How to know a good model

Richard Veryard, August 2001

Quality goals

Good Model

What are the modellers aiming for? Apart from having a good scope, what else makes a good model? (Quality of end-product)

- Comprehensible to all interested parties
- Correct - i.e. accurately representing the enterprise, to the satisfaction of all interested parties.
- Externally complete - i.e. completely describing the enterprise, covering all required functionality.
- Internally complete and consistent - i.e. with no missing cross-references, no unused objects, etc. (Some aspects of this can be automatically checked by an appropriate modelling tool.)
- Well-documented - i.e. with adequate and meaningful descriptions of all objects
- Non-redundant - i.e. only representing each fact/requirement once, without unjustified duplication or overlap between objects.
- Coordinated - i.e. consistent with architectures and policies, and with an appropriate level of consistency with other related models.
- Stable - i.e. capable of absorbing minor future changes to the enterprise without major changes to the model. (This characteristic is also known as resilience or robustness.)

Good Modelling Process

We can also identify the quality criteria for the modelling process:

- Ease of agreement by users of the model
- Ease of agreement by users of the design implications of the model
- Minimum 'thrashing' - i.e. going round in circles before agreement can be reached
- Minimum discovery of additional requirements during design and subsequent phases

- Relatively few surprises during implementation
- Low maintenance costs of system (owing to changes in model)
- Maximum learning for participants and entire organization
- Efficient & effective - i.e. achieving a good result with a reasonable expenditure of time and energy

Good Design

A design judgement needs to be evaluated at three levels:

- at the level of the system being designed, where the design should represent a good pattern, and embody good design values and practices
- at the supersystem level, where the designed system should contribute in a positive way to some broader system
- at the subsystem level, where the design should create an integrating structure/environment in which the lower-level detail can be worked out

A good design is a collection of good design judgements. We can talk about the overall value of a design - this is precisely what is required for a business case. A good design judgement increases the overall value of the design, but this overall value cannot be distributed arithmetically between the judgements that make up the design.

As the end-product of a design process, a socio-technical system needs to be evaluated in three ways:

1 From a combined socio-technical perspective

2 From a social perspective

3 From a technical perspective

In other words, some of the quality characteristics apply to either the social or technical aspects of the system, while some of the quality characteristics apply to both at once.

From a combined perspective, a systems design should be:

- Well-modularized - i.e. defining modules that have maximum cohesion and minimum connection
- Measurable and testable - i.e. the non-functional requirements should be objective and (if possible) quantified.
- Coordinated - i.e. consistent with architectures and policies, and with an appropriate level of integration with other related systems

From a social perspective, a systems design should be:

- Usable - i.e. fitting into the intended business environment, and providing useful support to the user in carrying out his/her job

- User-friendly - i.e. with computer functionality matching the structure of the business operations, so that the system works the way the user thinks

From a technical perspective, a systems design should be:

- Implementable - i.e. technically feasible on the chosen target platform
- Performant - i.e. with an adequate balance between speed, throughput and efficient use of computer resources.

Good Design Process

We can also identify the quality criteria for the design process:

- Correctness - i.e. no design errors found during testing or operations
- Maximum reuse of design components
- Minimum 'thrashing' - i.e. going round in circles before agreement can be reached
- Low maintenance costs of system (other than owing to changes in model)
- Maximum learning for participants and entire organization
- Efficient & effective - i.e. achieving a good result with a reasonable expenditure of time and energy

Reviewing quality of model

Introduction

In this section, we consider approaches and techniques for reviewing the quality of an information model. An inspection process may consider the structure of the model, and may test it for its ability to correctly represent all present and most future possibilities. Stability analysis considers the ability of the model to absorb likely future requirements without excessive difficulty.

Validation criteria

Good entity types

- **Clear Boundary.** There must be a clear boundary between the occurrences of the entity type, and the rest of the universe. In other words, if EMPLOYEE is to be an entity type, there must be no ambiguity as to who should count as an employee and who not.
- **Common Form.** The things that we are interested in about the entities must be reasonably alike, which then gives some structure to the entity type. We can define this structure in terms of the attributes of the entity type, and the relationships between the entity type and other entity types. Thus we can restate this condition as saying that the occurrences of the entity type are to have more or less the same attributes and relationships.

- **Standard Identification.** The entity occurrences must be identifiable in a standard way, capable of being distinguished as individuals, and practically countable. (Countability is a consequence of identifiability - if you want to know how many entities of a particular type you have got, you have to be able to avoid counting one entity twice.)

- **Finite.** The number of occurrences of the entity type must be finite. This is because the method is pragmatic - infinite models may be theoretically interesting, but are of no practical value.

- **Unity of Purpose.** The entity occurrences must play similar roles in the business or organization. It is likely that a common set of business processes are associated with a single entity type. The entity type must represent a single business concept. (Beware of **two-faced** entity types that fail to satisfy this criterion.)

Entity definitions

An entity definition should state two things clearly: a **membership rule** and an **identity rule**.

