Gamma et al.

Gamma et al. are widely and affectionately known as the Gang of Four. Their work is rightly acclaimed; it is an example to those teaching analysis of how to teach expertise (not just notations) via patterns.

Are design patterns relevant to Analysts and designers?

Analysts need patterns for processing persistent data

Most object-oriented designers work on systems that process transient objects; for example, compilers, graphical interfaces and financial modelling systems. So naturally, design patterns are mainly concerned with transient objects.

The data in a business database is composed of entity state records that represent real-world entities, long-lived entities that the business seeks to monitor and perhaps control. So analysis patterns must apply to persistent entities.

Fig. a repeats from chapter 1 a scale from transient objects to persistent objects. This is very closely related to the scale from type to state. The longer objects persist, the more that apparently fixed types become variable attributes or transient states.

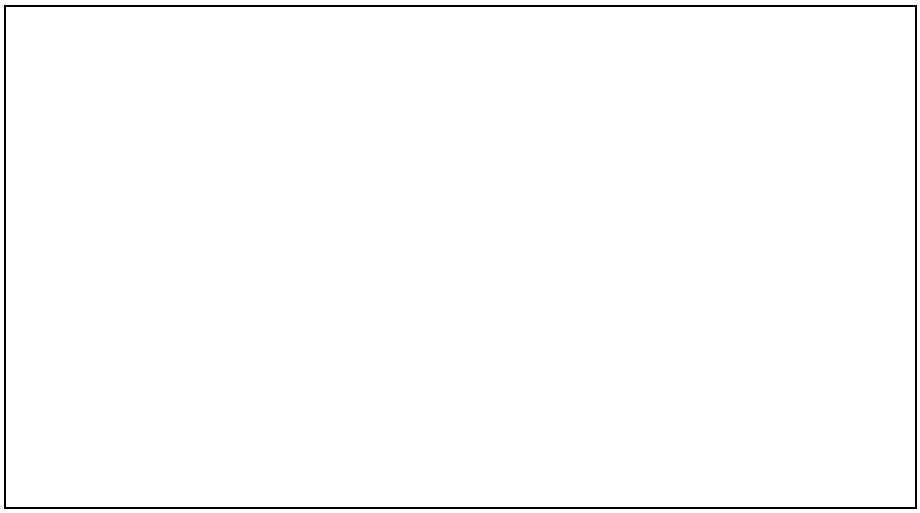


Fig. a

It turns out that the persistence of data has a big influence on patterns for software design, as

you shall see. You need a theory for how to manage states as well as types. Traditionally, different modelling theories have been applied to modelling types and states. One of our aims is to combine these theories.

Analysts need patterns that prompt questions

The Gang of Four say 'Design patterns solve many of the day-to-day problems object-oriented designers face.' Each design pattern fits to a given problem. You use a design pattern to solve a problem you already know you have.

Analysts need help with analysis, to discover what the problem is. A analysis pattern should help analysts to ask questions and find things out. It should help you to test and uncover problems in an existing specification. The most cost-effective training involves teaching bad patterns as well as good ones.

Analysts need patterns to do with real-world objects

Design patterns help designers to sort out computer-world objects. Analysts need to sort what things in the real world have to be represented in the system. Analysis patterns must help analysts to investigate the rules and practices of an enterprise in the real world, the one that is to be supported by the enterprise application.

Analysis patterns must be concerned with eternal verities in the way that real people and real businesses behave. At least, those eternal verities that can be captured in a 'conceptual model' of business objects and coded in the 'business services layer' of a system. Analysis patterns will be used mostly in defining the business services layer rather than the UI layer.

Analysts need patterns that are logical

Design patterns are expressed in physical terms, in terms of implementation mechanisms, and more specifically in terms of object-oriented programming mechanisms.

Analysis patterns should be expressed in logical terms. They must define characteristics of the problem domain rather than the implementation domain. Analysts should be able to use them without knowing what technology will be used to implement their design, be it C++, Java, COBOL or ORACLE.

For example, OO-style class diagrams specify where objects hold references to other objects. Fig. b shows two class diagrams on the left that are implementations of the same logical entity model on the right.

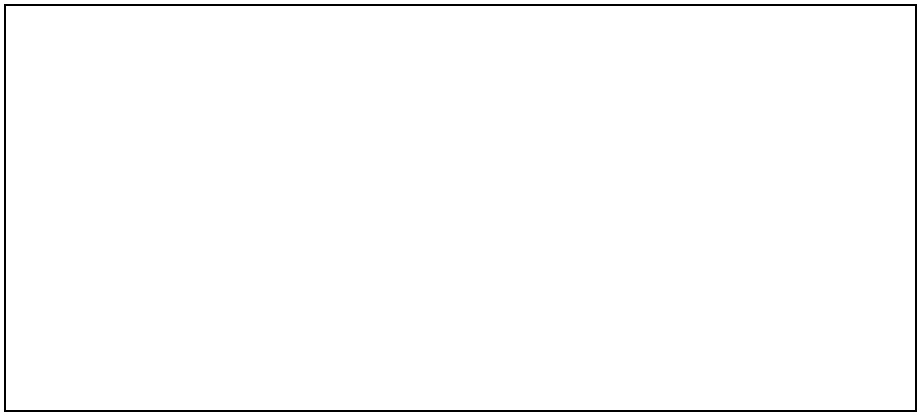


Fig. b

The logical notation above for modelling the cardinality of a relationship between classes is well known. See the chapter ‘Rules and relationships’ in *Analysis patterns.*

33.2 Inheritance and polymorphism in design patterns

Since the Gang of Four say ‘Almost all the [design patterns] use inheritance to some extent’ let us begin by reviewing the idea of inheritance. A class hierarchy or inheritance tree is a structure composed of superclasses and subclasses, wherein a subclass can inherit or override the properties of a superclass above it in the hierarchy.

object-oriented technologies help you achieve reuse by applying inheritance and polymorphism to a class hierarchy. See the chapter ‘Class hierarchies and aggregates’ for more about inheritance and polymorphism.

