AP-M Appropriate Process Movement

Domain Modelling

Paul Oldfield

Mentors of Cally Ltd.

Member of the Appropriate Process Movement

http://www.aptprocess.com

Abstract

The concept of Domain Modelling takes several forms. This document is about one of those forms. The concept, its philosophy, its uses and advantages, and ideas on how to perform domain modelling of this form are all introduced. In depth discussion is not attempted, here you will find the merest introduction to the topics. This document is not static and may be updated as the author sees fit.

Copyright Note

Contents

Abstract 1
Copyright Note 1
Contents 1
Introduction 2
Business to Analysis Model: Outline of Two Approaches 2
  The Robustness Diagram Approach 2
  The Class, Responsibility, Collaborator Approach 3
What is a Domain Model 3
  Why do different people have different ideas on this? 3
    Classes and Logical Model in Analysis model 3
    The RUP idea 3
  CRC origins 4
  Invariant Properties 4
How to Model the Domain: Basics 5
  Who, What, Where, When, Why, How 5
  Responsibilities in the Right Place 5

Abstractions 6
Responsibilities : Knowledge; Behaviour 7
Abstractions, Responsibilities and Relationships 7
Controller Objects? No, Thank You 8
Collaborations 8
Rules and Constraints 9
How to Model the Domain: Finer Points 9
Requirements Capture 9
Domain Expert has Left The Room 10
Independence of Use 11
Use Roles: Think of each abstraction in isolation 11
The 'feel good' factor 11
Controlling Scope 12
How to Model the Domain: Esoterica 12
Precondition – Postcondition Mapping 12
Model in Parallel 13
Domain Expert Learns Domain Modelling 13
Philosophy 14
Advantages and Disadvantages 14
Reuse 14
Flexibility 14
Extend don't Change 14
Cost 15
Summary 15
Acknowledgements and References 15
References 16

Introduction

Different people have different ideas about Domain Modelling; what it is, what use it is, how to do it, why one should do it, and in what circumstances. In this document, one particular set of answers to those questions is presented. In brief. The full answers would fill a book, I don't have time to write it, and it wouldn't sell very well.

Business to Analysis Model: Outline of Two Approaches

Of all the topics in OO software and systems development, the problem of arriving at a suitable Object Model starting from a Use Case Model is probably that which most often gives cause for concern. Here two approaches are outlined

The Robustness Diagram Approach

An outline of the Robustness Diagram approach is given in the article at

http://www.sdmagazine.com/documents/s=733/sdm0103c/0103c.htm and the subsequent article in that series.

In short, Use Cases are first modelled in terms of Boundary, Controller and Entity objects, with all responsibilities for behaviour initially assigned to Controller objects. As development progresses, these responsibilities are pushed down from the Controller objects to the Entity objects, until ideally the Controller object is slimmed down to at most a script for implementing the specific Use Case.

The Class, Responsibility, Collaborator Approach

An outline of a particular version of the CRC approach is given in this document.

In short, the domain is modelled independently of how the model is to be used in a given system, responsibilities are assigned to abstractions during that process, and the responsibilities required to realize specific Use Cases are identified as the Use Case is mapped onto the Domain Model.

What is a Domain Model

A domain model is a model of the domain within which an Enterprise conducts its business. The Domain Model for one Enterprise should be the same as that for any other Enterprise conducting business in the same domain. When we get down to more detailed levels, different people have different ideas about what constitutes a Domain Model.

Why do different people have different ideas on this?

There are several different, but related, concepts of a Domain Model when we look at the topic in detail. These derive from how the model is used in the context of particular families of methodologies. Here are two of the more common alternative interpretations that aren't discussed further in this document.

Classes and Logical Model in Analysis model

Often the term 'Domain Model' is used to refer to the class diagram that one finds in the Analysis model. Typically, this will include classes with attributes and operations, and will have related sequence diagrams. This reflects the views of the practitioners that the classes are created specifically for use with the current system. The responsibilities are already assigned to Attributes and Operations. This may or may not be the best assignment of responsibilities to support future changes, but the sequence diagrams are used to show that this arrangement supports the current requirements as captured in the Use Cases. Typically, practitioners do not consider the conversion of responsibilities into Attributes and Operations to be design or decision. The methodology may or may not have a task to try and assign the responsibilities to the right classes. Commonly, the assignment of responsibilities to classes is in the form of defining operations on the classes, as the sequence diagrams for the system are being built.

