FEABHAS
EMBEDDING SOFTWARE COMPETENCE

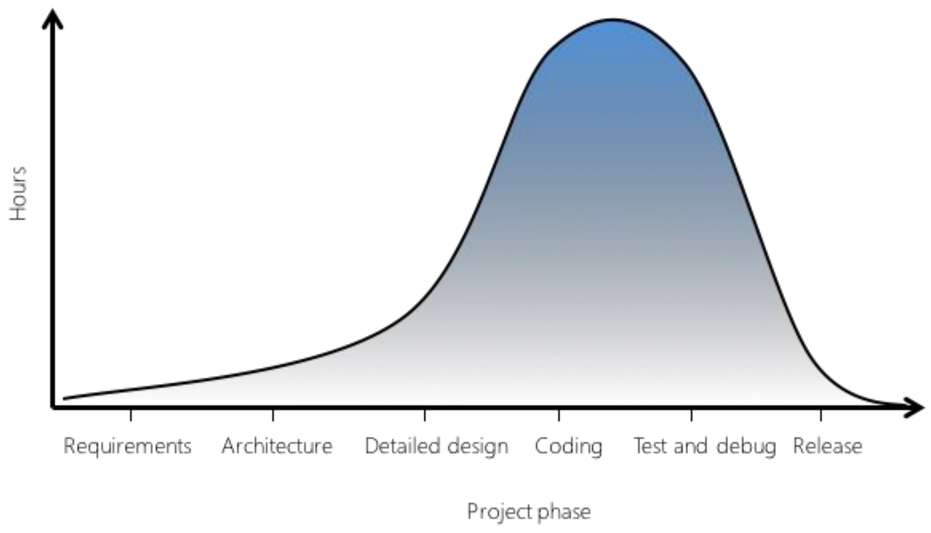
An overview of software modelling with UML

Why do we model?

Engineers are constrained by two conflicting factors of modern systems development:

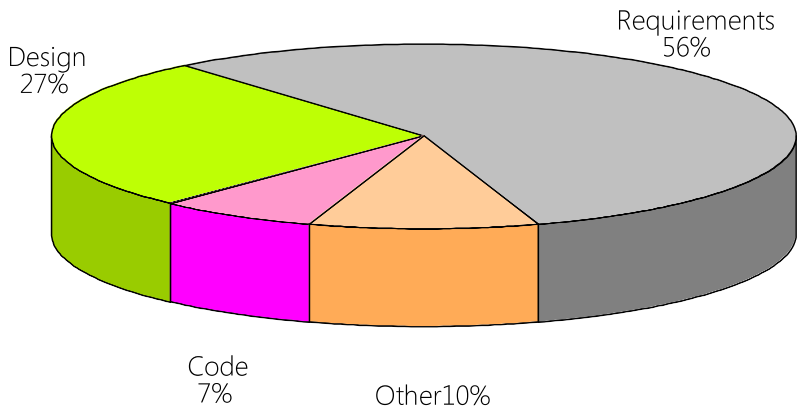
- The increasing complexity of system - more features, higher quality, higher performance, etc.
- The decreasing time to market

In a typical software engineering project effort is usually distributed with a heavy weighting towards the back-end of the project. That is, considerably more effort and cost is spent in the coding, test and debug phase than in the earlier phases of the project. Notice also that there is still effort being spent (usually on debugging) even after the project is released. For many projects this effort can be considerable.