The general shape of a design pattern

Many of the Gang of Four’s design patterns are rather similar, based on a common template involving an abstract class, shown in Fig. c.

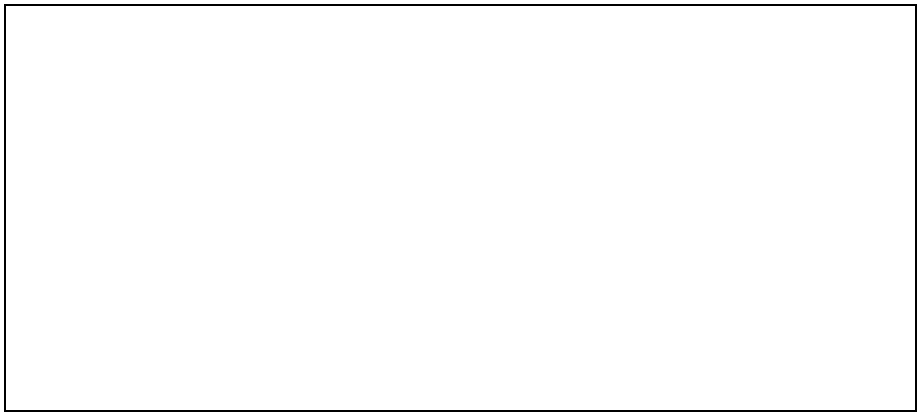


Fig. c

The ideas of patterns like this is to separate the interface of an object or a process from various possible implementations of it. Thus, design patterns of this shape capture expert knowledge about good uses for polymorphism and abstract classes.

The Gang of Four again: ‘When inheritance is used carefully (some will say *properly)*, all classes derived from an abstract class will share its interface. All subclasses will be subtypes of the abstract class.’ See for example their design patterns: Iterator, Observer and Abstract Factory.

Analysts need few patterns that feature class hierarchies

There may be a few over-enthusiastic object-oriented designers who believe that good design means explicitly spelling out all the class hierarchies you can find in the entity model of a system.

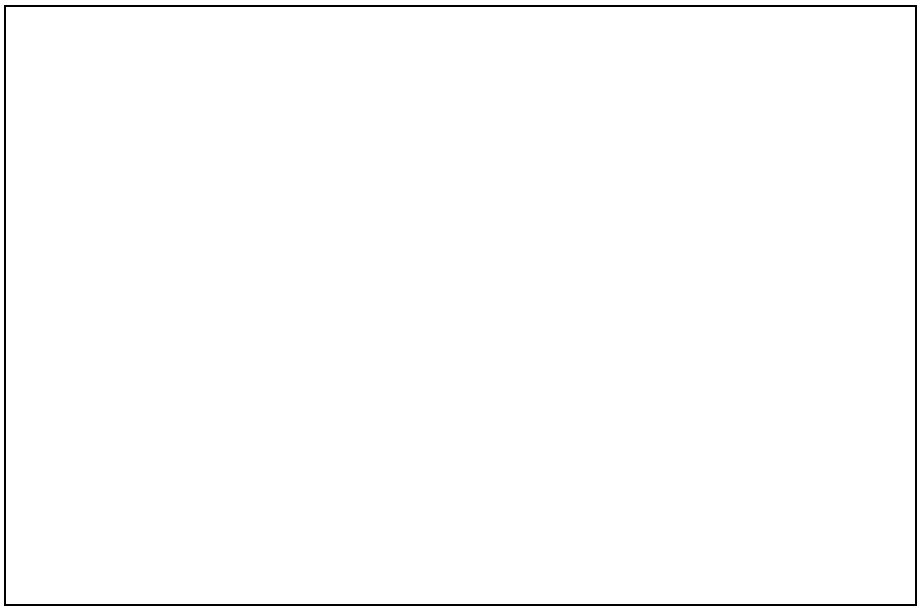


Fig. d

Class hierarchies are common in some kinds of software design. But chapters 5 and 6 have explained why you are unlikely to find so many in the persistent data structures of enterprise applications. Even the Gang of Four say 'Designers overuse inheritance. Designs are often made more reusable and simpler by depending more on object composition.'

Good analysts do not specify many class hierarchies in the entity model that specifies the persistent data structure of an enterprise application. Where the list of subclasses is very long, or variable, or there are complex overlapping hierarchies; then defining class hierarchies creates schema evolution problems.

Since analysts normally specify class hierarchies in other ways and places, few analysis patterns will involve inheritance, and very few will feature polymorphism, at least, not in the way that object-oriented designers think of these things.

You might suppose then that analysts will find little use for design patterns. But it turns out you can identify where some design patterns apply to enterprise application design. And you can reshape some design patterns into analysis patterns. We go on to reshape one design pattern for use by analysts, replacing the class hierarchy with classes connected by one-to-many relationships.

33.3 The State design pattern for object-oriented designers

Design patterns might be divided into three groups:

- not very useful in enterprise applications
- useful to analysts in the business services layer
- useful to designers in the others layers or the interface between layers.

The second group is the most interesting. The Gang of Four define a pattern called State that is designed to 'Allow an object to alter its behaviour when its internal state changes. The object

will appear to change class.' Let us look at how this design pattern can be reshaped for analysts.

Our tiny case study features one object class, Person, and two event classes, Employment and Death. The Employment event can only happen if the Person is unemployed. The Death event has two effects depending on whether the Person is employed or unemployed. Let us say Death (employed) goes on to affect Employer.

Fig. e shows the State design pattern in the entity model. Person and Person-Employment-Status are parallel associated objects. There is a class hierarchy under Person-Employment-Status of subclasses Employed and Unemployed.

