UML for Managers

Jason Gorman

Chapter 4

March 6, 2005

© Jason Gorman 2005

© Jason Gorman 2005

Applying UML Part II

People often make the mistake of thinking that UML is only useful for specifying or documenting software architecture. This is understandable, since software design has been the main thrust of companies and educators working with UML over the last 8 years.

However, an information system isn't necessarily something that lives on a computer. As much work goes on in the average organization as a result of *people* creating and manipulating information as goes on inside their computers.

Business architecture is arguably more critical than software architecture, and yet many organizations that invest time and money on software modeling neglect the design of the business itself. How many of us have suffered at the hands of poor customer service, only to discover that the culprit has invested heavily in Customer Relationship Management (CRM) software?

Software by itself rarely solves business problems: people using software – now that's a different story. Business analysts and enterprise architects should concern themselves with the way things are done in a business – whether it involves software or not. All too often, they concern themselves with software requirements and software architecture at the expense of understanding how the software will actually be used in real business processes. Failure to understand the problem will almost certainly lead you to the wrong solution.

In this chapter, we'll look at how UML can be used to help us describe business architecture, and how these business models can be effectively mapped on to software specifications to better ensure that the code you write is the *right* code for your business.

We will also look at enterprise architecture – an overloaded term that means many things to many different people – and try to clear up some of the confusion about what it is and why we need it.

Business Modeling in UML

There are 4 aspects of our business that we can model using UML:

- Business goals
- Business processes
- Business structure
- Business rules

These 4 faces of business modeling link very closely together. Goals apply to processes. Processes create and manipulate business objects. Rules apply to business objects and business processes.

© Jason Gorman 2005

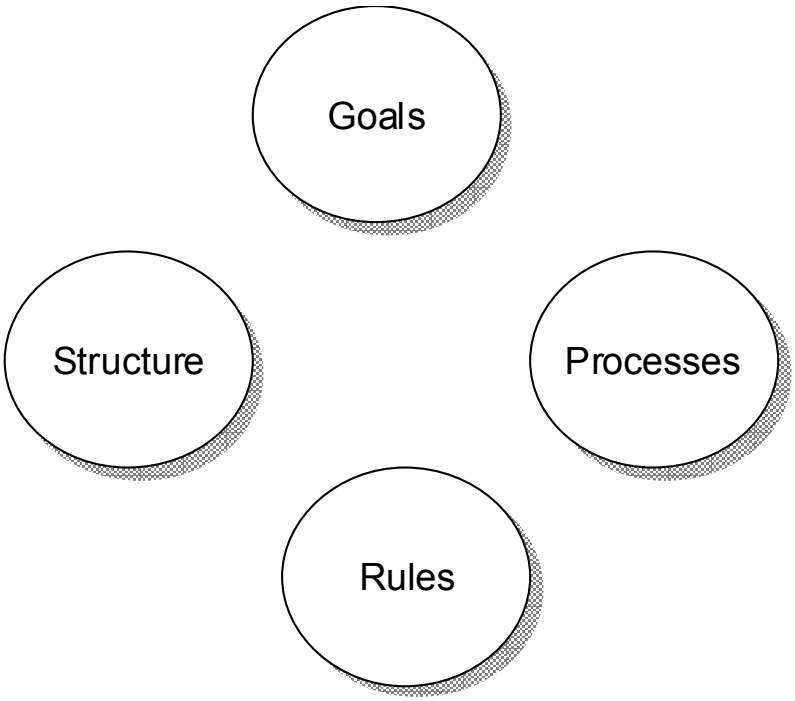


Fig 4.1. The 4 faces of business modeling in UML

In UML, business goals can be thought of as objects – instances of the type Goal. Goals can be measurable – which we call *quantitative goals* – or immeasurable – which we call *qualitative goals*.