The RUP idea

RUP describes a domain model as "an incomplete business object model, focusing on explaining products, deliverables, or events that are important to the business domain. Such a model does not include the responsibilities people carry". It is also described as an alternative to a glossary. It would appear that the responsibilities are not recorded at

all, either with the people or with the entities, unless in a very informal manner. Entities are typically described in terms of the dumb objects that are manipulated by workers in the processes used to perform the business of the enterprise.

CRC origins

On the one hand you have behaviour, on the other hand you have things; entities if you prefer that word. All systems need to relate behaviour to things. Computer programmers started by concentrating on behaviour, and bringing in the entities where they were needed. This culminated in Procedural programming. Organisation of the behaviour is the driving force; the entities follow the behaviour.

Object-Oriented programming takes a different approach, concentrating on entities (objects) first, and letting the behaviour follow the entities. However, requirements tend to arrive in terms of behaviour. These 'lumps' of behaviour will need to be broken up and divided between the entities. One of the mantras of OO programming since its early days has been "Get the Responsibilities in the Right Place". If we can do this, then building and modifying the system will be easier and success will be more likely. See the section Responsibilities in the Right Place.

Over the years, one of the most useful techniques in placing responsibilities has been Class; Responsibility; Collaboration card sessions (see Beck (1989) Wirfs-Brock (1990) Bellin (1997)). There are many variants on how to hold such a session. All benefit from the flexibility of using cards physically rather than on the computer. Each class is written on a card-index card or similar. Its responsibilities are listed briefly under the class name. Participants in a session use role-playing to describe how more complex behaviour may be produced, based on how the classes interact and use each other's capabilities.

Invariant Properties

You may well ask, if requirements tend to arrive in terms of behaviour, why don't we write our systems that way?

The domain model attempts to separate out what doesn't vary much from what does. The entities in the domain, at a fairly abstract level, haven't changed, and haven't changed their relationship to each other, for a long time, often centuries. We can assume this relationship between the entities will continue to hold through future changes. If the logical structure of our program matches this invariant structure of the domain, then the logical structure of the program is likely to be responsive to change. So if we want a program to be responsive to change, its logical structure should 'look like' the domain.

Similarly, the responsibilities held by these entities in the domain haven't changed much over long periods of time, at least at some fundamental and abstract level. Thus any system that is built around the entities, their relationships, and their responsibilities, has a logical structure at its core that is unlikely to need to change.

Variant properties, such as the architectural aspects and the usage of the system, may change.

Architectural aspects such as persistence, presentation, system management, etc, can vary in the way they are implemented, and can be separated to a greater or lesser degree from the domain invariant aspects. Where, for example, the non-functional requirements of a system change, the architectural aspects may need to change in response. Translation is based on a complete separation of architectural aspects from logical domain aspects. An architecture can in theory be applied to systems in different domains, while a single domain model can have different architectures applied to suit different operating parameters.

As the needs of the enterprise change, the uses to which a system is put will change. Where the system is structured to fit to the chunks of behaviour specified by the user, and the user's needs change, the system structure must change. Where the behaviour requested by the user is mapped on to the logical structure of the domain model, when the user wants to access different behaviour, only the mapping need change. It is possible that the user will need to access new behaviour, but this may be added to the existing structure at the appropriate places, rather than requiring a complete restructuring of the system.

Thus OO holds out the promise of flexible systems by structuring the system around those aspects that will not vary over time. Domain Modelling facilitates the identification of these invariant aspects.

How to Model the Domain: Basics

Who, What, Where, When, Why, How

Who: The Domain Modeller models the domain. He or she will be an Analyst. Different people use different names for similar roles, so any of the following roles may be appropriate to do Domain Modelling: Business Analyst; Systems Analyst, Technical Analyst; Systems Architect; OO Analyst.