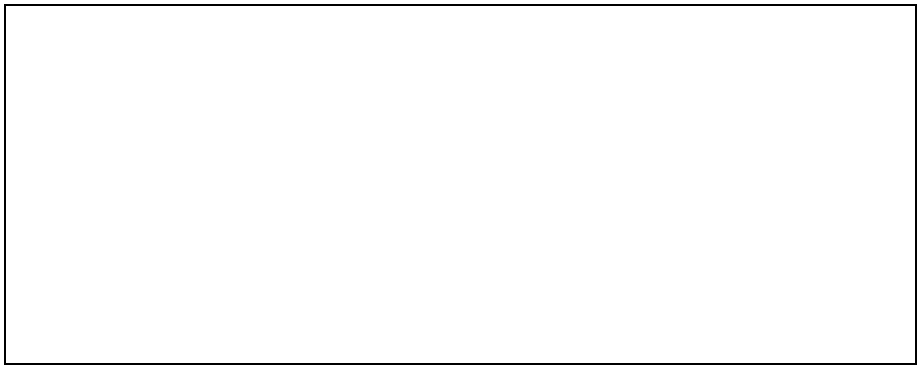


Fig. e

Messages to Person are delegated (by the implementations of the methods defined in its interface) to Person-Employment-Status where appropriate, i.e. where the response depends on the state.

You can use the State design pattern to implement one event that has different effects on an object in different states. You code each event effect as a distinct (polymorphic) method in a subclass of the status object. The status object divides the event between event effects.

Fig. f illustrates that you code the Death event in the Employed class and Unemployed class as two distinct methods. Personal-Employment-Status passes the Death event down to the appropriate subclass.

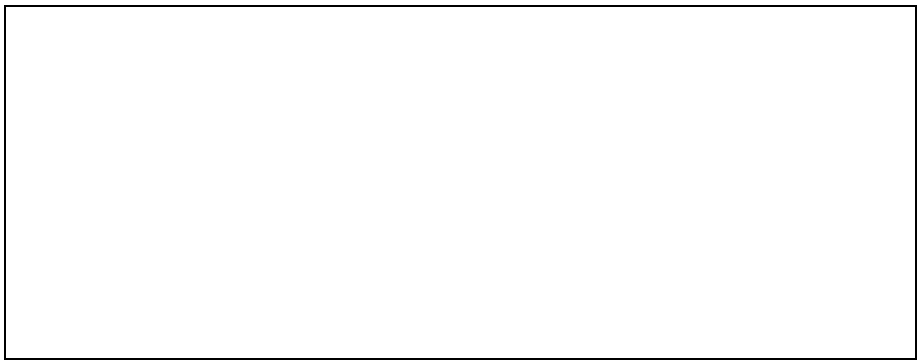


Fig. f

You have to code the selection between subclasses somewhere - in a data structure or a process structure. If you code it in the data structure, then an object-oriented programming environment can make the selection between subclasses 'under the covers' in any process that hits the status object. So you don't have to make the selection between types explicit in any process.

The State design pattern as a way to avoid selections in processes?

You might use the State design pattern as a device to avoid coding a selection or case statement within a method. You place the case statement in the data structure and code each option as a distinct method in a distinct class.

If the aim is to make code more maintainable, beware. First, what you gain in one way you lose in another; it becomes harder to see which methods are in fact related by mutual exclusion when an event is processed. Second, where data persists, it is easier to change the structure of a transient process than the structure of persistent data.

In enterprise applications, it is not reasonable or practical to remove all case statements from methods. It is like trying to define all constraints as state-transitions in state machines. This way of thinking, of trying to design everything using only one tool, is a trap to be avoided.

33.4 Parallel classes

There is one element of the State design pattern that is not so helpful to analysts - the class hierarchy showing each state as a subclass under each parallel aspect. Given that fixed class hierarchies do not abound in the data structures of enterprise applications, inheritance and polymorphism cannot be so useful as you might hope, and design patterns have to be reshaped for this kind of system.

However, there is another element of the State design pattern that analysts can use. We have argued from around about 1980, and most recently in the Computer Journal (1994), that a class is best divided into parallel aspects along the lines of its need to maintain state variables.

A state variable is an attribute with a short range of values that is tested as part of the precondition for one or more events. E.g. if a Person's Employment Status = employed, then an Employment event cannot happen. And if a Person's Employment Status = unemployed, then a Redundancy event cannot happen.

> Ask of a class: Does it maintain a state variable? If yes, create a parallel class to maintain it. Motivations include: keeping each class smaller and easier to comprehend on its own; suiting the paradigm of object-oriented programming; and tightly encapsulating the maintenance of a state variable.

This last means that the state machine for each class can be described elegantly using a regular expression notation, and this has further advantages in pattern recognition.

Where a class maintains several state variables, you should appoint a 'basic aspect' that is the master of all parallel aspects. Fig. g shows the basic class as the master of all the parallel status classes.

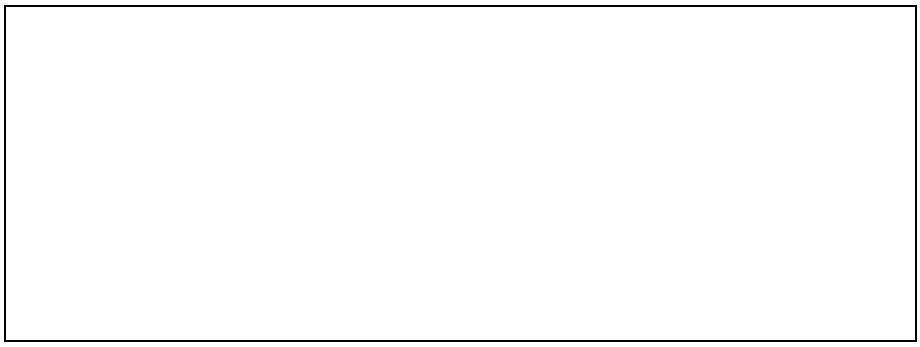


Fig. g

Fig. h shows a possible example. It has three cyclical states, each varying independently. There is a ‘boundary clash’ between the cycles.