- The membership rule defines when something counts as an occurrence of the entity type. For example, does the entity type EMPLOYEE include or exclude recruits, pensioners, freelancers, women on maternity leave?

- The identity rule defines when two things count as the same occurrence of the entity type, or where one occurrence stops and the next one starts. For example, does the A40 count as one occurrence of ROAD or several? Does the journey from London to Oxford count as the same ROUTE as the journey from Oxford to London? Does the journey from London to Oxford by train count as the same ROUTE as the journey from London to Oxford by road?

You need both the membership rule and an identity rule in order to count the occurrences of an entity type. Thus the ability to count occurrences is a good test of these rules. But the mere fact that the modelling team has estimated the volume of occurrences is not proof that their definitions are perfect.

To test the definitions, you have to come up with test cases. A test case for an entity type definition will be in the form of a candidate entity occurrence (i.e. an entity that may or may not be one or more occurrences of the entity type). A given entity may be a test case for more than one entity type - in other words, it may not be clear whether it is an occurrence of one entity type or another or perhaps even both.

A further criterion that a definition must satisfy is that the number of occurrences must be finite. For example, the entity type GEOGRAPHICAL AREA could have been defined as any continuous area of land. Or the entity type BLEND could have been defined as any possible mixture of raw materials, in any proportions. These definitions would not work, because there would be infinitely many occurrences. The business cannot be interested in an infinity of things.

Even with finite sets, it is possible to define an entity type with an unmanageably large number of occurrences. For example, the entity type HUMAN GROUP could have been defined as any combination of living human beings. (This is a finite number: $2^n - 1$, where n is the number of living human beings. Such large numbers are effectively useless.) There must be some manageably finite subset in which the business is actually interested; thus what is often needed is some restriction on the definition, to identify this finite subset.

Two-faced entities

One difficulty with information modelling, is that of **two-faced** entity types that serve more than one purpose. This is found particularly with abstract entity types such as MARKET or ACCOUNT. The principle of data sharing seems to urge that each entity type serve as many purposes as possible, but there are situations where a single term hides a multiplicity of purposes, and must be pulled apart.

A good example of a two-faced entity type occurring in many models is PRODUCT. This has two aspects: what is bought by the customer, and what is delivered to the customer. Superficially these appear equivalent, but they often turn out not to be. Consider tinned peaches. The consumer selects a brand, and perhaps doesn't care where the peaches are grown. Indeed, the same brand of tinned peaches may contain Californian peaches at one time of year, and South African peaches at another time of year. Furthermore, peaches from the same source may be tinned under several different brand names (including supermarkets' own brand labels).

Thus we have many facts about a tin of peaches, including its brand name and retail price, as well as the source of the peaches inside. Confusion arises if we try to model all these facts in a single entity type called PRODUCT. Instead, it is usually a good idea to distinguish two entity types: PRODUCT and BRAND, with a many-to-many relationship between them.

The same physical object may be sold in a number of different contexts. A piece of foam rubber may be sold in sports shops as a mat, in specialized health-care shops as a physiotherapy aid, in furnishing shops as a sofa lining, and in an art gallery (after some mutilation) as a sculpture.

With software products, a similar situation seems to hold. The customer asks for 'Microsoft Word', and gets 'Microsoft Word UK Version 4.0'. So the software title would be the BRAND, and the software version would be the PRODUCT. However, some customers may ask for a specific version, so the situation is more complex than with peaches, where the customer can only get what the manufacturer chooses.

One way to get yourself completely confused is to build an intersection between two two-faced entity types, for example PRODUCT and MARKET. Then the ambiguity is multiplied.

	MARKET		
PRODUCT	before sale	at sale	after sale
what the customer buys			
what the customer gets			

How do we recognize two-faced entity types? One sign is an apparent conflict of business objectives - for example to increase the flexibility of supply without proliferating brands, or to increase the segmentation of target markets without fragmenting the support infrastructure. Another sign is irreconcileable differences between rival descriptions of the entity type, such as in estimates of size.

Redundancy

One of the aims of information modelling is to identify and (perhaps) remove redundant objects. We define redundancy to mean that a fact is represented in the model twice. Since facts are represented by objects, or combinations of objects, this

is equivalent to saying that one object in the model can be derived from other objects in the model.[1]

Some writers insist that all possible redundancy should be removed; here we take a more moderate approach, and insist that the redundancy should be analysed and controlled, but not necessarily removed.

There are two reasons for removing redundancy: to make the model simpler (and thus easier to understand and use), and to make the ensuing system more efficient. During analysis, we should only remove redundant objects for the first of these two reasons, since the second reason is a matter for systems design, but we need to establish all the facts of redundancy at this stage.

Storing a fact more than once opens the door to inconsistency, if the two versions of the same fact are incompatible. Thus non-redundancy makes it easier to ensure consistency.

However, some situations demand a higher level of control, where consistency needs to be actively checked, rather than merely automatically ensured. Two or more versions of a fact are deliberately captured, so that they may be compared, and discrepancies highlighted. The classic example of this is double-entry book-keeping, which indicates its redundancy by its very name.