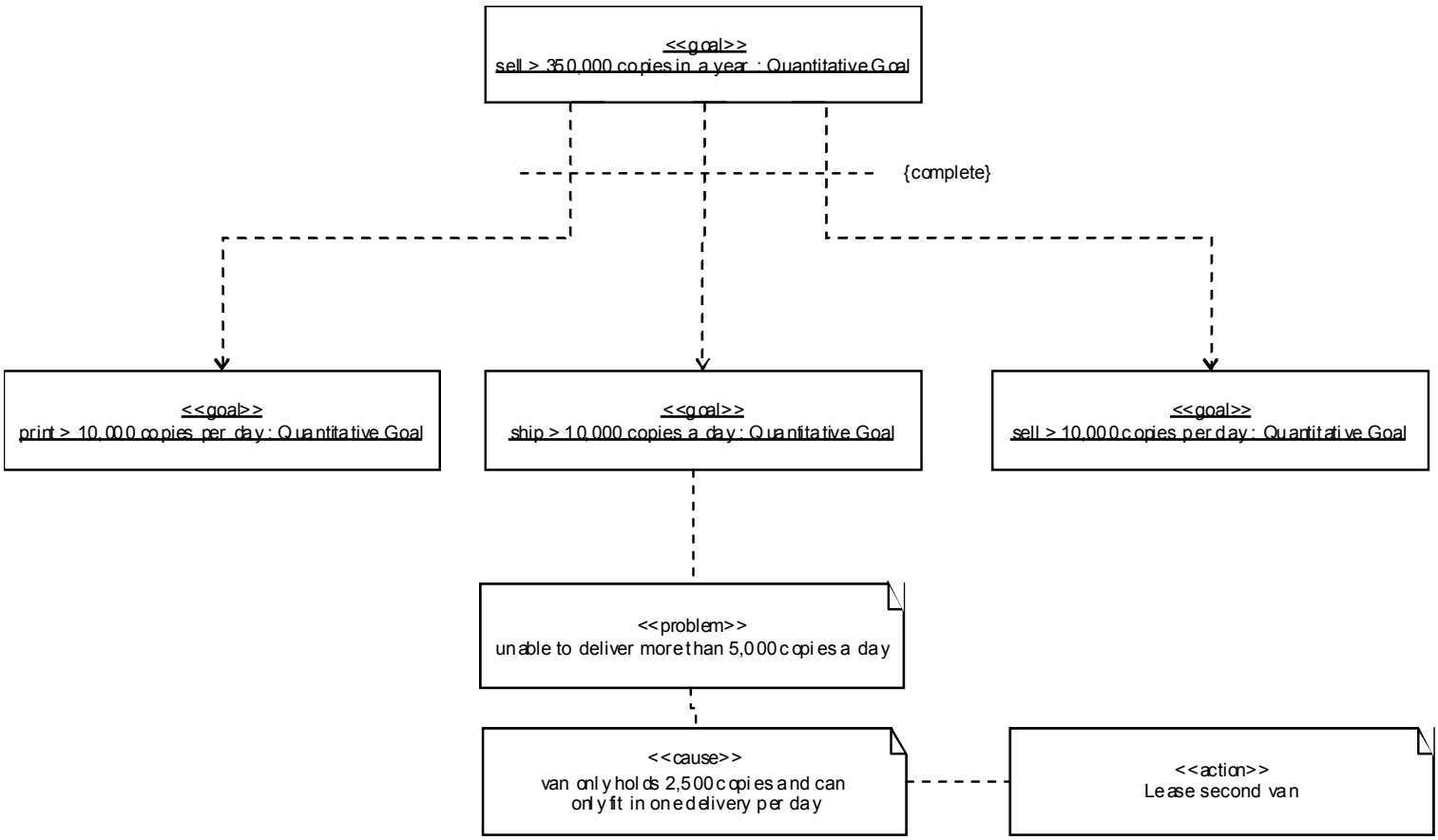


Fig 4.2. Business goals can be modeled as objects

© Jason Gorman 2005

In this example, we are using UML extensions for business modeling created by Eriksson and Penker[1]. You'll notice the use of stereotypes like <<goal>> and <<problem>> to add extra information to the object diagram that's specific to business modeling.

Goals can be composed of other goals. This tells us that, in order to achieve one goal, we must achieve other goals. For example, in order to sell more than 350,000 copies of our book each year, we have to print, ship and sell more than 10,000 copies a day. The constraint {complete} tells us that if we satisfy the three lower goals, we will have completely satisfied the higher goal.

We can attach stereotyped notes to goals to describe the problems we face in achieving them, as well as actions for overcoming those barriers.

Using this notation, we are able to describe complex, multi-level business strategies. This provides a great starting point for a strategic program of business change, because now we have a clearer idea of what we're setting out to achieve and what we need to change to achieve it.

Next, a business analyst might like to look at the business processes that will have to change if the goals are to be met.

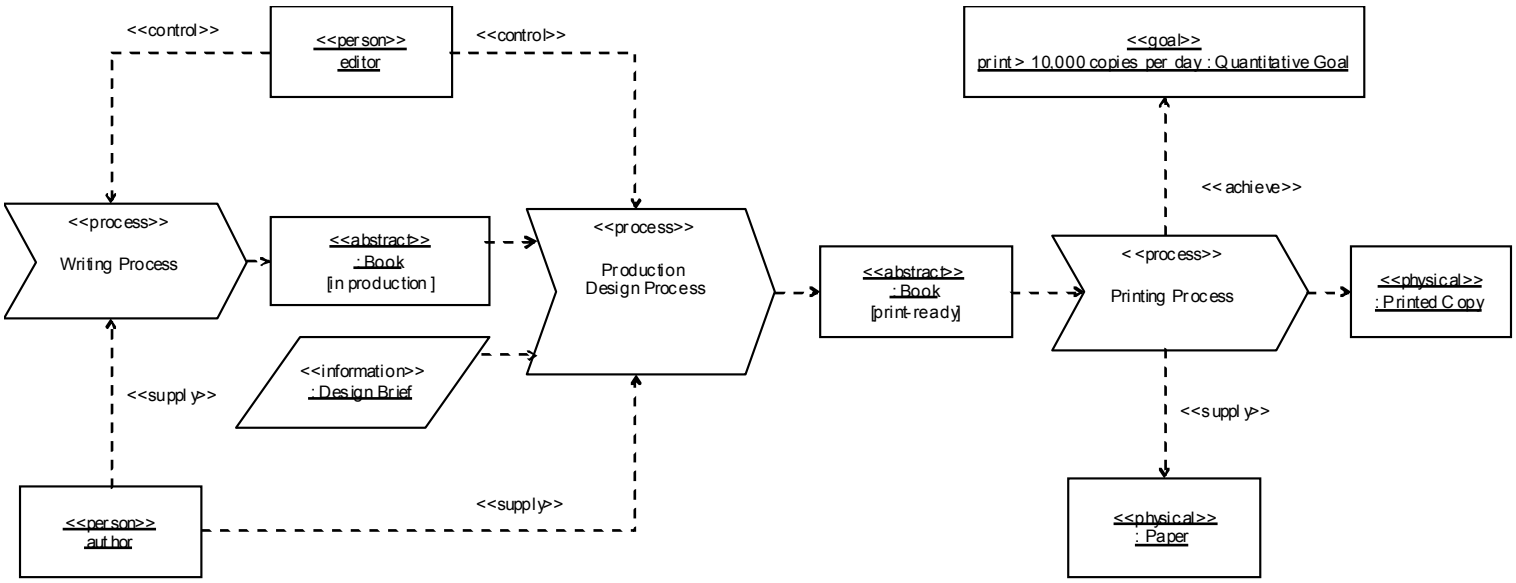


Fig 4.4. Business process models show how processes impact business goals, as well as the people, resources and information involved in those processes. It also helps to visualize at a high level how business processes fit together.

Again, we're using Eriksson-Penker UML extensions to add extra information to our model that's specific to business architecture.

What this example doesn't show is the detail inside each business process. We can use UML activity diagrams to model processes in more detail, but it helps enormously to have a bird's eye view to begin with – especially as business processes can be very complex and most organizations have many business processes.

© Jason Gorman 2005

In this example, we can see clearly that our program of change should focus on the printing process. The next step would be to explore that process in much more detail. At this point, many organisations make the mistake of using interviews and meeting rooms to help them learn about a business process. This is nonsense of course! Would you get on a plane that's being flown by someone who interviewed some pilots and drew a few process diagrams? Your analysts should go and see the processes for themselves and collect rich data (documents, databases, photos, videos etc) to help them fully understand what's going on. They can then bring this data back to the office and examine it at their leisure. The resulting models will be much more accurate and therefore useful.

Of course, if the aim is to improve a process, your analysts will need to work with the experts – the people who do that job day-in and day-out – to define new ways of working. As much as UML can help in gaining an overview of this, ultimately the end product is not a UML model, and you must always bear this in mind.

In understanding business processes, it's also very helpful to understand the objects that are involved and how they are related.