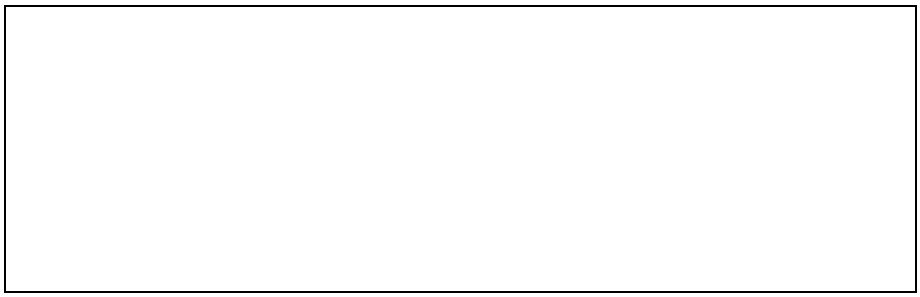


Fig. h

The basic class is responsible for maintaining object identity, and any attributes that can change in an unconstrained way as long as the object exists. The basic class so trivial it requires no state variable and there is little value in modelling its behaviour in the form of a state machine; it would be simply a sequence of creation, random updates, then deletion.

Some simple enterprise applications are composed of classes with only basic aspects.

Rolling up parallel aspects

In general, you should create a parallel class for each state variable that has to be maintained. But Fig. i shows that in simple cases, you might roll up one of the parallel aspects into the basic class. You don’t have to do this, but it is a harmless way to condense the specification and code in simple cases.

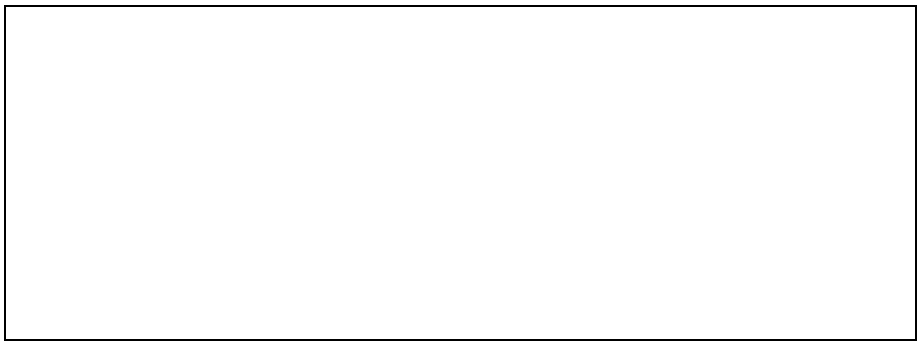


Fig. i

We have been talking about the specifying the *business services layer* of a system. You need not separate parallel classes in the data services layer. You can easily roll up all parallel aspects into one database table. One benefit: this speeds up performance, since each process

will have fewer data objects to retrieve and restore. One cost: it makes the interface between the business services and data services layers more complex.

33.5 The State design pattern reshaped for analysts

Applying the pattern in section 7.4 to the case study, you would specify a Person class that is careless of the state, and a Person-Employment-Status class that flip-flops between employed and unemployed. All the processing that depends on the state belongs in the Person-Employment-Status class.

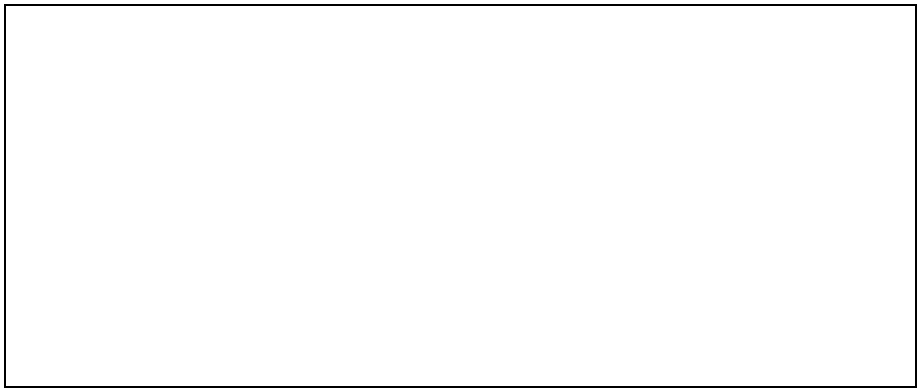


Fig. j

You can specify the subclasses not in the data structure but in the process structure of a Death event. Fig. k shows you specify the effect of Death on Person Employment Status as a selection between options Death (unemployed) and Death (employed).

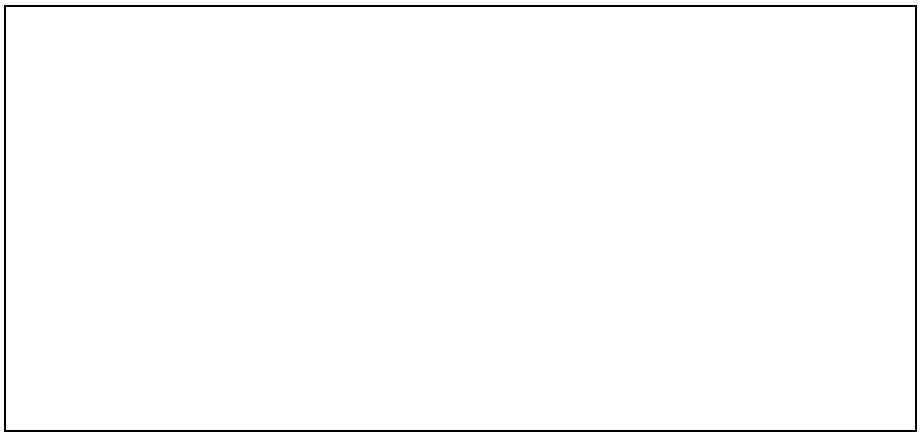


Fig. k

The event model is an abstract specification. When you come to code it, you might well code the selection between event effects as a case statement within the transient method for the Death event, rather than in the persistent data structure. This has advantages. In large enterprise applications, this will help to reduce schema evolution problems, since you can change the structure of a transient process more easily than the structure of persistent data.