Types of redundancy

There are four types of redundancy:

1 **Redundancy by repetition** - two or more objects in the information model representing the same fact in the real world.

For example, if the price on a customer invoice for an item is always the same as the price quoted in the catalogue, it would be redundant to represent this price twice, as an attribute both of PRODUCT ITEM and of CUSTOMER INVOICE ITEM.

2 **Redundancy by derivation** - an object in the model can be logically or arithmetically calculated from other objects in the model

For example, the price on a customer invoice for an item is always equal to the price quoted in the catalogue, minus the discount negotiated with that customer.

Some further examples of derivation are listed below.

3 **Partial redundancy by repetition** - two or more objects in the information model sometimes or usually represent the same fact in the real world, but there are some exceptions

For example, the price on a customer invoice for an item is always the same as the price quoted in the catalogue, unless the sales manager has authorized a different price for this sales order.

4 **Partial redundancy by derivation** - an object in the model can sometimes or usually be logically or arithmetically calculated from other objects in the model, but there are some exceptions

[1] Devotees of the relational model often use a different definition of redundancy, tied to the specifics of n^{th} normal form. Note that some intelligent translation of such concepts is required between the relational model and the entity-relationship model.

For example, the price on a customer invoice for an item is always equal to the price quoted in the catalogue, minus the discount negotiated with that customer, unless the sales manager has authorized a different price for this sales order.

Derived attributes

The most obvious form of redundancy is where attributes are repeated. More complex forms of redundancy arise where an attribute can be derived from one or more other attributes. We can identify the following common types of derivation.

1 To change units of measure (fixed) - e.g. to switch between feet and metres, multiply/divide by a constant.

2 To change units of measure (variable) - e.g. to switch between dollars and pounds, multiply/divide by a currency exchange rate obtained by table look-up.

3 To extract data (fixed) - e.g. to derive a calendar month from a date.

4 To extract data (variable) - e.g. to derive the age from the date of birth (depends on the current date).

5 To obtain a derived attribute by (arithmetical) manipulation of the other attributes of the same entity occurrence - e.g. GROSS AMOUNT equals NET AMOUNT plus TAX AMOUNT.

6 To obtain a single numeric value from a series of (input) numeric values. The simplest case is where the inputs are the values of a single attribute of a defined set of occurrences of some related entity type, and the derivation is based on a function such as COUNT, TOTAL, MINIMUM, MAXIMUM or AVERAGE. For example, ACCOUNT CLOSING BALANCE equals ACCOUNT OPENING BALANCE plus sum of (TRANSACTION AMOUNT)s.

In some of these examples, there is a loss of information in the derivation, in other words the derived attribute contains less information than the attribute(s) from which it is derived. In such examples, the derivation is necessarily one-way. So although we can derive the age of a person (in years) from his/her date of birth, we cannot derive the date of birth from the age alone. (This loss of information, or one-way derivation is sometimes referred to as **information entropy**). In such examples, there is no difficulty determining which attribute is derived, and which is non-derived.

In other examples, however, the derivation could be two-way. Thus we can either derive a measurement in feet from the measurement in metres, or the reverse. Thus it may be difficult to determine which of the two measurements is derived, and which is non-derived. The decision may finally be arbitrary, or based on majority convenience.

Derived attributes - design considerations

A derived attribute can be implemented in two ways: either stored and restored whenever the attribute values change, from which it is derived; or calculated and recalculated whenever its value is needed.

In the former approach, the derivation is performed at the earliest possible moment; in the latter approach, the derivation is performed as late as possible, on a 'just-in-time' basis. (There may be other, more complex design solutions, where the derivation is performed neither at the earliest possible, nor at the latest possible

moment, but at some intermediate point in time; for example, by a batch update program run overnight, when there may be spare computing capacity. Such solutions, although common, are more complex because the status of any given data item may be unclear.)

The choice between these approaches is a design decision; it has nothing to do with the meaning and use of the attribute itself, and has only to do with the technical efficiency of the computer software. This decision, therefore, should be left to the latest possible point in the software development process. (However, some tools blur this distinction between analysis and design, and encourage such design decisions to be made prematurely, in order to streamline the software development process.)

Inclusion of derived attributes

A derived attribute should be included in the information model:

1 When it represents a key performance measure (KPM) for the business or business area. Many managers measure performance of a business unit by a small number of ratios, and every such ratio is a derived attribute.

2 When it represents a shared information need of several users.

3 When it is required or referred to by several business processes.

4 When it is required as a status, partitioning or identifying attribute.

5 When its derivation is non-trivial.

During the early stages of analysis, it may be difficult to be certain whether to include a particular derived attribute in the information model. If in doubt, it is usually better to include it anyway, and then review the situation when analysing the logic of the business processes, or towards the end of the whole analysis project. At this stage, the necessity of including a particular derived attribute should be clearer, and unnecessary attributes can be deleted from the model. (Note: this requires conforming to strict model management procedures.)

Specific derivations

In some circumstances, a derived attribute or relationship be turn out to be **specific**. This means that the derivation specifies a particular attribute value or entity occurrence.