What: This document covers the questions of what is Domain Modelling. This section covers the topic of what goes into a Domain Model.

Where: Chiefly in the company of the Domain Expert, but also with other Analysts. See the section Domain Expert has Left The Room.

When: Domain Modelling can be done almost any time. Most common and most useful is slightly lagging the production of Business or System Use Cases – see the later section Controlling Scope.

Why: See the sections Business to Analysis Model: Outline of Two Approaches and Advantages and Disadvantages.

How: This is chiefly covered in this section and its companion How to Model the Domain: Finer Points.

Responsibilities in the Right Place

One of the rallying cries of OO methodologies is "Get the Responsibilities in the Right Place". Not all methodologies. It is a question of what the methodology considers to be

important. OO systems are widely touted as being flexible. In practice, this may or may not be true. One of the abilities of an OO system that supports flexibility is the ability to use existing objects in different ways. So if you want to use the system in different ways, all you need to do is to persuade the system to use the objects within the system in different ways. However, there is an underlying assumption; that the objects are capable of being used in different ways. If the responsibilities are in the right place, then the objects will be capable of being used in different ways, as befits the concept that the objects represent. If the responsibilities are in the wrong place, then trying to use the system in a different way involves altering the internal plumbing so the call to exercise a responsibility ends up going to where the responsibility resides.

If flexibility of the system in response to future requests for change is not a concern, then getting the responsibilities in the right place need not be a concern. It should be possible to build the system with less effort if we can ignore this concern, but modifying the system will take much more effort.

Anecdote: One project for which I worked, and against my advice, chose to design an OO system based on a data model. They made no attempt to get the responsibilities in the right place, despite an explicit requirement from the customer that the system be flexible in response to changing requirements. After major redesigns in response to changing requirements, the plug was pulled on the project before it delivered any code at all to the customer. It had become clear that the design process being employed could not keep up with the rate of change of requirements, and never would.

Abstractions

Let us start to describe the building blocks of Domain Models.

An abstraction is a concept. It is a concept relevant to the domain being modeled. It is a concept that represents an abstract or concrete entity in the domain, rather than a relationship or constraint.

Those of you who are already familiar with OO programming may think it useful to consider an Abstraction as a Class. This is a good starting point, but take care. The Abstraction in a Domain Model includes in the single concept the ideas of the class, the class instance, and any collection of class instances. It is also a placeholder, as it were, for all the attendant helper classes and added on behaviour needed to support system architectural needs such as Persistence, Presentation, System Management, Object Life Cycle, Distribution etc. All these needs are abstracted away from the Domain Model.

Abstractions are important when attempting to get responsibilities in the right place, because Abstractions are the ‘place’ right or wrong, where the responsibilities are assigned.

Using UML to draw models of Abstractions, we represent them as we would represent Classes. Some people may choose to have a specific stereotype for Abstractions, others may rely on the fact that these ‘classes’ appear in Domain Models and so must be Abstractions.

Responsibilities : Knowledge; Behaviour

Responsibilities belong to Abstractions. Hopefully, they belong to the 'right' Abstractions; that is, hopefully we have put them in the right place. Abstractions have responsibilities of two types; Knowledge and Behaviour. That is, an Abstraction can be responsible for keeping track of certain information, and it can be responsible for doing certain things.

Let us consider an example. From the Air Traffic domain, we have an Abstraction called Track. Track is something that gathers measurements for an Aircraft. Among other Responsibilities, it has a knowledge Responsibility, it knows what Aircraft it belongs to. It also has a behaviour responsibility. For a certain time around the present time, it can say where the Aircraft was, or where it is predicted to be, at a given time.

In UML there is no direct support for the representation of Responsibilities. We have a choice of what we can do, and our choice may be restricted by the choice of modeling tool and the support it gives, or doesn't, for the alternatives.

One way is to have a specific compartment for responsibilities – possibly two special compartments, one for Knowledge and one for Behaviour. These compartments must be named. The responsibilities are listed within these compartments. I favour short sentences, one per responsibility. Others prefer to run the words of the sentence together into a NameThatRepresentsASingleResponsibility. If your modeling tool doesn't support additional compartments, you may use the 'Attribute' compartment for knowledge responsibilities and the 'Operation' compartment for behavioural responsibilities, relying on the naming convention and the knowledge that this 'class' is an Abstraction to mark these as Responsibilities and not Attributes or Operations.