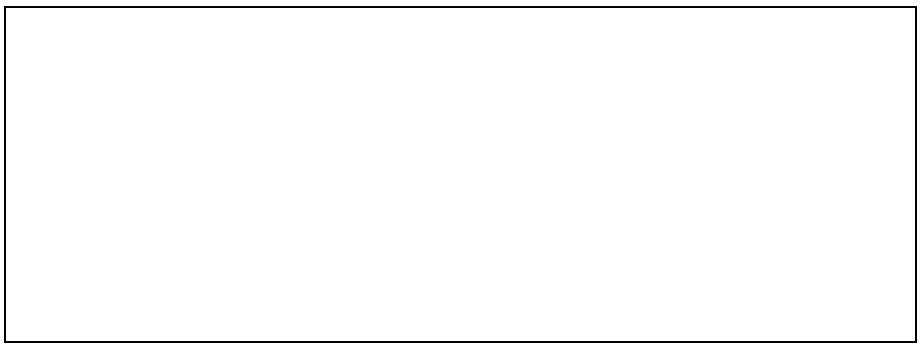
Some variations on this theme are shown below.

Status cycle as a historical record

a cyclical state, do users want to remember the history of past cycles? If yes, you can introduce a one-to-many detail class.

Fig. l introduces a detail class called Job.

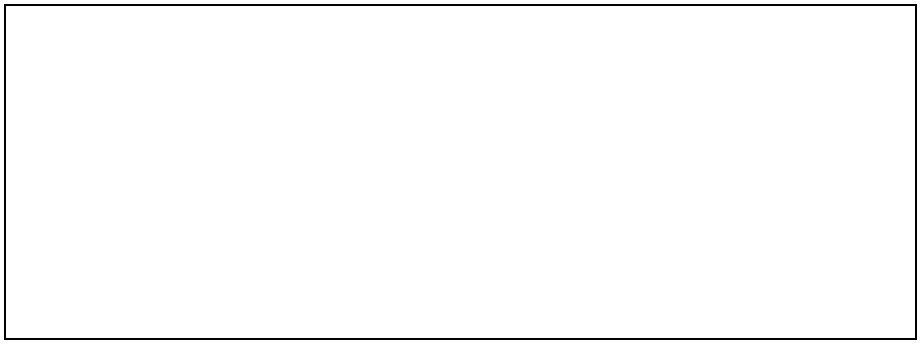
Fig. l



Status as an optional detail

Fig. m shows that if you don't want to remember the history of past cycles, only the current one, you might remove the fork from the relationship in Fig. l.

Fig. m

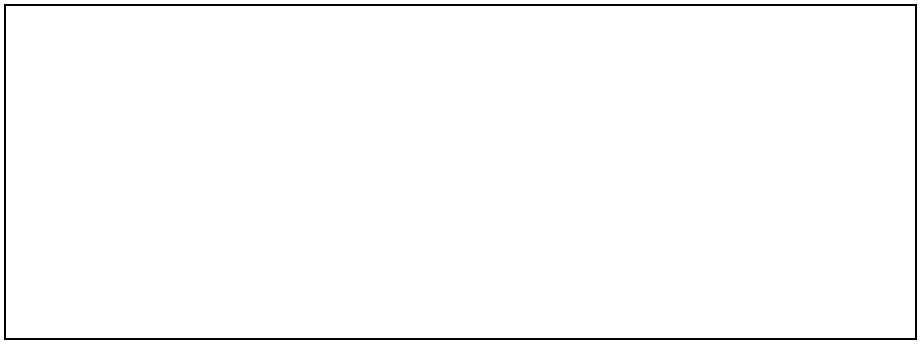


You don't normally see this shape however, because designers normally roll an optional aspect like this into its master class.

State variable as a domain class

Fig. n shows you might add a domain class for the state variable attribute, called Employment Status.

Fig. n



It is helpful to distinguish domain classes defined in the business services layer (under end-

user control) from classes defined in the UI or data services layer (under designer control).

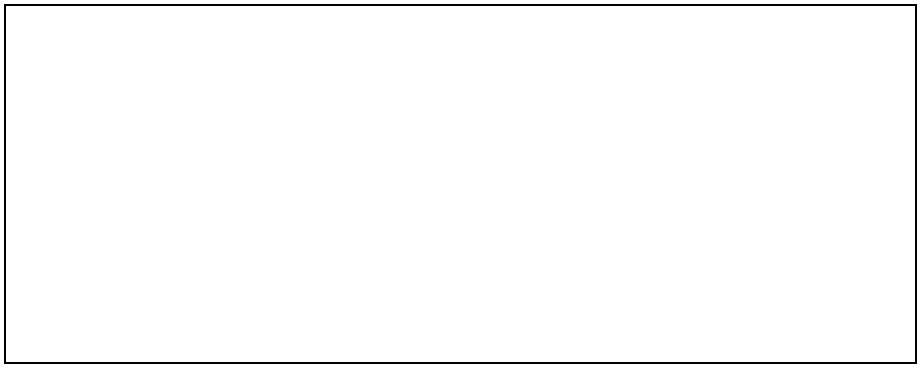
If designers want to define the values of a state variable in some kind of table, perhaps along with an expanded description of the state that is useful in error messages, you should define the domain class for the state variable in either the UI or data services layer.

If users want to be able to change the description of a state (‘unemployed’ to ‘redundant’), you may define the state variable as a state class in the business services layer. But be careful not to expose the class’s specification too far to manipulation by users; you surely don’t want users creating or deleting states, and thus changing the rules of the application.