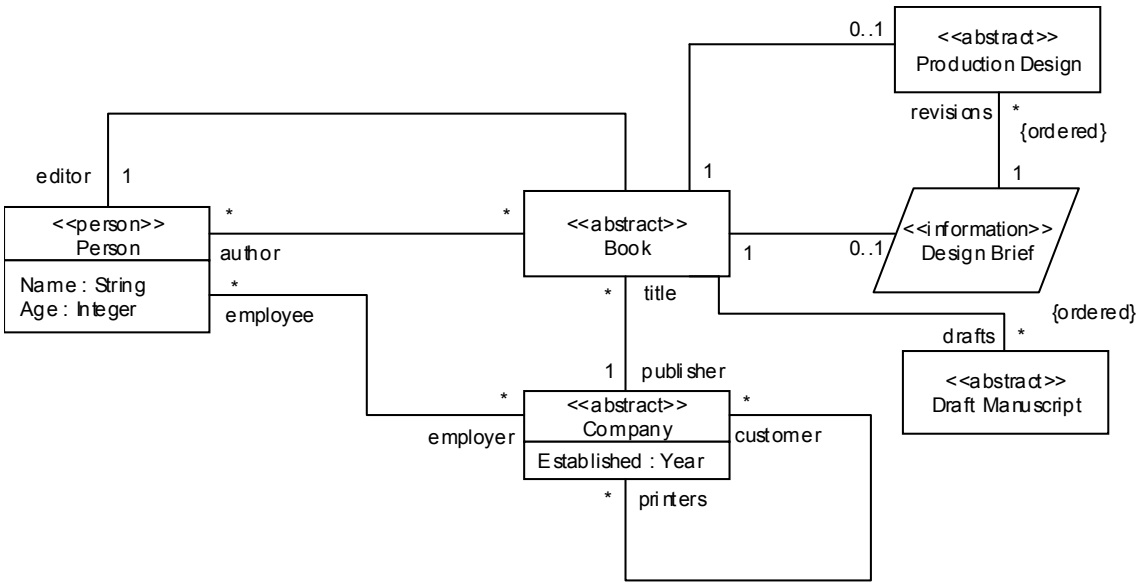


Fig 4.5. A business structure model describes the types of objects involved in business processes.

You will hear these kinds of models referred to in several different ways, including *business object models* and *business domain models*. Simply, they are models of the types of things involved in business processes and the relationships between them. I'm sure you can see how useful this model would be if we were ever asked to build software to help in the publishing process.

Finally, we need some way of describing the rules that apply to our business objects and business processes. In a previous chapter, we already saw how UML models can be extended using constraints. This is exactly how we should model business rules. Ideally, our rules should be clear and unambiguous – particularly if we're thinking about building software in the future.

© Jason Gorman 2005

The Object Constraint Language provides us with a precise language for defining rules that apply to our models. The irony is that, while OCL has most value in business modeling, most business analysts find learning OCL very hard. To be effective with OCL, you need to think like a programmer and have an innate understanding of logic and object oriented programming. Many analysts come from business backgrounds and are ill-equipped for precise object oriented modeling using OCL.

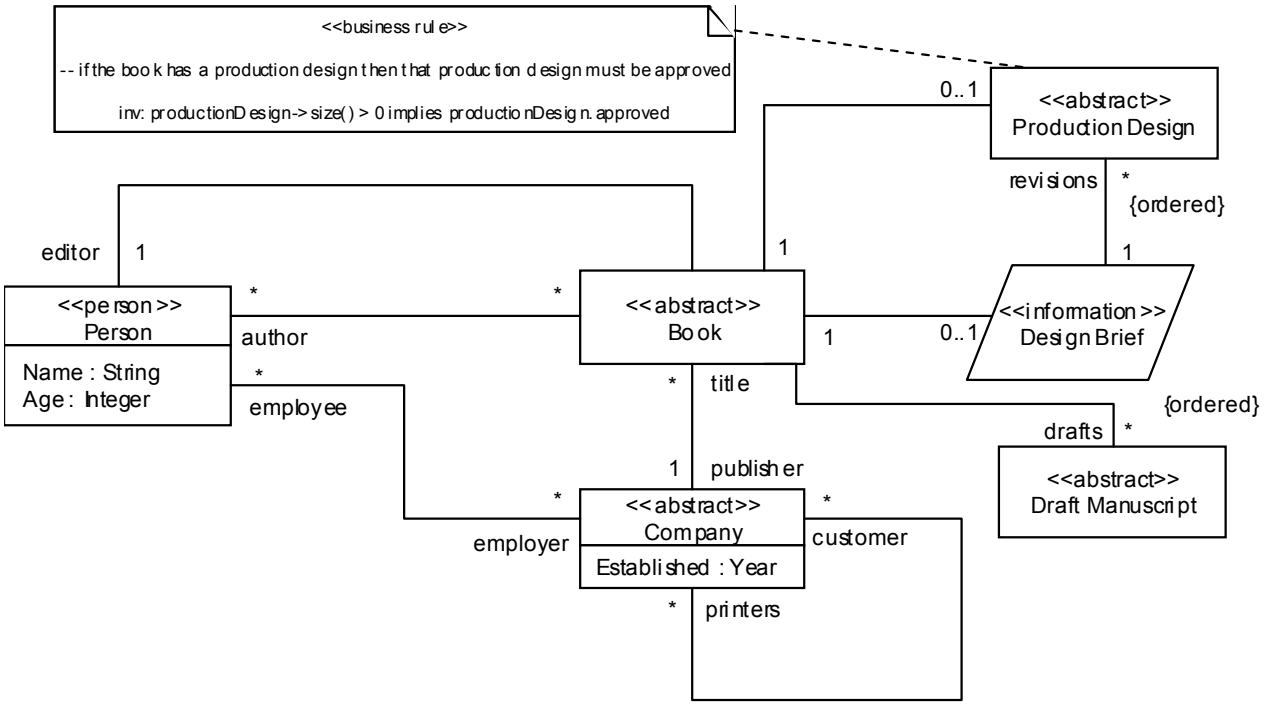


Fig 5.6. The same business object model with a constraint.

With the Eriksson-Penker extensions, we can use UML to describe the 4 faces of our business architecture and effectively exploit modelling in strategic business change programs.

However, experience has shown that there is one important aspect of business architecture missing from their interpretation. More and more these days, businesses want to express a balanced view of their strategic goals. The modern enterprise cannot succeed just by pursuing a narrow set of financial goals. They must also address the needs of employees, customers, the local community, the environment, the government and many more interested parties who are impacted by what businesses do.

The *Balanced Scorecard* is a tool for grouping the goals of multiple stakeholders into *perspectives*. Commonly, a scorecard has at least 4 such perspectives:

- Financial
- Internal Processes
- Customer
- Learning & Growth

© Jason Gorman 2005

Many companies create customised scorecards to suit their specific needs, adding extra perspectives for employees, the environment and so on. Using a balanced scorecard, we examine the goals of multiple stakeholders in our business and show how they impact each other.