Abstractions, Responsibilities and Relationships

In the earlier section, we described how Track is associated with Aircraft. Track has a Relationship with Aircraft. A line drawn between the two Associations will denote this Relationship.

This is a simplified picture. In a real Domain Model, the Track is associated with a different Abstraction, which for historical reasons is called a Target. Aircraft is a subset of Target. That means all Aircraft are Targets, but not all Targets are Aircraft. As a regular air traveler, I find this terminology somewhat disconcerting, so it is some small relief to be able to say an Aircraft, while it is on the ground, is probably not a Target. This depends on where you draw the Domain boundary. It is somewhat disconcerting to me as a regular Domain Modeller that a decision to represent Aircraft as a subset of Target could be invalidated by expanding the Domain to include surface movements. These things happen.

So how shall I show this on my Domain Model? Well, I shall say that the Target may be an Aircraft, and the Aircraft may be a Target. There's no rule against that. If I need to, I can say the Aircraft is a Target, and the Target is an Aircraft. Traditionally, Is-A relationships are implemented as inheritance. Many implementation languages would disallow multiple inheritance. Few, if any, would allow reciprocal inheritance. That's not my problem, that's somebody else's problem. When I put on my Designer hat, it

becomes my problem. Until then, I can say what I need to say, what the realities of the domain demand.

Controller Objects? No, Thank You

Why, you may ask, is the track not merely a data object, with some other object making the association between the Track and the Aircraft, and thus holding the responsibility for the knowledge? I shall give you two reasons for this, starting with the more complex and less important reason.

Such an object would be a 'Controller' object, and these are frowned upon in the OO world. Since a Track must be associated with a Target, even if the Target is not known (where do you think the term 'Unidentified Flying Object' originated?), then any Track must have an associated Controller object to associate it with its Target. Wherever the Track goes, the Controller must follow. Now if the Controller is doing multiple jobs, all its associated work is likely to follow. To avoid this, we would split up the responsibilities of the Controller, so only that responsibility related to the Track and its Target remain. Now wherever we have a Track, we have a Controller – unless somebody makes a mistake. We can ensure nobody makes a mistake by merging the Track and the Controller, so the two are now one.

This is the logic that is followed by the Robustness Diagram approach to moving from Business Model to Analysis Model. In fact there are other, better, reasons for not having Fat Controller objects that I shall not investigate here.

The second, simpler reason is that what we are saying is that Track has an associated Target, and knows about it. How that gets implemented is somebody else's problem.

Collaborations

Certain Abstractions work together (literally, collaborate) to support behaviour that requires the combination of responsibilities of more than one abstraction. For example, if we want the reported position of an Aircraft, the Aircraft may be able to supply it. However if we want the predicted position, we have assigned this responsibility to the Track. Yet the responsibility of identifying an Aircraft belongs with the Aircraft object. To provide the predicted position of a given Aircraft requires collaboration, it requires identity and position prediction. It requires responsibilities assigned to Aircraft and Track, and thus these must collaborate.

In principle, Track could go to Aircraft for identity to provide positional predictions, or Aircraft could go to Track to ask for its predicted position. For the moment we shall not say which, nor shall we say in what sequence the messages to support this behaviour occur. Later in Domain investigation we might find that Aircraft could collaborate with FlightPlan to obtain a prediction. This predicted position would have different qualities.

So these collaborations have quite different qualities than those collaboration diagrams one uses at Design, to give detail on how a set of objects collaborate to provide specific behaviour.

Unfortunately, again UML does not really support Domain Modelling, and we must use the same notation as we would use at Design time. It is important to bear in mind that what we mean when we construct the diagram is somewhat different from the meaning one would read into a Design stage collaboration diagram. In the Domain Model, we may choose to have the 'client' approach the Track for the predicted position, but this is merely an indication of a possible use of the collaboration. What is important is where an Abstraction would be able to go for help, and what type of responsibility it might access there.