E.g., if there is a basic attribute AGE, then the user may have information needs such as OLDER THAN SPOUSE, OLDER THAN 18, OLDEST IN AREA or OLDEST IN LONDON, all derived from AGE. These may be what is meaningful to the user.

Such specific derivations must be treated with especial caution, because of the danger of their proliferation. (If OLDEST IN LONDON, then why not also OLDEST IN WEST LONDON, OLDEST IN EALING or OLDEST IN WEST EALING ?)

The contrary should also be considered: a non-derived relationship or attribute may be specific, in the sense that it specifies some other object or value, but that object or value is not represented in the model. For example, there might be an attribute OLDEST IN LONDON, but no entity type of which London is an occurrence, and no attribute of which London is a permitted value.

Such a relationship or attribute is of course not derivable from anything in the model. However, there may be a strong argument for considering adding the

specified object(s) to the model, in order to define the specific derivation within the model.

External derivations

Any attribute which is outside the control of an enterprise can be considered basic for the enterprise. Thus the postcode or zip code is under the control of the Post Office, and could be derived from the address and other geographical information. However, it is usually not necessary to model such complex geographic derivations explicitly.

Derived relationships

A relationship can also be derived, if it duplicates information represented elsewhere in the model. Apparently the most common situation is where one relationship is the 'sum' of two other relationships.

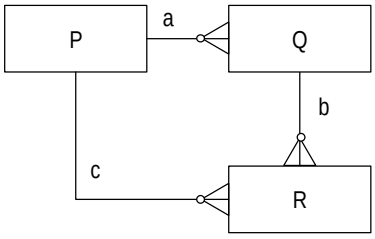

Figure 1: Redundant Relationship?

When I teach classes in information modelling, I display Figure 1 and ask which is the redundant relationship. There is always someone who tells me that the relationship **c** can be derived from the other two, and is therefore redundant. But I haven't yet told them what the letters stand for, so it is a trick question! Redundancy cannot be determined from structure alone, but depends on an equivalence of meaning.

> Before you read further, think of at least one situation, fitting this structure, in which **c** is redundant, and at least one situation in which **c** is not redundant.

If the relationship **c** between P and R is exactly equivalent to the relationship **a** between P and Q plus the relationship **b** between Q and R, then the relationship **c** can indeed be derived from the other two, and is therefore redundant.

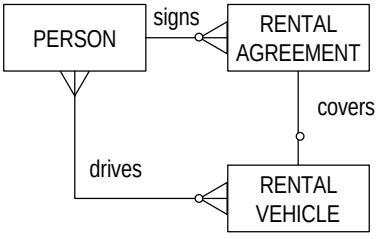

Figure 2: Another Redundant Relationship?

Suppose the business insists that the only person that can drive a rental vehicle is the person that signs the rental agreement. We can express this by saying that, for each occurrence of RENTAL VEHICLE, the occurrence of PERSON that you get by following the COVERS relationship followed by the SIGNS relationship is the same as the occurrence of PERSON that you get by following the DRIVES relationship. (In short: DRIVES = SIGNS + COVERS.) Then the DRIVES relationship is redundant, because it represents a fact that is already otherwise represented.

On the other hand, if the person that drives the vehicle need not be the same as the person that signs the agreement, then the DRIVES relationship is not redundant, since it represents a fact that is already independently represented.

This notion becomes more difficult to apply, however, when we progress to more complex examples. Figure 3 shows an example, modified from a real project.

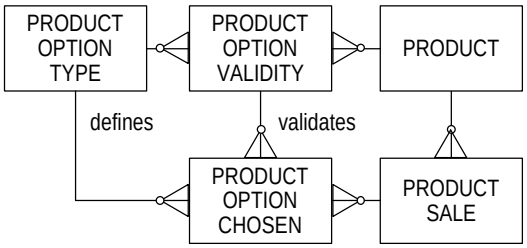

Figure 3: Redundant Relationship – more complex example

The two intersection entity types are PRODUCT OPTION VALIDITY, which indicates that a particular PRODUCT OPTION TYPE is available for a given PRODUCT, and PRODUCT OPTION CHOSEN, which indicates the options selected by a given CUSTOMER, within a given PRODUCT SALE.

The entity type PRODUCT OPTION VALIDITY is basically a look-up table, indicating to the salesman what s/he can offer to a customer. The entity type PRODUCT OPTION CHOSEN is basically a sales order detail. The relationship VALIDATES between them has a bad name, and raises all sorts of problems about exceptions and changes (what happens when a large customer is allowed a non-standard option, what happens when an product option is discontinued). Whereas the relationship DEFINES is much clearer, and allows changes and overrides to the look-up table to be managed properly.

Thus we would want to remove the VALIDATES relationship, and retain the DEFINES relationship. Or if only the VALIDATES relationship were included, and the DEFINES relationship were missing, we would want to replace the VALIDATES relationship with the DEFINES relationship.