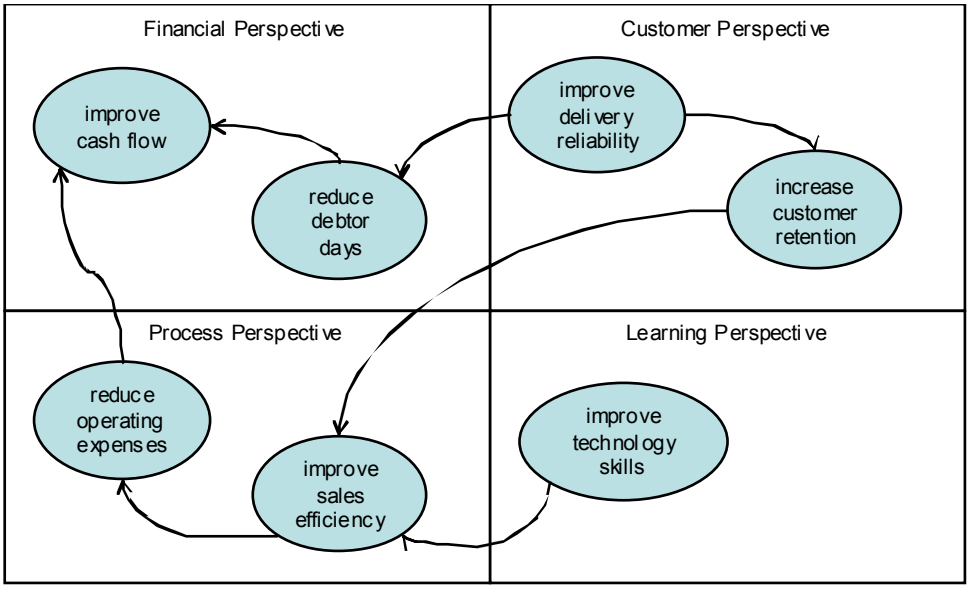


Fig 5.7. A balanced scorecard shows the goals of multiple stakeholders and how they might be related.

Next, they design performance measures – formulas that describe how they'll know whether a goal is being met or not – and set targets for improvement. Many large organisations employ what are called "digital dashboards" – simple user interfaces that managers can see at a glance what the value of these measures are in their business at any point in time. They then use their understanding of how these measures are related to "steer" the enterprise. (eg, increasing employee bonuses to boost production).

It is unlikely that our theories about how different measures relate to each other will be spot on at the start. We should learn from experience and refine our scorecard with each new lesson. Over time, the aim is to build up an accurate high-level picture of the "levers" in our business and what the effect of pulling each lever will be.

© Jason Gorman 2005

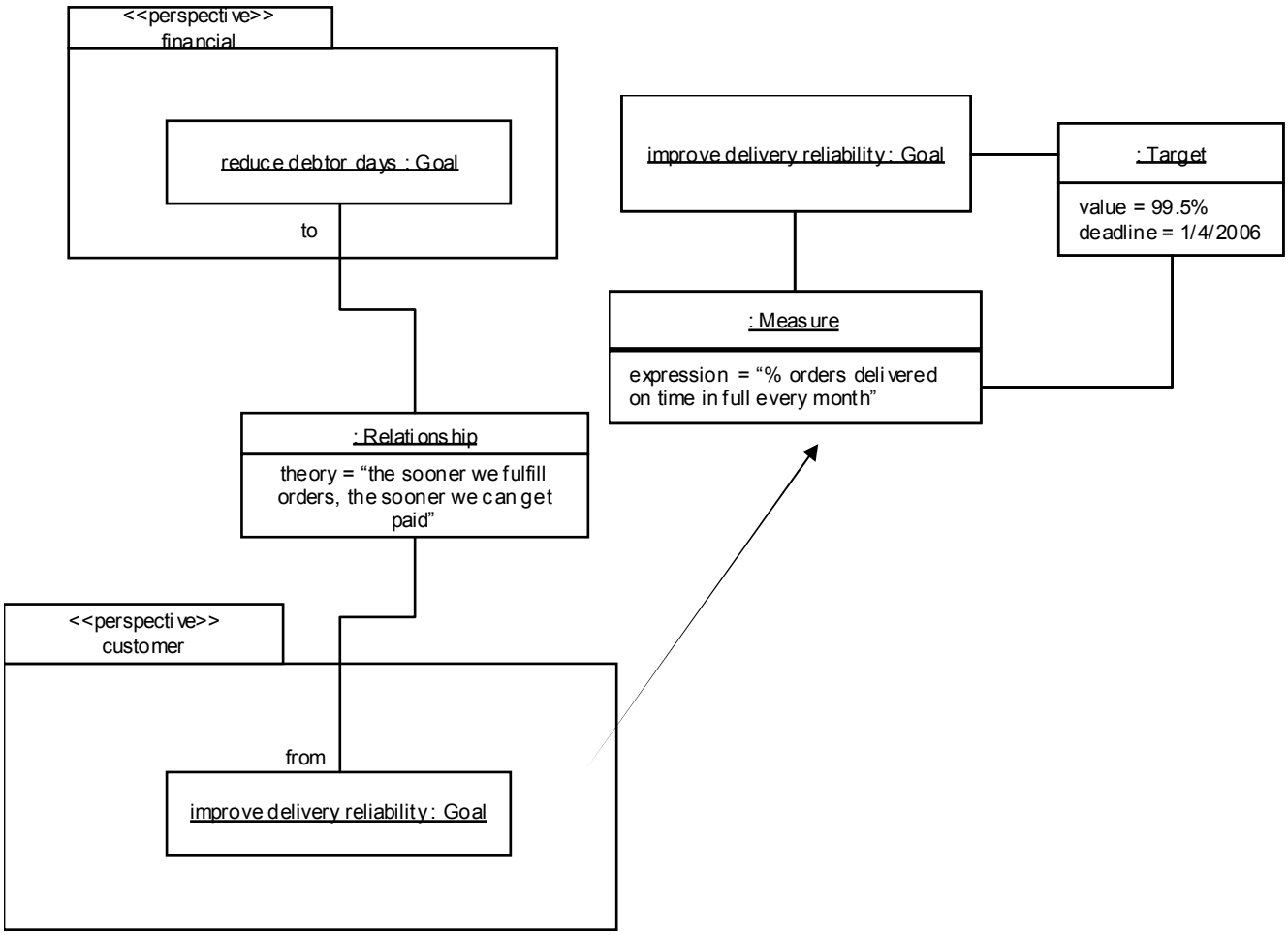
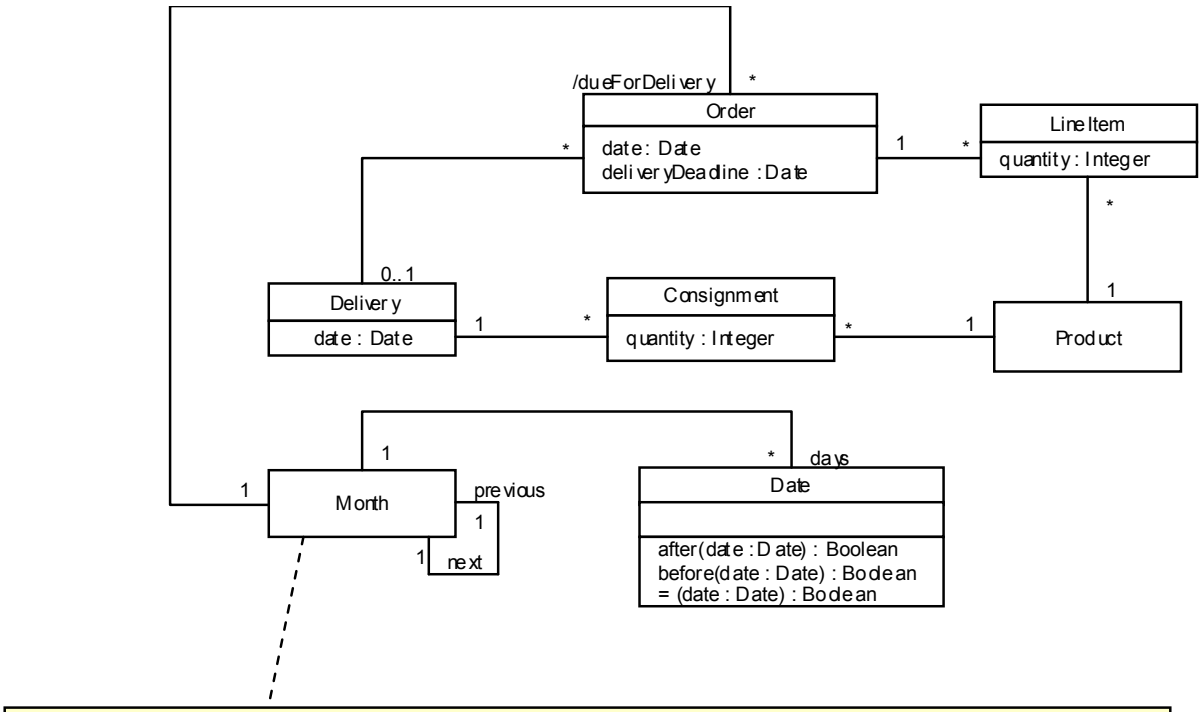