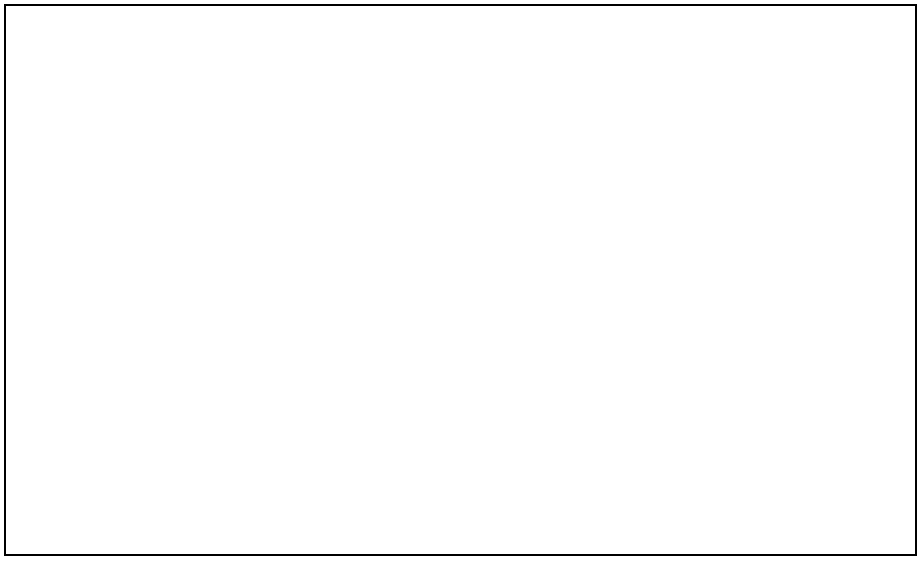
Domain classes are discussed further in other volumes in this series.

33.6 Recursive composition design pattern

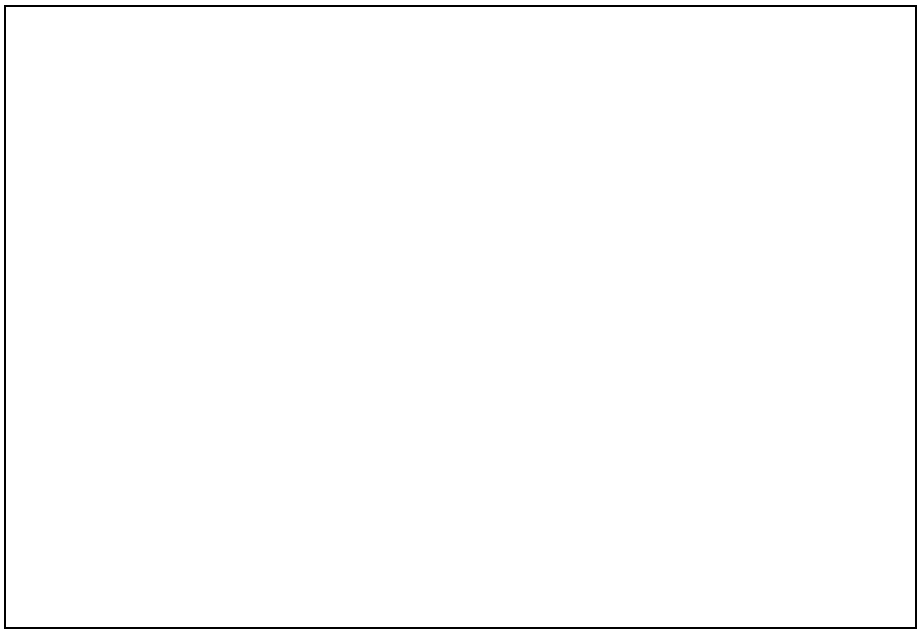
Composition defines an abstract class that provides a common interface for every level of a hierarchical structure. It specifies the bottom ends of the hierarchy as a special case. Curiously, it does not specify the top end as a special case, though this is sometimes necessary.



Recursive composition is familiar to most database designers. When database designers specify fixed-depth recursion, a different pattern emerges, in which the top and bottom ends of the structure appear under parallel classes.



However, the recursive structures found in enterprise applications are normally of variable depth; three varieties are possible.

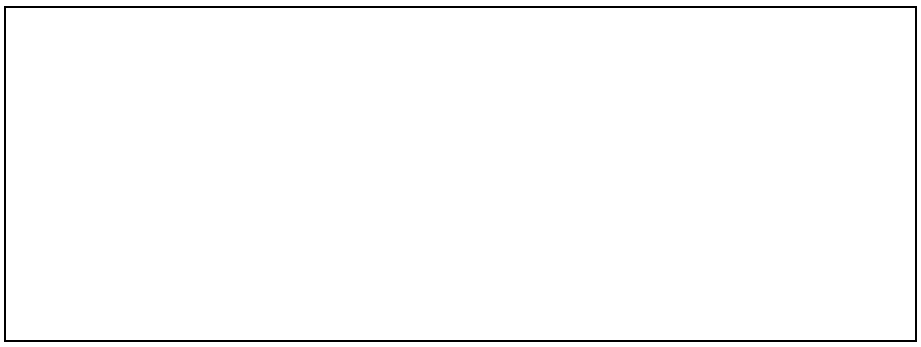


The volume ‘Patterns in entity modelling’ says more about such recursive patterns.

33.7 Recursive decoration

You can specify attributes as classes, then specify a new thing as a subclass of each relevant attribute class, using inheritance to obtain the attributes. But multiple inheritance can lead to complex structures, difficult to manage. You cannot make schema changes, alter the attributes

of a class, without changing the data structure and losing the instance data. To avoid these problems you can use the Wrapper pattern to add properties to a basic thing, one on top of another.



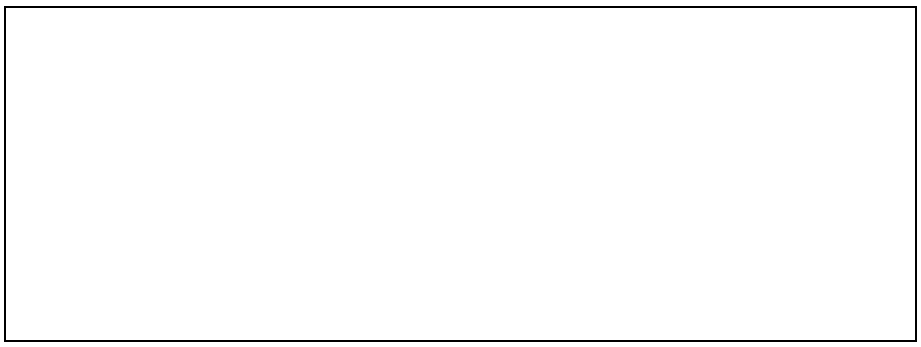
You store new attributes as object instances without changing the data structure and thereby losing all the instance data. The price is that you hide the basic object beneath layers of attributes. When a wrapper is added, the object identifier appears to change.