This is a special case of the structure shown in Figure 4:

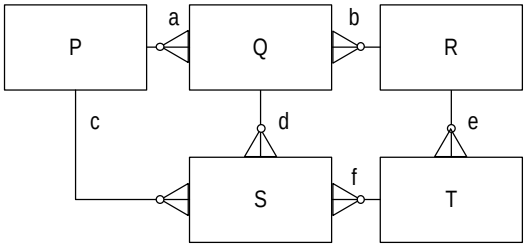

Figure 4

Suppose that **a** + **d** = **c**. In other words, for each occurrence of **S**, you get the same occurrence of **P** via the relationship **c**, as via the two relationships **a** followed by **d**.

Suppose also that **b** + **d** = **e** + **f**. In other words, for each occurrence of **S**, you get the same occurrence of **R** via the two relationships **b** followed by **d**, as via the two relationships **e** followed by **f**.

Clearly there is some redundancy. Which relationship is redundant? We could remove relationship **c**, without any loss of information, since it is equivalent to **a** + **d**. In the absence of **R** and **T**, this solution would probably be adopted. But in this example, removal of **c** only addresses one of the two redundancies. And once **c** is gone, all of the remaining relationships are required (i.e. none of them could be removed without loss of information from the model).

However, if **c** is retained and **d** is removed, it does away with both redundancies. Navigation between **Q** and **S** requires the intersection of the two navigation paths **ac** and **bef**.[2]

If we have a model with **d** but without **c**, we have a situation we can call **partial redundancy**, since there is no relationship that is entirely redundant. There are many other possible structures with this property. Such situations often emerge when entity types have many-to-one relationships to entity types that are merely intersections. It is recommended, wherever this is found, to follow the following procedure:

1 Verify that the relationships concerned are mandatory.

2 Verify that the two (or more) navigation paths always yield the same result, in other words that the two (or more) 'foreign keys' truly have the same meaning. If they do not, make sure this is clear from the names of the relationships.

3 Seek to restructure the information model to remove the redundancy. If the model does not have the relationship c, it may need to be added, so that the relationship d can be eliminated.

4 Alternatively, build the necessary integrity conditions into the process logic to control the redundancy, if it is convenient to retain it. (This is not addressed here.)

Besides the redundancy argument, there is another advantage of replacing **d** with **c**. It is that a relationship to an intersection entity type is often hard to name and define, whereas a relationship to a non-intersection entity type is often more meaningful and stable. Thus the change not only improves the logical quality of the information model, it may also improve its clarity and relevance to the business.

In conclusion, we should note that there are very complex structures that can arise in an information model, and considerations of relationship redundancy are by no means always easy to resolve. The analyst must understand and confirm the business rules and integrity conditions that are being expressed. Weaknesses in an information model will show up as awkward or redundant structures in the database. A good modelling tool makes this visible, which should provide help to the analyst.

[2] It will be seen that this solution assumes the intersection of **a** and **b** is unique. In other words, for each pair <p,r> of occurrences of **P** and **R**, there is exactly one occurrence of **Q**. But if this is not true, there will be another identifier of **Q** that can be 'normalized' out, and separately related to both **Q** and **S**.

Derived entities

If all the attributes and relationships of an entity type are derived, then the entity type itself is derivable. Such an entity type may be useful, however, because it encapsulates some information that is required frequently, perhaps by several different business processes.

For example, if a supermarket has electronic point-of-sale equipment, to record details of each item sold, the current stock levels could be derived from two entity types: DELIVERED ITEM and SOLD ITEM (perhaps together with a factor for breakage, spillage and shoplifting). However, it may be useful to define a derived entity type called STOCK ITEM. This enables the derivation algorithm to be defined in one place (against the derived object), rather than repeatedly for each business decision process that refers to the stock levels.

Benefits of derived objects

Defining an algorithm once, rather than repeatedly, has three benefits:

1 It reduces the amount of analysis work, since the algorithm only has to be defined once

2 It reduces the amount of system development and maintenance work, since any code required to implement the algorithm only needs to be generated once. Subsequent changes in the algorithm can be carried out in one place.

3 It ensures consistency of decisions made using these derived data.

This is one of the ideas behind the object-oriented approach, but can equally be supported using the entity-relationship-attribute model, since entities, relationships and attributes can all be regarded as 'objects' (in a very abstract sense).

Finite knowledge

Here is an example of how the need for a finite model may affect the definitions of entity types and relationships.

Suppose we have a many-to-many involuted relationship on PERSON, indicating that one person is a parent of another. Not everyone has children, so the relationship must be optional downwards. But everyone has two parents, so it would seem appropriate to make the relationship mandatory upwards.