Fig 5.8. Using UML packages and associated extensions to model a balanced scorecard

My own contribution to business modelling with UML is in the modelling of these scorecards, as well as the precise modelling of performance measures that go with them.

© Jason Gorman 2005




```
{dueForDelivery = Order.allInstances->select(days->contains(deliveryDeadline))}

{deliveredComplete = dueForDelivery->select(delivery.consignment->forAll(c : Consignment|
                                                    lineItem->exists(quantity = c.quantity and product = c.product))}

{deliveredCompleteOnTime = deliveredOnTime->asSet()->intersection(deliveredComplete)}

{percentDeliveredCompleteAndOnTime = deliveredCompleteOnTime->size() / dueForDelivery->size()}
```

Fig 5.9. OCL constraints can be applied to a business object model to precisely define performance measures relating to business goals

The advantages of modelling performance measures in precise UML are threefold:

1. **Unambiguous measures are easier to test** – we can use snapshots and filmstrips to check that our measures work for a range of business scenarios. This is particularly important because organisations have a tendency to play "games" with poorly designed performance measures. For example, one airline famously introduced a measure to help them improve the turnaround time on baggage handling, which relied on staff getting luggage onto a conveyor belt as quickly as possible. They were not measuring the time it took for the passengers' to actually receive their baggage, so staff were literally throwing bags off the plane and on to the conveyor, and then leaving them there! An unambiguous specification of the measure would have helped highlight such opportunities to "play the system" without actually delivering performance improvements.
2. **Measurement systems specified in unambiguous UML are easier to implement using software** – we can see a direct route from our UML specifications to working systems. Many – if not most – performance measurement programs result in some kind of software system for recording and reporting the measures, so of all our business change programs, we should naturally expect to end up building software after a scorecard has been agreed.

© Jason Gorman 2005

There are many commercial off-the-shelf software solutions for reporting scorecards, but we still have to write the code that implements each measure.

3. **Measures specified in unambiguous UML are difficult to misinterpret** – quite often in large organisations, different offices/branches will interpret the same measures differently. So baggage handling efficiency for our airline at JFK airport in New York may well be measured differently at London Heathrow. If our baggage handlers in London use the "time to conveyor belt" system, but our baggage handlers in New York use a "time to get to passenger" system, it's quite possible that London could score higher with worse actual business performance. Management may decide to transfer business practices from London to New York in a misguided attempt to improve performance at JFK. Inconsistency in performance measurement can produce misleading results and cause managers to "steer" their strategy based on false intelligence.

Most valuable for me is that specifying business goals and associated performance measures in UML allows us to tie them closely to our models of business processes, business structure and business rules, giving a much more complete and consistent picture. If the same types of objects involved in business processes and business rules are applied to business performance measures, we can build a *truly unified picture of the business architecture* – from strategy right down to the design of database tables – using the same modelling language.

In many senses, we should view UML as a business modelling language just as much as we view it as a software modelling language. Indeed, the further up we go from software, the more valuable our models become.

Enterprise Architecture

I class the kinds of models we've discussed in this chapter as *enterprise models*. They form the basis of *enterprise architecture*. There's a lot of noise being made in the IT industry at the moment about enterprise architecture. It seems every man and his dog has something to say about it.

Some people define enterprise architecture as the architecture of enterprise software systems and how they fit together to execute business processes that span multiple systems and potentially multiple organisations.

We distinguish enterprise software from other kinds of software – like desktop applications, for example – because they share a set of similar characteristics.

The typical characteristics of this interpretation of "enterprise architecture" are systems that are:

- **Multi-layered** – the key responsibilities of accepting user input, displaying data, co-coordinating transactions and processes, modeling business data, enforcing business rules, and storing and retrieving that data (this is an oversimplification, you understand) are packaged up so that they can exist as

© Jason Gorman 2005

independently of each other as possible. Why take the time to do this? Because an important goal of architecture is to make software easier to change, reuse and extend. The less dependencies there are between different kinds of logic (display, process, business), the easier it is to change one without impacting the others. Many of these patterns focus on separating architectural concerns.