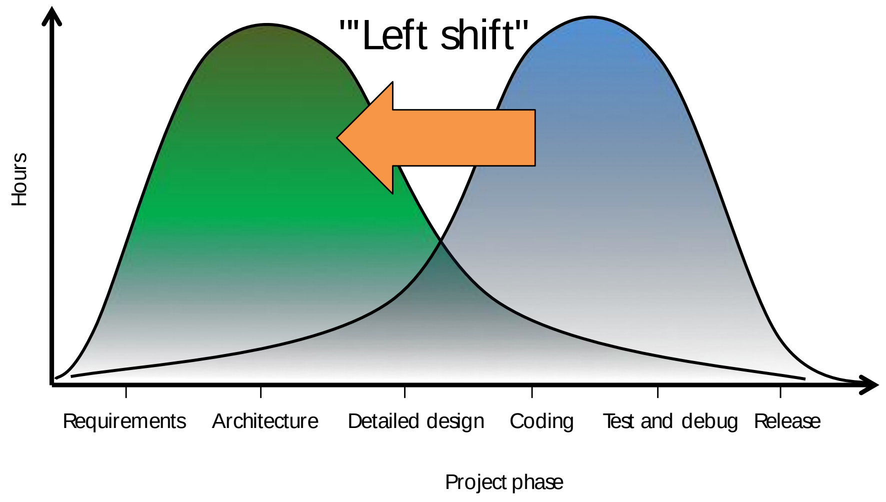
This effort curve ignores one vital aspect of engineering reality: later you find and fix bugs the more it will cost you.

It's not that simple; not all bugs are created equal. The distribution of bugs in a typical project tends to look like this:



What does this tell us? That engineers tend to make relatively few mistakes in their domain of expertise (code), but tend to misinterpret or miscommunicate designs; and (generally) have a pretty poor grasp of their customer's world. Ultimately, of course, the customer is predominantly interested (and disappointed by) errors that unduly affect the way they work; which, sadly, is where our engineers are making the most mistakes.

The intuitive solution is therefore to "left-shift" – apply more effort to those areas that are causing us pain.



Whilst seemingly an obvious approach, practically this raises some important limitations. When we build a software system we can verify that it is working correctly by testing it against requirements. Without a system to test how can we demonstrate the correctness of our system?

The solution is to build *effective* models.

What is a model?

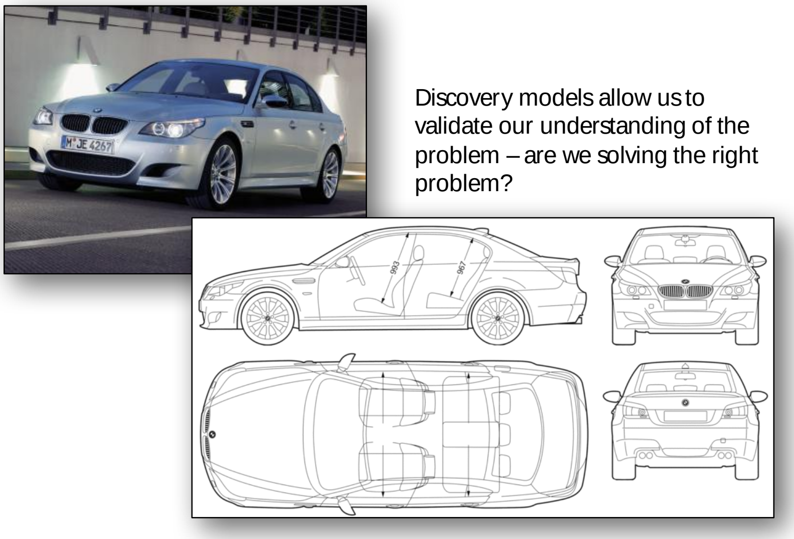
Models are a *simplified* representation of the real world. Model represents our understanding of the problem to be solved. Most importantly, we can ask *questions* of our models, we can query them, verify and validate them.

Producing a model requires an explicit action. There needs to be a clear understanding of the problem in order to express things in the model. That is, the models should not "lie" – there should be no implicit or invalid constructs concepts or relationships in the model.