Rules and Constraints

Take a simple business rule such as Double Entry Book Keeping. Here the rule is that for every amount of money debited from an account there must be an equal amount of money credited to another account, and vice-versa. If we try and assign the responsibility to an abstraction, we have a choice; we could say Account has a responsibility for knowing to which other Account Money goes, alternatively, from which it comes. Or we could say Money has a responsibility for knowing to which Account it belongs, and it is constrained to belong to one and only one at any time.

However instead of doing this, we could put a rule in the model, and associated with the relationship of Money to Accounts, without at this time assigning the responsibility for enforcing the rule to any particular abstraction. In this particular example, I would prefer using something stronger and more explicit than the cardinality of the relationship between Money and Account, though in effect we are saying, "Money belongs to precisely 1 Account".

When writing rules and constraints, I recommend always employing 2 forms, a formal definition in a language such as OCL (see Warmer (1999)), and an informal description in natural language. The formal definition gives the precision and removes ambiguity, but is inaccessible to many readers. The informal description lets these readers know what is being said, but runs the risk of being ambiguous and imprecise, or alternatively very long-winded with tedious attention to detail.

How to Model the Domain: Finer Points

Requirements Capture

While requirements capture is not the main thrust of this document, a brief overview of the situation is of use. I find it useful to consider three types of requirements. There are requirements, originating from the users and other stakeholders, that define what purpose the system is to serve, how the system is to be used. These I call usage requirements, and they are best captured in Use Cases (see Cockburn (2001)). In addition, there are non-functional requirements, all the –ilities; reliability, performance, security, recoverability and potentially many others. Finally, there are the domain requirements.

Usage requirements are typically captured in Use Cases, and feed into analysis and early design, describing the functionality that the user wants to see, and driving the production of interaction diagrams (usually sequence diagrams) that will realize these requirements. Non-functional requirements feed into the architectural phase of design, driving the

selection of certain options in preference to others. Domain requirements feed into the domain model, onto which usage requirements are mapped in the production of the interaction diagrams that are the abstract realization of these use cases.

The separation of requirements into these three types is useful, because the ideal techniques and times to capture them differ. Current wisdom at the time of writing says that Use Cases are the best way to capture all requirements, with possible separation out of non-functional requirements into a supplementary requirements specification.

I will take an example from a fairly well known source (Kruchten (2000)) to illustrate why I do not favour this approach. Here the Use Case flow of events for dispensing cash from an ATM is described. The description is pretty complete from the user's perspective. However, at no time does it say that the amount of money dispensed will be deducted from the account accessed by the user. In fact this is a typical scenario, the Use Case is very good for describing the desired interaction, but very haphazard in capturing the need for concomitant actions based on the rules of the business. The user, describing what is of interest to him, is not interested in, and possibly is completely unaware of, the business rules that apply to the situation.

The ideal way of capturing these business rules is by dialog with the domain expert, and the ideal place to capture these rules is direct into the domain model. The process of building a domain model aids in efforts to ensure all entities and relationships have been investigated in an attempt to discover the rules of the business.

Domain Expert has Left The Room

There are two approaches to Domain Modelling, I will call one the 'pure' form, and one the 'impure' form; terms for use in this section only. One of the constraints on Analysis has been stated as; analysis is something that only happens in the presence of the Domain Expert. To put it another way, if we 'discover' any facts when the domain expert is not present, when he cannot confirm they are true observations of the domain, then what we are doing is guessing, not Analysis. So the 'pure' form of Domain Modelling is based solely on facts provided by the Domain Expert. All we can do outside the presence of the Domain Expert is alter the way we present the previously elicited information in the domain model.

However, several analysts can get together in a role-playing CRC card session, to investigate the responsibilities and their distribution among the abstractions. This form is not pure analysis, as the Domain Expert is not present to confirm or gainsay any allocation of responsibilities.