Each wrapper completely encapsulates the original object and any previously created wrapper. Each wrapper has a different object identifier. The identity of the original object remains the same, but since an external client can only call the outermost wrapper, the identifier appears to be that of the last wrapper.

Client --> Wrapper3 --> Wrapper2 --> Wrapper1 --> Object

In fact, some calls are dealt with in the wrapper without forwarding, or are supplemented before forwarding. This is the way the wrapper is able to add extra functionality.

This kind of data structure is too inefficient for database designers. Both multiple inheritance and recursive decoration are devices for designer-maintained data rather than user-maintained data. Enterprise application designers don't normally specify attributes in either of these ways. They use relational theory. They view objects as rows of a table, each row identified by a unique key. They view attributes as columns of the table. They may invert attributes to become key-only master classes at the top of one-to-many relationships as shown below.



You can make schema changes, add new attributes to a relation, without losing all the instance data; you can preserve the identity of objects stored so far. But you do have to recompile the data structure, and probably some of the programs, and retest the system.

33.8 Using design patterns to separate subsystems

The Gang of Four say 'Each design pattern lets some aspect of the system vary independently of other aspects, thereby making the system more robust to a particular kind of change.'

Many design patterns are about decoupling servers from clients. They help you to separate concerns for ease of maintenance, to keep distinct subsystems apart yet also connect them.

Experts advise keeping the bridges between subsystems as narrow as possible, keeping interfaces simple and economical. This is very much the idea behind one of the Gang of Four's design patterns called 'Facade'. This and other design patterns can be useful in bridges between subsystems of the 3-tier architecture.

Below, we've slightly edited and rearranged a contribution by Patrick Logan to the patterns group on the internet, in which he suggests the application of other design patterns to the 3-tier architecture:

'Constraints
'Despite the variation of user interfaces and databases, the system as a whole must maintain its integrity (adherence to system requirements). The logic and the system integrity checks represent most of the new development required.
'The user interface tier should interact with the user, but refer to the middle tier (business logic and integrity) for the computation. The middle tier should be implemented in terms of abstract objects, hiding the business logic from the user interface, and from the details of the databases.
'You can separate the three tiers using the structural patterns described in Design Patterns, such as Adapter, Bridge and Proxy.'

Analysis patterns will be about coherence and constraint, apply within a coherent subsystem, within a layer of the 3-tier architecture, rather than between them. Analysis patterns must apply within the business services layer of code, help you to get the functionality of a system right.

Analysis patterns must help you *integrate* concerns, help you to specify the coupling between business entities, to tighten the constraints as far as possible, so that these objects remain consistent one with another.

So broadly, one might say: 'Apply design patterns to loosen the interfaces between subsystems. Apply analysis patterns to discover and specify the constraints within a subsystem.'

33.9 More of what analysts need from patterns

We started by suggesting analysts need what we are calling analysis patterns. These will be similar to design patterns for object-oriented programming, but different in a number of specific ways. We've already suggested that analysts need:

• patterns for processing persistent data

• patterns that prompt questions

• patterns to do with real-world objects

• patterns that are logical

• few patterns that feature class hierarchies.

Both design and analysis patterns are concerned with smallish structures of relationships between elementary components of a system. Analysis patterns tend to be simpler than design patterns, more abstract in the sense of technology-independent, and they are perhaps more numerous.

There are a few more things to say about the use of patterns in analysis. Pointing up differences between design and analysis patterns sheds a new light on both fields of research.

Analysts need only a few patterns that feature recursion

Several of the published design patterns for object-oriented software construction feature recursive communication between instances of a class. The few analysis patterns that do feature recursion are interesting, but perhaps not so commonly used. See Footnotes.

Analysts need patterns that model business rules

Design patterns can help you build enterprise applications that are more robust in the face of changes, while analysis patterns will help you build enterprise applications that are correct in terms of applying constraints. Both can help you make the step from naive database use towards more complex database use. See Footnotes.

Analysts need patterns for object behaviour analysis

Design patterns appear in two dimensions of conceptual modelling - entity modelling and Event Modelling. Confusingly, the Gang of Four refer to patterns in object interactions as ‘behavioural’ patterns. We use the word ‘behaviour’ in a different dimension.

What we call the object behaviour analysis face of the conceptual modelling cube is to do with specifying the long-term behaviour of persistent objects in the form of life histories or state machines. There are many analysis patterns in state machines. This is an area in which analysis patterns work might contribute to design patterns work.

Analysts need patterns that suit database processing

Design patterns help with object-oriented programming technologies. Analysis patterns must help with systems that use database and transaction processing technologies.

But the distinction between technologies is not as fundamental as it looks. Analysis patterns may be implemented in object-oriented software. Design patterns may appear in enterprise applications.

Further reading

The volume ‘Introduction to rules and patterns’ takes up the theme of analysis patterns, or analysis patterns.

34. Appendix A: very general principles

This book is largely practical. There are a few abstract principles that underlay the discussion of patterns in this book and its companion.

34.1 There is no silver bullet.

A system is composed of many small elementary things (objects, facts, types, states, events and rules) connected together in various ways. You have to get down to the bottom level. You have to define all the elementary things and the relationships between them, at a level of description that can be executed on a computer. There is no way to avoid this. There is no way to avoid the pain.

34.2 A system is composed of connnected things

Everything in a system must be connected to everything within that systems, otherwise there must be two or more distinct systems. Patterns are about connecting things. There are recognisable and reusable patterns in how things are related. Patterns that connect just two or three things are the most reusable, but patterns that connect four or five things are more valuable.