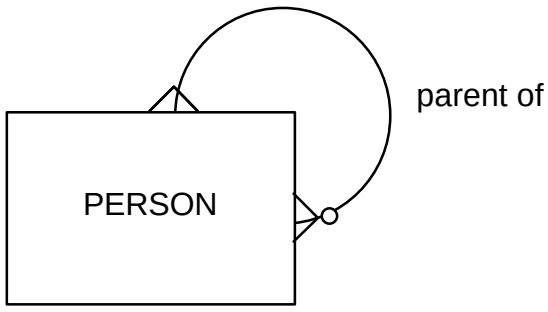

Figure 5

But although this conforms to biological theory, it doesn't correspond to what we know (or are capable of knowing). Not everyone has two **known** parents. Even if A knows that B was his ancestor, and B knew that C was his ancestor, it doesn't follow that anyone now can trace the line of descent from C to A. If you try to make

the relationship mandatory, you cannot stop anywhere, you have to go all the way back to the apes (or angels, if you prefer). If you want to stop somewhere, then the first person in your model doesn't have an ancestor in your model.

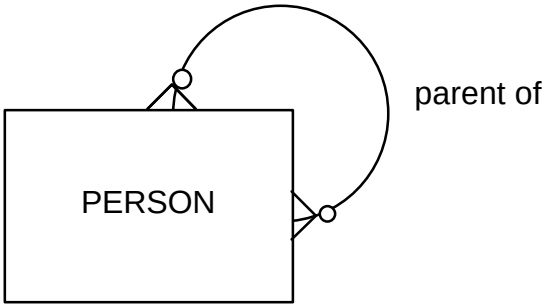

Figure 6

This illustrates two general points: first, that a model has to be a finite subset of the real world. We can only cope with a finite number of occurrences of PERSON. This has the consequence that some concepts (such as the *parent of* relationship in this example) do not bear their common-sense meaning. Instead, the restrictions of the model impart a subtle difference to what the concept means and how it can be used.

And the second point is that an information model, although it is not limited to what we happen to know already, is limited to what it is possible for us to know. (We may have to make some practical rules about the level of certainty that we require to accept something as knowledge.)

Signs of Trouble

Entity types

- Entity type name hard to find. Failure to find an acceptable name may indicate that the concept is not useful, but it may also mean that there are several overlapping concepts that must be disentangled.
- Entity type refers to existing representation of something: REPORT, LINE, ITEM, RECORD, CODE, RESULT.
- Single-occurrence entity types HEAD OFFICE
- Container entity types ARCHIVE, CATALOG, DIRECTORY, STORE

Relationships

- Relationships that imply transactions or events. This suggests another hidden entity type.
- Hierarchical structures with fixed number of levels. Consider replacing with involuted relationship.
- Universal relationships – where everything is related to everything – provide no information

Attributes

- An attribute can only belong to one entity type. Thus two attributes, with perhaps the same name, but apparently belonging to different entity types, should be examined to discover whether they are in fact the same attribute (although it is not always a trivial matter to determine this).

- An attribute can only exist once. Thus two similar attributes belonging to the same entity type, for example: STOCK LEVEL and INVENTORY QUANTITY. should be examined to discover whether they are in fact the same attribute, under two different names.

- Repeating groups. This suggests another hidden entity type.

Improving the Model

Questions about entity types

- How does it enter the world of interest? How long may it have existed before we become interested in it? Are we then interested in its history before that point (i.e. backdated interest)?

- How does it leave the world of interest? Does it continue to exist in some sense outside the world of interest? Can it then re-enter the world of interest, and do we care to connect it to its previous manifestation?

- How does it change? How much does it have to change before it becomes something else?

- Can it merge with another entity? Can it exchange aspects of itself with another entity? Can it divide into two or more entities?

- How do we count several occurrences of the same entity type? How do we tell that we only have one occurrence, rather than several, or several rather than one? What does this mean, for this entity type?

- What roles does it play? Do the answers to the above questions differ according to the role we are focussed upon?

- How do we discover that two entities we have met playing two different roles are in fact the same? What does this mean? How does it change our perception (if at all) to assert identity between two previously separate entities?

Improving relationships

The weakest form of relationship is a transferable fully optional many-to-many relationship, since this imposes fewest restrictions on the entity types. A mandatory one-to-many relationship is much stronger, since it rules out many possible combinations.

Some modellers prefer to start with relationships in their strongest form, and then look for counter-examples. They will therefore assert, for example, that every CUSTOMER always does business in a single CURRENCY, until they can find evidence to the contrary.

Other modellers prefer to start with relationships in their weakest form, and then add properties to the relationships only when these can be demonstrated. They will

therefore assume that at least one CUSTOMER does business in more than one CURRENCY, unless they can find a good reason to restrict this.

The advantage of the former strategy is that the models will be simpler and easier to understand. Complexity is added only when required, and it is added progressively, thus enabling participants to follow the process. The disadvantage is that many people become attached to their models, and find it difficult to motivate themselves to find contrary evidence. Thus the model remains simple, and fails to reflect the true complexity of the area being modelled.

The advantage of the latter strategy is that it is safer. The model will be more powerful, and the modellers will be motivated to find as many valid simplifying restrictions as possible. The disadvantage is that the initial version of the model may be extremely complicated, and thus exclude some participants from the process.

Test cases

Introduction

This section contains guidelines for the critical review of an information model. It will be relevant to those performing quality inspections on entity-relationship models produced for strategic planning, analysis, or any other purpose. Although models for different purposes, at different stages of the systems development life cycle, carry different expectations about the level of detail and the level of abstraction, similar principles and methods apply.