In reality, the two approaches are complementary, with CRC confirming or exposing shortfalls in information supplied by the Domain Expert. Frequently the Analysts in the CRC session will know enough about the domain to be able to assign responsibilities correctly, and frequently the Domain Expert will be unable to perform the abstract reasoning that is required to assign responsibilities to abstractions. He will be more capable of describing who performs the work in the context of the enterprises with which he is familiar.

Independence of Use

One of the important aspects of a domain model that aids re-usability is that it has no inbuilt assumptions of how the entities, relationships and collaborations it describes are to be used. For this reason, it is helpful to forget any considerations of system requirements for a system under development, when performing domain modeling activities. The analyst should talk to the domain expert in abstract terms, about what these entities are, how they behave, how they relate to other entities. A model that has no inbuilt assumptions on how its modelled elements are to be used will not break when these assumptions prove false. A system built without these assumptions in its core structure will not need the core structure to change when the assumptions prove false.

Unfortunately, we tend to be paid to build specific systems, and we need to consider limiting ourselves to modeling only those parts of the domain that are relevant to the system under development. See the section on Controlling Scope where this topic is addressed.

Use Roles: Think of each abstraction in isolation

For each entity in the domain model, and for each relationship it has to the other entities, it is useful to consider what that other entity means in terms of this entity. You, for example, might be any sort of Person, but to me you are a Reader; as I, to you, am an Author. These Roles are a result of the collaboration between us that this document imparts.

The important aspect of using Roles is that the entity at the other end of the relationship can be any entity at all, as long as that entity can act in the role specified for the relationship. This means that we can now consider each entity in isolation, and its relationship to the roles with which it interacts. The entity is no longer coupled to a particular type of other entity to fill a role.

While this has little direct benefit in the production of systems, it can help the modeler gain an understanding of the domain and the entities therein. It is also of use in both the capture and the application of Analysis Patterns.

The 'feel good' factor

In each case of modeling a new domain, there comes a point at which one knows the domain well enough to be able to construct a top-level model of the domain. This will consist of say between 5 and 30 classes. You will be able to look at the model, and say, "This is what the domain is really all about". Don't try and force this point, it will come when you're ready. However, you will recognize it, because there's a certain "feel good" factor that applies, where one "just knows" that this model is "right".

This same factor can apply in collaborations and in assigning responsibilities to classes. In most cases you need to get the top level for the domain first; everything else needs to be related to its position within this context. There are certain relationships and collaborations where this constraint doesn't apply. These are usually Analysis Patterns (see Fowler (1997) for a selection of these), commonly occurring situations that one

learns to recognize. One can't guarantee that this "feel good" state will arise for all collaborations. Sometimes one needs to progress anyway, and accept that what we have is good enough.

Controlling Scope

It will normally be the case that the development of a domain model will be part of a piece of funded work. We will be modeling for a purpose, usually for the purpose of building a system to fill certain user requirements.

Let us consider the case where these requirements are presented in Use Cases. The entities mentioned in the Use Cases are a good set of entities to start as the scope of the domain model that needs to be covered. Some of these entities are likely to be synonyms, subsets etc. of other entities listed. However, to create a system that fills the usage requirements, we may need to implement concepts that aren't mentioned in the Use Cases. To understand the domain, and to be able to write a 'top-level' domain model, we may need to further broaden the scope of our enquiries. Of course, the degree of rigour both in our investigation and in our recording of the results can be cut down to a minimum in this latter case. We are modeling to understand, and once that understanding is achieved, the results may be discarded. Of course, we may choose to record the results in case this ground needs to be covered again in future.

How to Model the Domain: Esoterica

If the subtitle caught your eye, check the spelling again – this is the section most readers would want to skip.

Precondition – Postcondition Mapping

Those of you familiar with some of the more formal ways of representing behaviour may be familiar with preconditions and postconditions. For the uninitiated, these are respectively statements of what must be true (or have happened) before the behaviour occurs, and what will have happened (usually in terms of the changes that have happened) once the behaviour is complete. There is no specification apart from this of what the behaviour is, or how it takes place. For our purposes, a single mapping between one set of preconditions and one set of postconditions is known as a clause.