Models fall into three main categories

Models for discovery

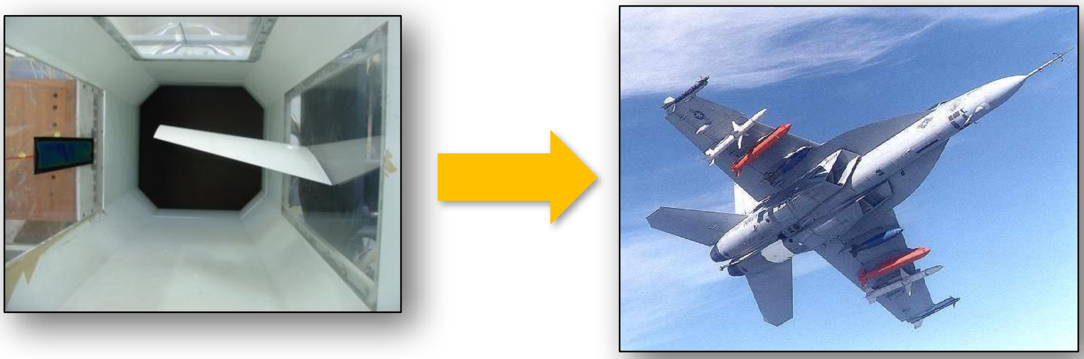
A discovery model is typically a semi-formal, or even informal, representation used to validate our understanding of the problem domain. We build discovery models to confirm we have understood the problem; to ensure we have a complete description; to clarify ambiguity; and to ensure our understanding is consistent with that of the customer.


Discovery models allow us to validate our understanding of the problem – are we solving the right problem?

Models for discovery are deliberately simplistic – you are not required to actually *build* anything from these models. Their primary application is validation – *are we solving the right problem?*

Models for understanding

The most common use of modelling is the construction of simplified representations of the system and its responses. The model allows us to verify the real system without having to actually construct it. The simplified model may be a different scale, different materials or may not even look like the real system.

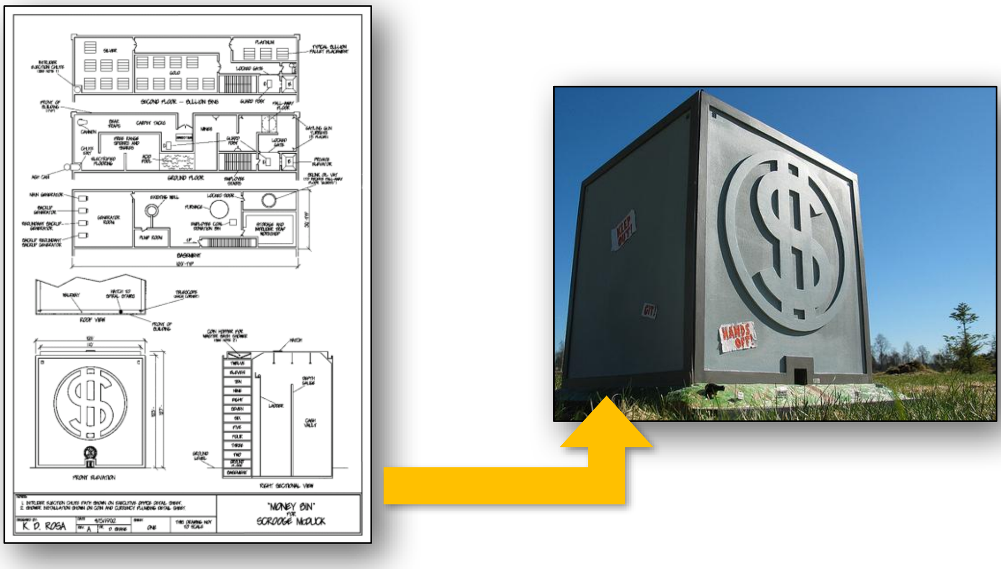

Exercising the model gives information about the real system.

Since the model is a simplification it can only be used to study a small number of aspects of the real system. Thus, we may have to build many – simple – models to get a true appreciation of how the real system works.

Models for construction

A construction model is a blueprint from which a real system can be constructed. The model must be detailed enough so that it can be built using defined construction techniques and mechanisms.