This may seem to beg the question, whether quality assurance is best served by external experts carrying out quality inspections. By external here we mean external to the modelling team, thus a person who was not involved in building the model. External inspections can be carried out by employees of the same company, while a consultant who played a significant role in the project, is perceived to have played such a role, and is therefore implicated in any weaknesses, is not external in this sense.

External inspections inevitably create an air of tension. It is never easy (although important) for the inspector to phrase criticisms in a positive way, to avoid getting the modelling team's backs up. And it is not easy for the modelling team to remain cheerful and positive while its hard work is taken apart. If the inspections are to be carried out by such external inspectors, therefore, we can expect the modelling team to become defensive and the inspectors to become adversarial.

Full-time inspectors tend towards one of two possible demeanours: a fixed joviality or an unshakable gloom. For this reason, it may be healthier to have part-time inspectors, who perform an occasional inspection interspersed with other activities, rather than professional critics.

For the inspections to be successful, it may even be necessary for the focus of the inspection not to be predictable by the modelling team, lest it be possible to deliberately hide the dodgy parts of the model from the inspector. A degree of idiosyncrasy on the part of the inspector may therefore be worth cultivating. And if a model is to be inspected several times during the project, there may be as much benefit in having a different inspector each time (who will uncover different problems) as keeping the same inspector (who will become familiar with the model).

Internal inspections, where the team itself carries out a structured walkthrough, may avoid some of the confrontation of external inspections, but despite this they may be less effective.

But it is not our intention here to analyze the psychology of criticism in detail, nor to prescribe managerial and motivational techniques for making one's criticism more palatable. Our intention here is merely to provide techniques for spotting potential weaknesses in an information model.

Reading and critiquing models

Many people find it very difficult to be objective about a model that they have not participated in building. They critique it by comparing it with the model they would have built themselves. This is a disastrous technique, and usually leads to complete rejection of the model by the inspectors, and complete rejection of the inspection by the modellers.

A much better technique is to try and understand what the model is saying. What would reality be like, if the model were true? What would the implications be? Is the model internally consistent?

Thus, instead of going in heavy with both boots:

> *"Well, I wouldn't have done it like that, oh no. This bit is obviously redundant, and what on earth do you need that bit for?"*

the inspector should be able to achieve just as much with a softer approach:

> *"Gosh, you see the enterprise in much more complex detail than I do. I'd never have thought you'd need this bit. And to be honest, I'm still having difficulty working out what you are going to do with that bit."*

Model test procedure

The procedure is then as follows:

1. The inspector generates a test case.

2. The modelling team then says which entity type(s) the test case belongs to, and whether it counts as one occurrence or several.

3. If the modelling team shows uncertainty or disagreement, then this suggests that the test case has not been considered before. It may then be worth following the test case through the rest of the business model - are the relationships and attributes optional or mandatory, are the processes valid?

4. Even if the modelling team shows no uncertainty or disagreement, the answer should be compared with the definitions. If the definition is too simplistic or too vague, then it should be revised/expanded. It is always worth adding good test cases as examples.

5. If the modelling team claims that the test case is entirely outside the scope of their model, this should be documented and notified to Development Coordination. If the modelling team asserts or guesses that the test case has been covered in a different business area, then this should be checked against the model of that area.

Generating test cases

There is a problem here. How does the inspector generate the test cases in the first place? Familiarity with the industry or function certainly helps, but is not absolutely essential. Here are some suggestions:

- Look for entities playing several roles simultaneously. For example, subcontractors acting both as ORGANIZATIONAL UNIT and as SUPPLIER.
- Look for the beginning and end of the entity life cycle. What about people before they are employed, after they are employed? Are they covered in the entity type definition? Do they contradict the relationship and attribute properties?
- Pay attention to the abstract entity types. Are their identifiers meaningful? Would you know where one occurrence stopped and the next one started? Is this clear from the definition?
- Actual performance is likely to be compared with plans, budgets, forecasts, actual performance elsewhere.
- Time and place are always difficult areas. Consider whether the model has established a sufficiently general and powerful set of concepts.
- What are the obvious information needs for management and control of the area? How are these supported by the model?

Relationship and attribute definitions

The inspector must also examine the definitions of relationships and attributes, especially those where the meaning is not obvious from the name.

- Every attribute called SOMETHING TYPE or SOMETHING CATEGORY needs to be defined. If there is a set of permitted values (as for example with SOMETHING STATUS) these may also need to be defined.
- The definition of an attribute of an entity type should make it possible to state the value of the attribute for any given occurrence of the entity type.
- The definition of a relationship between two entity types should make it possible to state, for any given occurrence of one entity type, which (if any) occurrences of the other it is paired with.

A test case for an attribute definition is therefore an entity occurrence (possibly hypothetical). The question is then: what would be the value of the attribute for this entity. If the answer is unclear, if the attribute could take several values, or a value outside the defined domain, or (for a mandatory attribute) no value at all, then the attribute is problematic.