34.3 The cardinality of connections is fundamental

The ‘how manyness’ of things in relation to each other is a fundamental kind of rule that has to be specified in each view of a system. You define one-to-one, one-to-one-or-zero, and one-to-many relationships not only between classes (in an entity model), but also between the concurrent objects affected by an event at a moment in time (in an event model), and between the events that affect an object over a period of time (in a state machine).

34.4 Most connections are associations

Things are naturally related to each other by association. E.g. A shoulder is related to an arm. An arm is in turn related to a hand. A husband is related to a wife. A divorce must relate to a previous wedding.

34.5 Persistence undermines compositions

Longevity turns composition relationships into associations. A composition relationship is an association, but strengthened by the rule that all the related objects live the same length of time. You can try to relate things by saying one is composed of others. E.g. A hand is composed of a palm, four fingers and a thumb. A car is composed of an engine, chassis, body, etc.

This is OK over short time, but longevity turns composition relationships into loose associations. You might lose a finger from your hand, or replace the engine of your car by another. You would better say a car is associated with a number of parallel aspects, each of them potentially optional or replaceable.

34.6 Persistence undermines type classifications.

Longevity turns subtypes into states of parallel aspects. You can relate things by saying one is a subtype of another. E.g. a Man is a Human; a Woman is also a Human. Similarly, Leg, Arm, Wing, Flipper and Tentacle are all subtypes of Limb.

This is OK over short time, but longevity turns apparently fixed types into variables or states. Under some legal systems, a Human can change Sex. You would better say a Human is associated with a number of parallel aspects - Sex, Job, etc.

A caterpillar turns into a butterfly. Exactly when in evolutionary history the forelegs of a monkey became the arms of an ape is an interesting question. You would do better to say a Limb has a number of optional parallel roles - Supporter, Hanger, Swimmer, Flyer, etc.

34.7 There is more than one paradigm, more than one orientation

Events coordinate interacting objects. Object-orientation and event-orientation are not in competition. They are orthogonal views of the same phenomena; equally valid and useful views.

34.8 Interaction is different from, more fundamental than, communication

An event reflects a natural phenomenon. An event model specifies the interactions between concurrent objects in a formal way. An event model is a directed graph; the event travels along each relationship in a one-way direction. But an event model 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 model, or another route. The interaction is more fundamental, more objective, than the messages that make it work.

34.9 Berrisford's law of assymetry

Nature abhors perfect symmetry. Assymetry tends to assert itself. If you discover two perfectly symmetrical things, you will normally destroy the symmetry by placing one over the other, or by inventing a third thing and relating both to it.

34.10 Redundancy is to be avoided - normally

If you say my shoulder is related to my arm, which is in turn related to my hand, there is no need to say my shoulder is related to my hand - this is implied. But not all redundancy is bad, since introducing redundancy into one perspective may reduce redundancy in another.

35. Appendix B: On the SmallTalk paradigm

Meyer discusses three technical advantages of the SmallTalk paradigm. These benefits apply largely to designers working with visual programming environments rather than business databases, and more to programmers than to analysts.

Conceptual consistency from using a single object-oriented paradigm

Only having to think in one object-oriented dimension is great for the programmer, but teaching analysts that everything in a real-world enterprise is an object doesn't help them. We should teach and encourage analysts to consider all dimensions of the problem they are studying. They need a framework that clearly separates the different parts and orthogonal views of the problem domain they have to analyse.

Manipulation of classes at run time

This is great for the programmer, and perhaps for iterative prototyping, but positively dangerous in full enterprise application development.

Soon after enterprise application is set live, analysts are faced with the need to change the database structure or the rules that guarantee data integrity *while the running system retains its stored data.*

Where run-time manipulation of rules is required, analysts should define the rules as attribute values of some kind of classification or rule entity type.

Where the business entity model is to change more fundamentally, beware that the stored data is a valuable company asset. The necessary reprogramming, retesting, retraining and data conversion are expensive. Analysts need help to tackle such 'schema evolution' in a strictly controlled and methodical way.

Use of class-level methods alongside instance-level methods

Meyer suggests that programmers may find this a mixed blessing. Part of the art is to keep levels of abstraction apart. Analysts have two orthogonal ways to separate levels of abstraction.

Instance from type

Analysts do separate type from instance in the business services layer by one-to-many relationships between persistent classes: say:

Road Type ---< Road ---< Road Use

Programmers may later introduce class-level 'methods' to process any event that cascades down these one-to-many relationships.

Class from metaclass

This is not a separation that Analysts worry about, but there is a sense in which the three-tier software specification architecture separates class from metaclass. It keeps apart:

- business services layer classes, such as Road and Road Use
- UI layer classes: such as Window and Button

• data services layer classes: such as Table and Commit Unit

Might one view the data services layer classes Table and Commit Unit as metaclasses representing business entity and business event?

36. Appendix D: 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 paper 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 behaviour 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 behaviour of each entity type and document it in a state machine diagram. He favoured using regular expressions as the notation and called them life history diagrams after Jackson I think.

• Object interaction diagrams: 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 modelling 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 schema 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 modelling techniques and incorporated object-based process analysis and design techniques.

Keith and John deemed object interaction diagrams 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 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 diagrams. 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 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 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 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 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 paper in the Computer Journal that showed how the benefits of inheritance (reuse and extendibility) can be achieved through modelling 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 for adoption by SSADM version 4.2.

1997 This book has examined the practical application of inheritance and polymorphism in enterprise applications. The companion volume ‘Event modelling for enterprise applications’ introduces improvements in the teaching and usage of event models, e.g. to include constraint discovery and specification.

37. Appendix C: References

Assenova P. and Johannesson P. [1996] Improving Quality in Conceptual Modelling by the Use of Model transformations Stockholm University

Boman M. et al. [1993] *Conceptual Modelling* 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

Gamma 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] paper for the OOPS group of the British Computer Society