In its purest form, whenever the preconditions are true, the behaviour will happen and the postconditions will become true. There can be no conditional branching within the behaviour. If any such conditional branching is desired, the behaviour is represented by two or more clauses, and the guard conditions on the branches are added to the precondition set.

This rather esoteric way of documenting behaviour satisfies one of the more rigorous requirements that may be imposed on a domain model. If analysis is to be represented in its purest form, then there must be no decisions, other than representational, in the model. Splitting, or grouping, behaviour into operations is a decision, and thus can be regarded as design, not pure analysis. Of course, there will in most cases be sensible ways of grouping these mappings of precondition to postcondition such that the grouping

represents the behaviours required to fill a single 'responsibility'. However, it is wise to bear in mind that the defining of operations for a class is design, not analysis.

Model in Parallel

Modern thought in Agile methods would have modelers modeling together, sharing their ideas – frequently to the point of always modeling in pairs. The reason for this is that there are two minds applied to the problem at any time, and instant feedback is available. However, some proponents of domain modeling, including the author, make other recommendations – but *only* in this particular situation.

When possible, and at early stages of domain modelling, it is better to set two (or more) analysts to start creating a domain model independently and without communication, than to have two modelers modeling and in communication with each other. When the models are fairly advanced, bring the modelers together and review the similarities and differences in the models. Discuss the differences and try and choose the better way of treating each situation.

The rationale here is that the ideas that go into making a domain model need considerable time to mature, and to be arranged and analyzed within the modeller's mind. If we allow communication too early, the ideas have not had time to develop. In such conditions, the advantage of having a second opinion cannot immediately be achieved, because the second opinion needs time to develop. Early exposure to other opinions could poison what may have turned into a productive and useful insight.

Domain Expert Learns Domain Modelling

One of the problems of developing a domain model is in the verification of its contents. The modeller can read the model and convey its meaning to a domain expert. However, this extra translation is not efficient. One alternative is to have the domain expert learn how to read a domain model as the domain modeller creates the model based on the expertise that the domain expert is currently supplying. Thus as the domain modeller is learning the domain, the domain expert is learning to read the model. This can be useful, in that the domain expert can detect certain information is incomplete or incorrect if he can read the model. He can convey his suspicions to the modeller, who can either update the model or explain why the expert's suspicions are unfounded or the information is not relevant. Ultimately, the domain expert can look at the model and agree that to the best if his knowledge it is correct and sufficient for the purpose.

This is particularly useful, because frequently the domain expert gains new insight into the domain and its organization as a side effect of watching the domain modeller enquire, learn and organize the elicited information. The process of representing the domain knowledge requires the asking of questions that normally nobody would ask. The consideration of the question may lead the expert to consider things in a new and useful way. Unfortunately, we cannot rely on this side effect. Some experts really do know their domain 'inside out and back to front' before we start eliciting their knowledge.

Philosophy

A Domain Model is the abstraction behind what the business does and what the system does. If we start with a model for the domain, we could make it more concrete by developing it toward the logical model of a computer-based system in the domain. However, we could also make the model more concrete by developing it toward the Business Object Model for the current practices and business processes of an enterprise within the domain. Whereas in an Object-Oriented system designed for flexibility, the responsibilities are kept with the classes that implement the abstractions of the domain model, in a Business Object Model these responsibilities are 'pushed out' and assigned to the various worker roles. The resultant entities are dumb, and human workers that manipulate the entity objects enact all the smart behaviour. These entity objects are little more than data holders.

If the responsibilities in the domain model are pushed out in a different pattern, to a different set of worker roles, then in effect the domain model is supporting a Business Process Re-Engineering activity. Domain models may not be the best way to support the thinking that goes into BPR, but can be used to validate that the candidate re-engineered processes cover all the necessary responsibilities.

Advantages and Disadvantages

Reuse

The domain model contains no information pertaining to how it is to be used by any system. As a result, it is potentially capable of being used to aid the development of any system within the domain. Thus any enterprise working within the domain can reuse it for all systems (but see the section Extend don't Change) and to aid in the integration of systems of other enterprises working in the domain, either through merger or collaboration. A software house can use the domain model for any enterprises working in the domain for which it is retained to aid the specification and construction of systems. In as much as the domain model can aid Business Process Re-engineering, the model can be re-used for multiple re-engineerings.