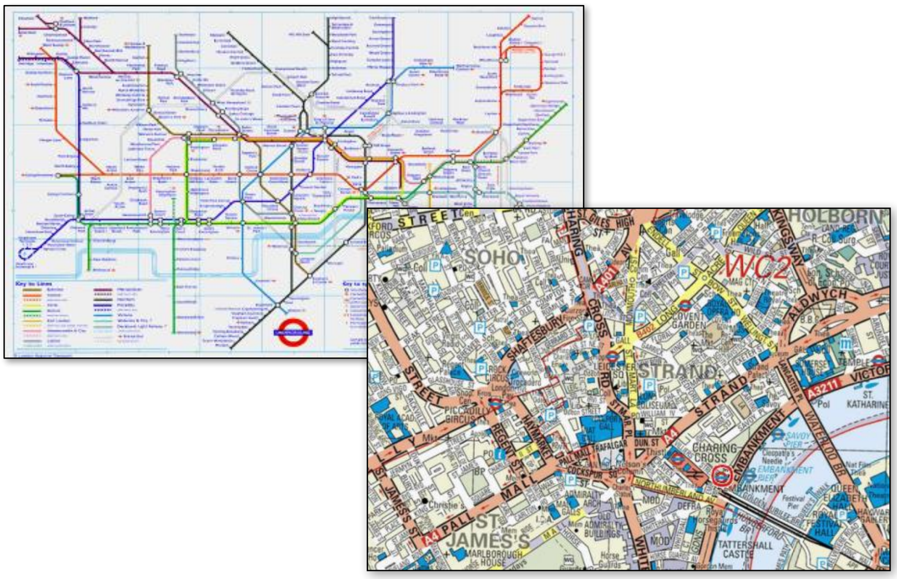
That is, the model must have enough information so that the builder can construct the system such that is matches the designer's vision.



The model must contain enough detail to construct from

Different needs, different models

Different model users often have different concerns. Different models can be used to present a viewpoint that satisfies those concerns. Consider the following abstractions of the London Underground system:



The top diagram presents an abstraction from the viewpoint of a tube traveller

The bottom diagram presents the underground system from the perspective of geographical location of stations

There is no correct, complete set of views; the models you produce will depend on your audience and their concerns. For example, a safety-critical system may require models demonstrating fail-safe or reliability behaviours of the system; high integrity systems may require security models; etc.

The model views for dynamic systems

Dynamic systems, irrespective of notation or methodology, all have certain characteristics in common:

Structure

They contain structure information – the elements that make up each element of the system, and how those elements are organised.

Function

Inputs to the system will be processed and manipulated to form outputs – this may be a flow of materiel, or information being manipulated by algorithms.

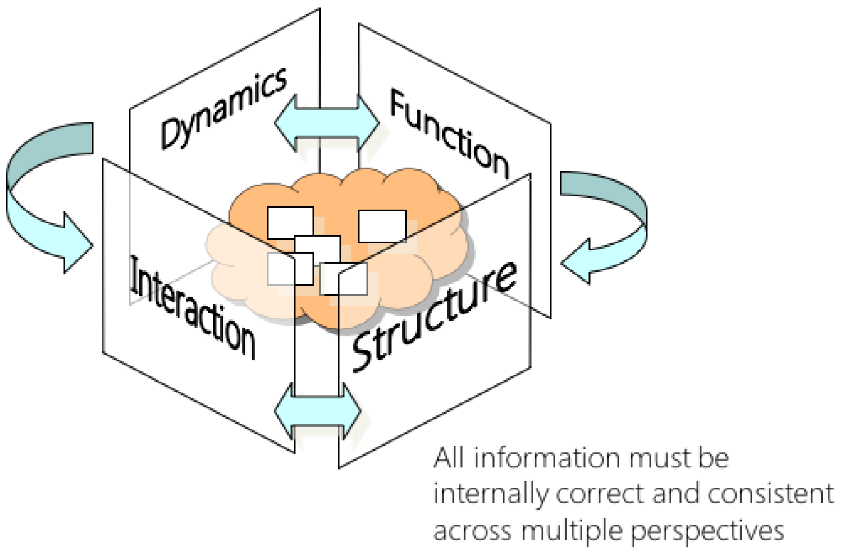
Dynamics

The behaviour of the system can change over time

Thus, when we want to model such dynamic systems we need tools (in our case, diagrams) to support each of these aspects.