For example, consider an attribute that depends on a subjective assessment of the entity. Even for a well-known occurrence of the entity type, everybody might differ as to the value of that attribute. You could consider replacing the subjective assessment with a more objective measurement. The trouble with subjective judgements is precisely that there can be several of them, by different people at different time. If it is necessary to retain such a subjective judgement in the model, then either make it clear in the attribute definition whose judgement is to be captured, or define a separate entity type called, say, JUDGEMENT or ASSESSMENT.

A test case for a relationship definition is also an entity occurrence (also possibly hypothetical). The question is then: what is paired with this entity under this relationship? If the answer is unclear, or if the answer conflicts with the properties of the relationship, then the relationship is problematic.

For example, there might be a relationship between POLICY and PLAN, called INFLUENCES. How do you define which policies influence which plans? Perhaps all policies influence all plans to some extent, so it is not a useful relationship. You should explore the plans influenced by a given policy, or the policies influencing a given plan, to discover what information really needs to be modelled.

Strategy versus detail

What is the difference between reviewing a strategic information architecture and an information model for analysis purposes? Should the same standards apply? Some might argue that the above procedures are too pedantic for strategic models. It all depends on what you think the strategic information architecture is for. And what are the consequences of getting it wrong.

The point of a strategic information architecture, in my view, is to identify opportunities for generalization and abstraction, so that analysis projects can be given clear and minimally overlapping scope. This means that the definitions of the major entity types should be thought out fairly thoroughly. Even if the entity type has no formal identifier, it should at least have an identifying strategy. (In other words, some thought needs to have been given to the kind of identifier that would be appropriate.) However, the relationships and attributes need not be defined in detail.

We could give many examples of inadequate thought during the strategic planning phase, causing serious problems for subsequent projects. Here is just one example. In the Oil business, petrol is sold, not directly to motorists but to petrol stations. Some departments refer to the petrol station as the customer, while others refer to the motorist as the customer. In one Oil Company's information architecture, however, there was a major entity type CUSTOMER, completely overlooking this ambiguity. This caused problems for several development projects, and for Development Coordination. Such homonymy was too broad to be sorted out within the information model of a single business area, and needed to have been addressed at a strategic level.

Permutations and combinations

Where we have entity subtypes, or attributes with a small finite number of possible values, we can generate all the permutations of these values or subtypes, to check that we have considered every combination.

A useful albeit primitive tool for doing this is to create a strip of card for each attribute (including the classifying attributes for subtypes). Onto each strip you write the possible values. Then the strips are placed parallel on the table, and slid up and down, to generate test cases.

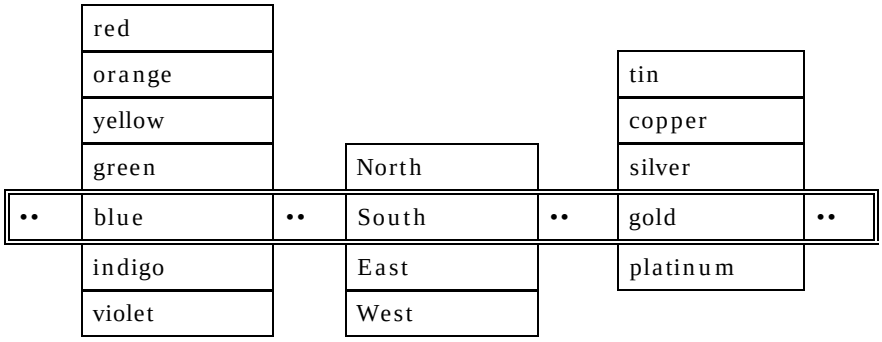

Figure 7

This technique can also be used to generate test data.

Conclusion

An information model develops through a dialectical process of example and counter-example. An external inspector contributes to this, with test cases against the model, and at the same time can assess the overall quality of the model by its ability to absorb such test cases. But our method is not dependent on external inspection; it is equally open to the modelling team, and to consultants attached to the team, to generate these test cases.

But the generation of such test cases is an art, not a science. There is no mechanical way of producing them. Their production requires a mixture of relevant experience, abstract models (e.g. general models of marketing intelligence, or of management control) and intuition.

Model inspection checklist

Initial data model

- Are all entity types within the scope?
- Has each entity type been properly described?
- Does the definition make clear what our interest in the entity type is?
- Is each relationship a true relationship?
- Are any new entities types really subtypes of existing entity types?
- Do selected entity subtypes have sufficient common attributes or relationships?
- What business "rule" does the relationship represent?
- Are there any redundant relationships?
- Are there any many to many relationships? How are these justified?

Final data model

- Has each entity type been completely defined?

- Have classifying attributes been identified for each subtype?
- Are values/permitted ranges of classifying attributes specified?
- Has a unique identifier been declared for each entity type?
- Has each relationship been defined?
- Should any remaining involuted relationships be expanded into separate entity types? Which ones?
- Has cardinality and optionality of relationships (eg. one to many) been confirmed?
- Have these business rules been confirmed by a user group?
- Could these business rules change in the future? Specify changes.