- **High-volume** – typically enterprise systems must handle thousands of simultaneous users and deal with gigabytes or terabytes of business data. Traditional object oriented architectures, the kind we used when we were mainly building single-user desktop applications, fall apart under the strain of these huge volumes. We want to retain some semblance of object orientation – largely because it's always good to stick close to the problem to build a comprehensible solution – but we have to be very creative about how we use resources like memory, database connections, threads, and so on.
- **Component-based** – back in the day, enterprise applications ran on very, very big centralized computers called "mainframes". They were written in old languages like COBOL and Assembler. They were largely batch processors that shifted and sorted through huge amounts of data very efficiently and quickly. The code, however, was an absolute nightmare to change. Old technologies like COBOL suffer from a lack of modularity, which leads to large amounts of duplication in the code. A simple change to the structure of a record might require changes to hundreds or thousands of modules that depend on that structure. The edict of "do everything once and in one place only" is only possible with an underlying technology that allows us to package and reuse the logic effectively. Component technologies like COM, CORBA, Java and .NET make it easier to isolate logic and to reduce the dependencies between different parts of the system.
- **Persistent** - although there are several key kinds of enterprise application, they all have one thing in common. They need to store data somewhere so it can be retrieved, manipulated and analyzed perhaps years or decades later. If your business objects can be stored and later retrieved – even after the computer has been switched off – they are said to be *persistent*. A bank cannot afford to lose data, so these data stores have to be extremely robust and resilient. They also have to be capable of handling enormous amounts of data, and enormous numbers of database transactions. The most mature database technology available is relational. IBM, Oracle, Sybase, Microsoft and many others, have mature relational database products that are the engine of choice for the vast majority of business applications. It's just a shame that object and component technologies like Java and .NET don't quite fit with the relational model – which is very outdated and constraining. Solutions exist for mapping objects onto relational databases, and some database vendors have strong offerings in the specialized object-relational (or sometimes "post-relational") database market space. There are also pure object databases, which cut out much of the hassle with mapping objects onto relational databases, but they are not widely used in business applications and as a result carry a high price tag.
- **Transactional** – as well as being able to handle large amounts of users and large amounts of data, enterprise applications have to ensure that data never gets into an invalid state. Imagine a funds transfer from one bank account to another failing half way through. The amount is debited from one account, but

© Jason Gorman 2005

never credited to the other. We would wish this process to either be completed 100% or not at all. A process that must either be 100% complete or not at all is usually called a *transaction*. The vast majority of enterprise applications will have processes that are to some degree transactional. Managing transactions also requires that you design your software a certain way so that changes to business data aren't committed to the data store until the transaction is complete.

- **Concurrent** – the data stored in an enterprise application could be used in multiple transactions at the same time. While one user is transferring funds from one account to another, another might be withdrawing cash for the same payer account from an ATM. The first user might start out thinking they have sufficient funds to cover the transfer, but before the transfer is complete the account may have been emptied by the second user. Concurrent applications need to implement policies for handling these scenarios – not an easy task when you consider the explosion of possibilities with just one account and two concurrent users. When you have millions of accounts and millions of users, concurrency needs some serious thought.
- **Distributed** – typically, enterprise application code doesn't all sit on the same computer. You may need multiple computers to handle the processing load. You may need multiple technology platforms for the same application – perhaps putting your J2EE business logic on an AS/400 mid-range computer, and your ASP.NET web front end on a cluster of Windows servers. Distributing logic and data has many consequences for the design of an application. Distributed, concurrent applications are an order of magnitude more complex. The most significant impact of distributing your logic is that each node in the distributed application has to somehow share common data with the other nodes.
- **Heterogeneous** – enterprise applications typically involve multiple technologies and platforms. To build a web application, developers might use a programming language like Java, but may also need to use HTML, JavaScript, SQL, XML, XSLT, WAP/WML, COM+, CORBA, and so on. Gone are the days when business applications systems could be written entirely in C.

For the application architect, the leap from single-user desktop applications to multi-layered, high-volume, persistent, transactional, concurrent and distributed applications is as big as the leap from programming in COBOL to programming Java. It's a whole new kettle of fish!

But enterprise applications are becoming the norm. Most software being built these days has enterprise characteristics, and the skills of the majority of architects and developers are just beginning to catch up. Hence, there is a high demand for people with knowledge and experience of "enterprise architecture". When a skill set is on the ascendant, we tend to see the market going into hyper-drive. "Enterprise architecture", in this sense, is arguably nearing the zenith of this curve.

Meanwhile, a new interpretation of "enterprise architecture" is starting to gain momentum. There are those of us who have long seen it as the "architecture of the enterprise" rather than just design patterns for high-volume software applications. Though software is undeniably a significant factor in the running of a business these

© Jason Gorman 2005

days, it's by no means the be-all and end-all. In most businesses, people are a far more significant factor. People use software to do their jobs. It is the "doing of jobs" that interests me far more than the design of the software being used to do them – if software is being used at all (and in many business processes, software still plays no part).

The new "enterprise architects" are concerned with the *business information model* – with or without software. An information system does not have to live on a computer. In our daily lives we create, manipulate, analyze and exchange information all the time. It's what our brains have evolved to do, and it's why we have such rich and sophisticated languages.

Understanding how information is used in a business allows us to exploit many of the skills and tools we've built up for understanding information on computers. After all, all computing concepts had their genesis in real-world abstractions. Computer science is a branch of mathematics, and logic doesn't need silicon chips to exist.

The Unified Modeling Language is built on logical principles. At its most fundamental, it allows us to model any kind of information, as well as the processes and rules that apply to that information.

As an "enterprise architect" – one who has progressed from design patterns for enterprise software applications to the actual informational design of the business itself – I use UML more to help me visualize and communicate business ideas than I do to visualize software. This is where UML is at its most powerful – when we're *just* talking about information and logic.

I see *modeling* as the key component of enterprise architecture – not software or design patterns. "Enterprise architecture patterns" for me are about what model we should apply to a specific problem. When I look at the famous Zachman Framework for Enterprise Architecture, I think about what models would fit into each box and about how I can join all those models together to give a more complete picture of the business.

The Zachman Framework (overleaf) is a simple tool for visualizing how the different pieces of enterprise architecture fit together, and provides us with a conceptual framework for putting our software and systems into their correct business context.

Many business change programs use the framework – or one on the similar frameworks for enterprise architecture available today – as a touchstone for co-coordinating their efforts. You will see that UML can be exploited in the middle three layers of the Zachman Framework, although most modelers and modeling tool vendors focus on the 3rd and 4th layers, largely because they approach enterprise architecture from a software-centric point of view. While they acknowledge the need for the 1st and 2nd layers in any sufficiently complete picture, they, perhaps misguidedly, leave those aspects of enterprise architecture to non-modelers – which often creates the disjoint between business and IT strategy that many organisations suffer from. Used wisely, UML can help bridge that gap by providing a single, unified language for describing the strategic aspects of enterprise architecture, as well as the system/software aspects.