These views are known as *Constructive* diagrams, since they can be used to specify the details of the software (our end product)

There is a fourth view included here: Interaction.   Interaction diagrams show how, why and when objects 'talk' to each other. These diagrams are known as *Descriptive* diagrams, since they do not specify the construction of the software but demonstrate *how it works*. These diagrams are key for design verification.


Dynamics
Function
Interaction
Structure
All information must be internally correct and consistent across multiple perspectives

Essential to building models is *consistency*. Ensuring diagram consistency requires consideration of each diagram's:

Correctness	Ensuring that the diagram syntax is correct
Validity.	Ensuring the diagram actually says the right thing!

It addition model elements and behaviour in one view must support (and be supported by) other views (cross-validation of the model). This is the basis of an *executable model* – that is, a model that could be used to generate working software.

Building effective modelling hierarchies

An effective modelling process features a hierarchical approach to modelling. At each level there are a different set of conceptual elements and the focus of the process is different.

At the highest level we consider the system as a black-box entity. The focus is on (requirements) analysis, rather than design.

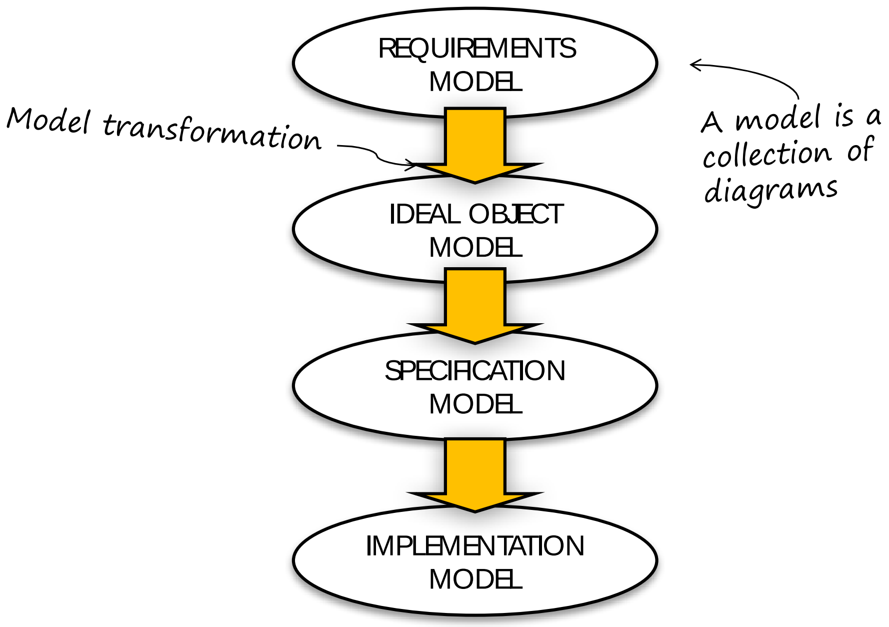
At the next level the focus is on synthesising a solution. At this level the focus is on good, modular, maintainable design. At this stage the elements will be oriented towards the problem domain, rather than implementation domain elements. Design evaluation is a key aim at this level.

The software design level extends the 'pure' design of the level above and applies 'engineering knowledge' to the problem.

The lowest level in the hierarchy focuses heavily on the solution domain. The emphasis in on efficient language (code) level design.

Conceptual elements	Focus
System	Behaviour of the system as a black-box entity External behaviour Requirements
Problem Domain Objects	Object oriented design Abstraction, coupling, cohesion, encapsulation Evaluation of concept
Solution domain objects Subsystems Composite structures Active/Passive classes	Software design Software structure Software dynamic behaviour and function Concurrency
Implementation domain objects Classes Tasks	Implementation design Code structure Build process

We should organise our design into a set of models. Each model consists of a set of (one or more) UML diagrams.



The purpose of the Requirements model is to facilitate understanding of the problem. It is a model of *discovery*.