Flexibility

The construction of systems that build their core logical structure round the invariant aspects of the domain are adapted with less effort to change of use within that domain. This capability is the basis of the much promised, but not always delivered, flexibility of OO software.

Extend don't Change

There is, of course, no guarantee that the modellers will get the model right first time. However, the common observation is that subsequent uses of the domain model may extend the model, but tend not to change what is already there. A fact recorded about the domain tends to remain a fact about the domain. Note that this is a tendency rather than a hard and fast rule. Where the model contains imperfections, these may be changed in subsequent iterations. Where the model was more sketch and less rigour, more rigour

may be added in subsequent iterations. With these exceptions, the work in subsequent iterations, for the most part, is either confirming or extending what is there.

Hypothetically, and as far as I know this hasn't been observed in practice, a domain model with seniority can contain more complete domain knowledge than any single domain expert. This observation was made for Expert Systems, whereby the accuracy of the advice given could be greater than that of any single expert, once the expertise of many experts had been encoded. The same principle applies hypothetically to domain models. However, these tend not to be honed any further once they are sufficient to fill the roles they are asked to perform.

There are often one or two small areas of any domain model where the modeller never seems perfectly happy with any alternatives. These areas may also change in subsequent iterations. However, care must be taken that one does not expend too much effort striving for perfection. 'Good Enough' is good enough.

Cost

Domain models can be constructed with almost any degree of rigour. The simplest and least rigorous may be a sketch of a few key classes and their relationships to use as an ideal to work toward, or a rough road map of the system to be developed. The most rigorous and detailed is that developed in preparation for the application of a Translation Engine. The cost of the sketch is minimal, assuming the work done to obtain the knowledge that would be needed to create the sketch would have been done anyway.

In general, it is accepted that the cost of producing a relatively formal domain model 'up front' is rarely recouped on the costs of development of a single system. However, of all the documents that may aid those that come after, a domain model is the most generally useful. A sensible approach if one is unsure that the model will be re-used is to capture the information relatively informally the first time the model is used, and invest more effort in subsequent times of re-use to extend and add more rigour and formality to what you get as a beneficiary of the earlier work.

Summary

Domain modelling should be of interest both to System modellers and Business modellers. In many cases, a domain model that is merely a sketch of the key concepts within the domain will be sufficient to clarify and convey ideas. There are few if any times when there is no gain from this effort. There are circumstances where more effort may be adequately rewarded. The criteria for making such a decision cannot fully be defined within the scope of this document, yet it is hoped that sufficient pointers have been given to allow the reader to reach his or her own conclusions.

Acknowledgements and References

Few if any of the ideas in this document are original; this is a collection primarily of other peoples' ideas. Only the major contributors of these are listed in the references. The origin of some of the ideas can probably never be known. It may be that this is the first time they have all been pulled together – if so, that's purely coincidence, and down

to the repeated promptings of Sue Burk of MassMutual and Hüseyin Angay, fellow collaborator with the Appropriate Process Movement.

This document is revised as necessary; comments are welcome. Thanks to the initial reviewers:

Thanks also to the following, who provided comments that improved the document:

References

Beck (1989)

Beck, K. and Cunningham, W. (1989). *A Laboratory for Teaching Object-Oriented Thinking*. Proc OOPSLA '89, pp 1-6

Bellin (1997)

David Bellin and Susan Suchman Simone (1997). *The CRC Card Book*, Addison-Wesley Object Technology Series

Cockburn (2001)

Alistair Cockburn (2001). *Writing Effective Use Cases*, Addison-Wesley Agile Software Development Series

Fowler (1997)

Martin Fowler (1997). *Analysis Patterns: Reusable Object Models*, Addison-Wesley Object Technology Series

Kruchten (2000)

Philippe Kruchten (2000). *The Rational Unified Process An Introduction Second Edition*, Addison-Wesley Object Technology Series

Warmer (1999)

Jos Warmer and Anneke Kleppe (1999) *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Object Technology Series

Wirfs-Brock (1990)

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