© Jason Gorman 2005

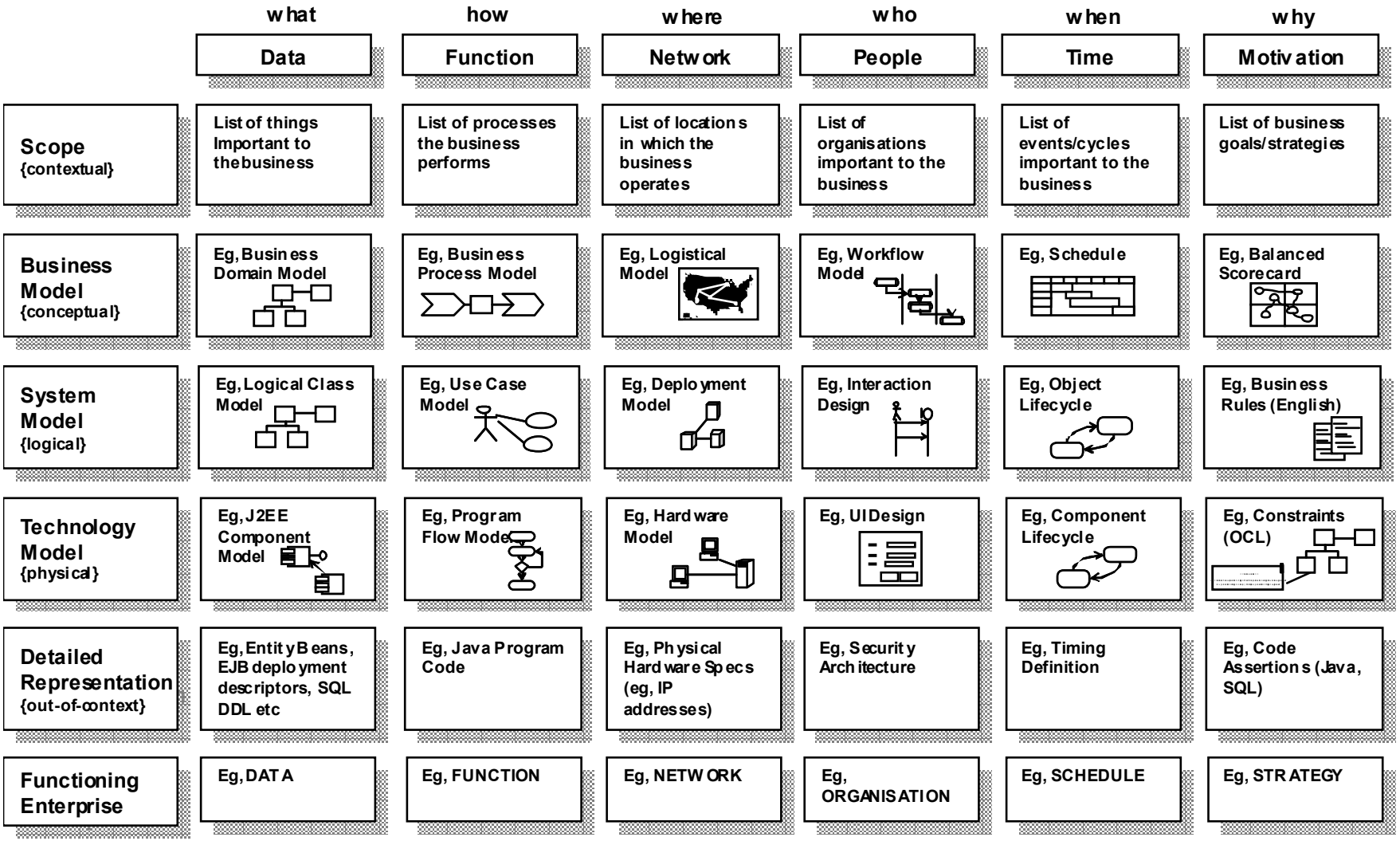


Fig 5.6. The Zachman Framework for Enterprise Architecture.

© Jason Gorman 2005

Enterprise Traceability & Model-driven Architecture

Many IT and business managers find the notion of traceability between models and other – perhaps lower-level - models, and traceability between models and their IT implementations, very attractive.

The goal of Model-driven Architecture is to provide a seamless and largely automated path from high-level business-centric models to working software systems. The theory is that all the logic necessary to understand business applications is captured in abstract, technology-independent models, and all the knowledge about how to implement that business logic in, say, J2EE or .NET is defined as automated transformations that take the abstract models and generate working code from them.

The promise of MDA is a much quicker and smarter route from business logic to working software – so, theoretically, business change should be quicker and cheaper. The reality of MDA today falls far short of this promise, despite what you may have heard. We will look at MDA, and other model-driven software development processes, in the next chapter.