The Ideal Object model focuses on creating an object-based design and evaluating it against the system requirements and constraints. In this model practical implementation considerations are (largely) ignored. The Ideal Object model is a model for *understanding*.

The Specification model extends and enhances the Ideal Object model to take into consideration practical elements of embedded and real-time software design.

The Implementation model focuses on language-level details and producing an efficient implementation based on the target language and platform considerations.

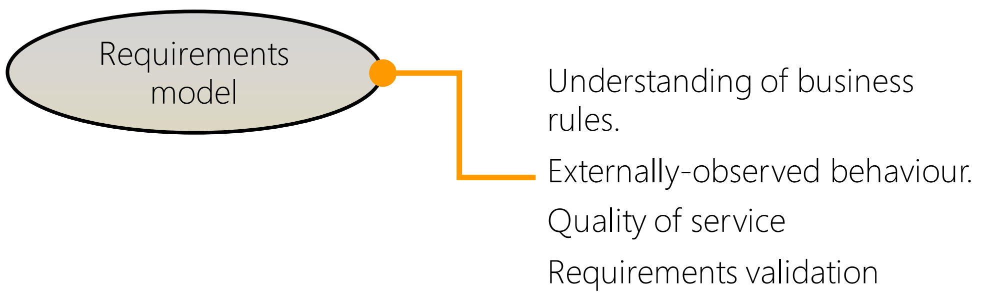
The Specification and Implementation models are models for *construction*.

Between each model are *Transformations*. The transformations describe the rules for getting from one model to the next (and back again – also known as *traceability*). The transformations are a key part of the design documentation.

Notice, as we go through the modelling process our models will become more and more rigorous and contain more information and more detail.

The Requirements model

The purpose of the Requirements model is to provide a stable understanding of the problem, prior to any design work.


Requirements model
Understanding of business rules.
Externally-observed behaviour.
Quality of service
Requirements validation

There is almost no synthesis (creation of objects / elements) in the Requirements model. In general, only a small number of objects are created - the system itself, and the interface elements that connect it to its environment.

The focus of the Requirements model is validation - have we understood, and are we solving, the right problem? Validation must be done via the project stakeholders (or their internal representatives)

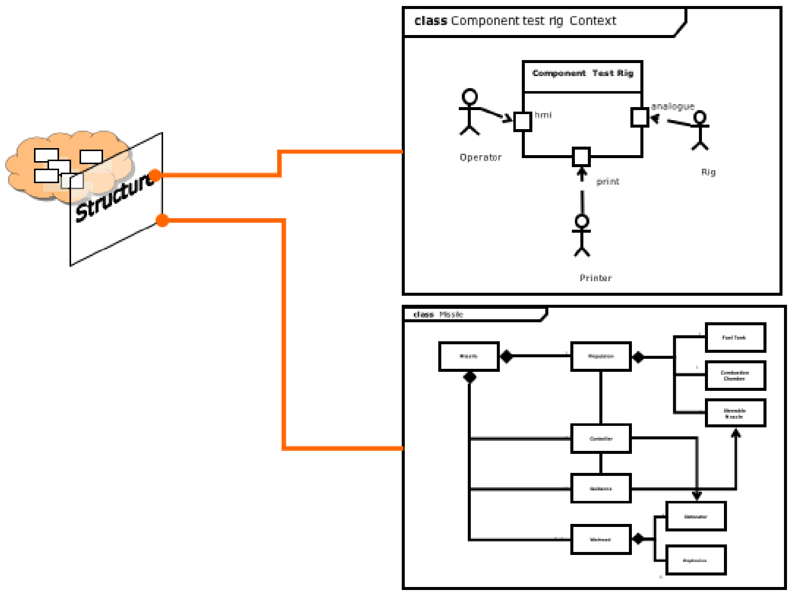
You should not be creating system objects at this level of abstraction. If you are, then you are probably (unintentionally) doing design, not analysis. Only the system and its interfaces should be defined; and these should be obvious from the systems design.