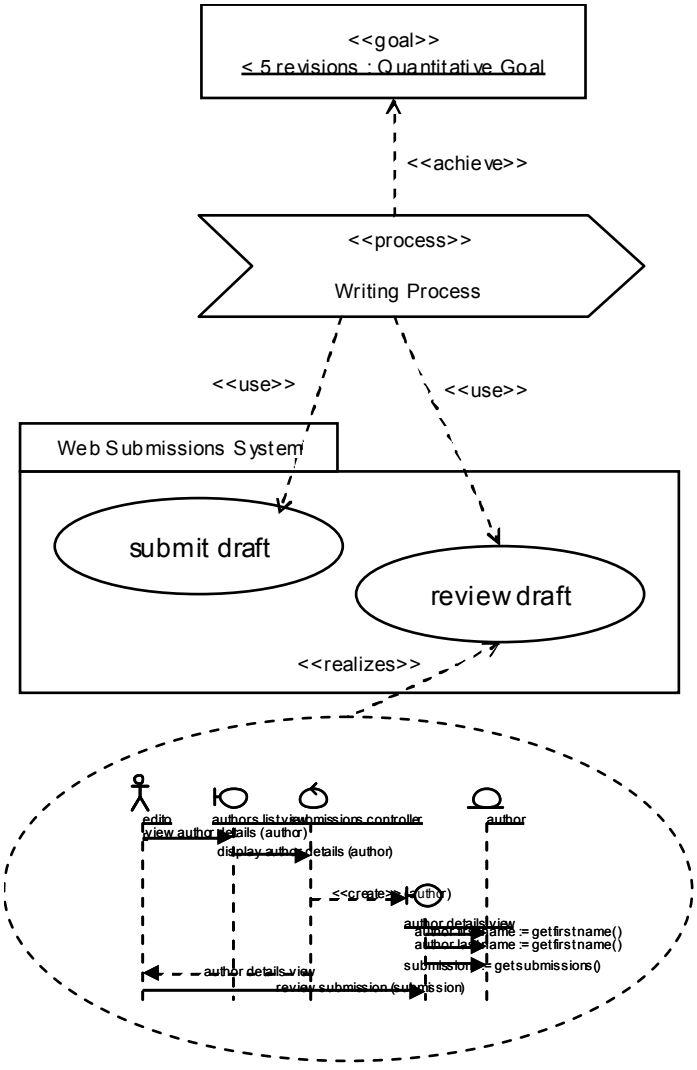


Fig 5.7. Traceability between different aspects of enterprise architecture

© Jason Gorman 2005

That said, it can be very useful to understand the relationships between models and other models, and models and software, to help manage the enterprise architecture process. Knowing how business goals map on to business processes helps us identify what processes might need to change if we change the goals. Understanding how business processes map on to software usage scenarios (use cases), helps us identify what software features may need to change if we change the business processes in which they're used. Understanding what Java components play a part in a usage scenario helps us identify what code may need to change if we change the use cases.

Traceability can help us to get a feel for the potential impact of change at any level in our enterprise architecture – which is why it is so attractive to many managers.

Some modeling tools provide what is called a traceability matrix that allows people to define the dependencies between different aspects of their models and show these in a table that maps one aspect of the model on to another (eg, use case scenarios on to design classes). The aim of these tools is to allow managers to predict the impact of changing one aspect of the model on other aspects that depend on it, in order to help them plan for change more accurately.

	Analysis Class				
Use Case	Person	Book	DraftManuscript	DesignBrief	ProductionDesign
submit draft	x	x	x		
review draft	x	x	x		
update draft status	x	x	x		
view submission status	x	x	x		
submit production design	x	x			x
review production design	x	x		x	x

Fig 5.8. A traceability matrix shows dependencies between model elements

I would warn you not to get too excited, though. Model traceability is not an exact science, and estimates based on traceability matrices tend to be no more accurate than estimates based on the developer's gut instinct. Most tools require the dependencies between model elements to be maintained by hand, and it's an expensive and very hit-and-miss business.

My advice is to maintain traceability between only the highest-level elements – business goals and business processes, business processes and use cases – and to draw your estimates from experience and the instincts of your senior developers.

© Jason Gorman 2005

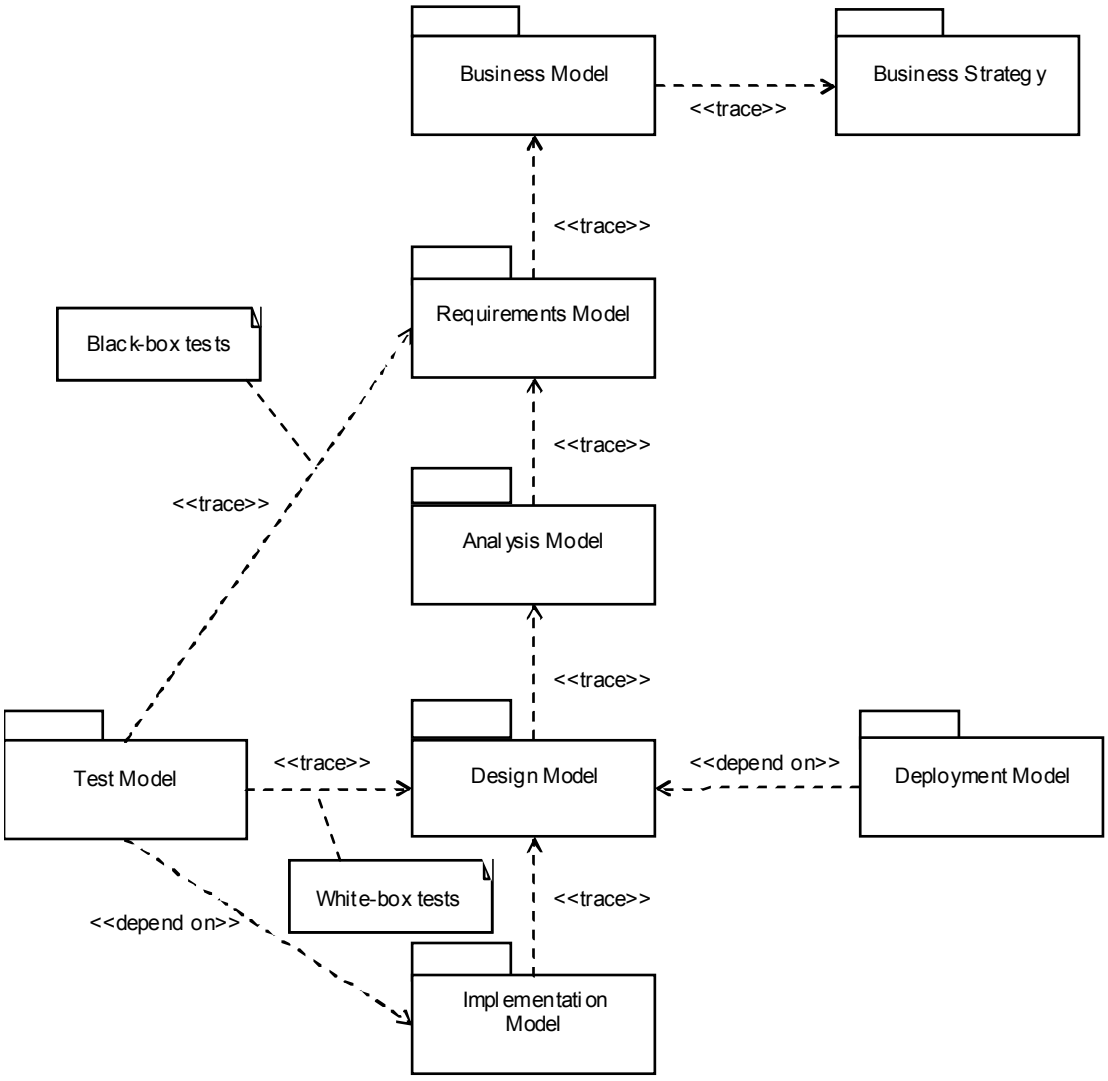


Fig 5.9. Traceability between different aspects of enterprise architecture (high-level view)

© Jason Gorman 2005

Further Reading

Business Modeling with UML – Eriksson & Penker

http://www.amazon.com/exec/obidos/ASIN/0471295515/002-8032971-2157620

The Balanced Scorecard

http://www.balancedscorecard.org

The Zachman Framework

http://www.zifa.com

© Jason Gorman 2005