Structural diagrams

There are two primary structural diagrams that can be constructed for the Requirements model:

The Context diagram defines the physical scope of the system. It is based on a corruption of the Composite structure diagram.

The Domain model defines the information (data) content of the system and the relationships between the information (sometimes called the 'business rules' of the system). It is captured as a class diagram.

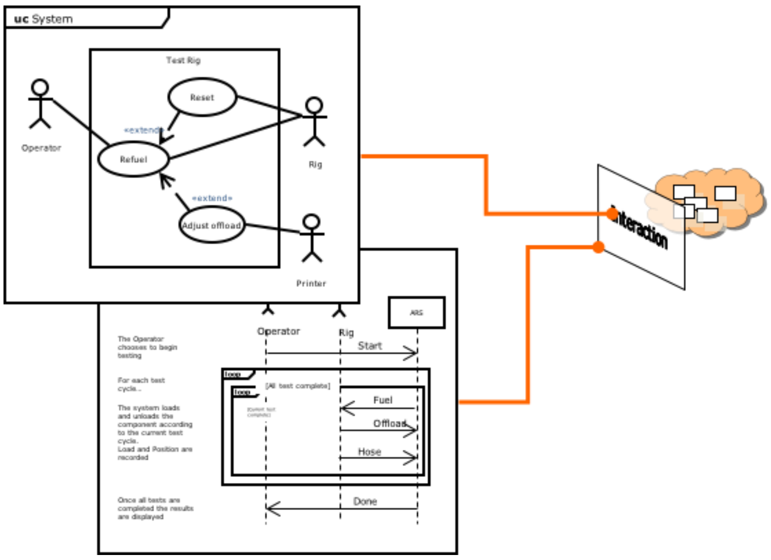


Interaction diagrams

The interaction diagrams provide usage-scenario based descriptions of the system's (externally-observable) behaviour. As with all scenario-based techniques, you cannot completely describe the system's behaviour; but the more scenarios you model the more complete your understanding is likely to be.

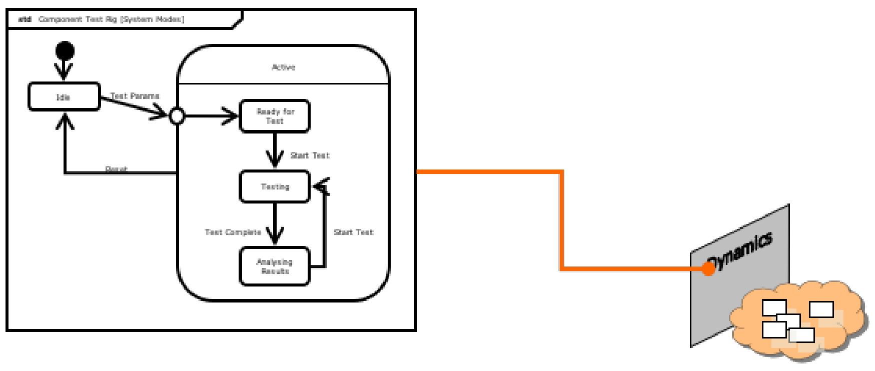
The Use Case model defines the functional scope, and transactional behaviour, of the system.

The Use Case Sequence diagrams formalise the use case behaviour as individual scenarios


uc System
Test Rig
Reset
«extend»
Refuel
«extend»
Adjust offload
Operator
Rig
Printer
Interaction
Operator
Rig
Start
loop
[All test complete]
Fuel
Offload
Hose
Done
The Operator chooses to begin testing
For each test cycle..
The system loads and unloads the component according to the current test cycle.
Load and Position are recorded
Once all tests are completed the results are displayed

Dynamics diagrams

The System Modes diagram shows the (externally observable) dynamics of the system as a finite state machine

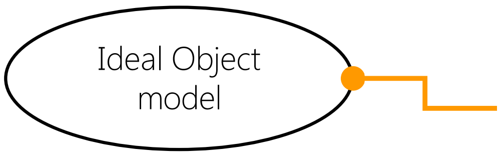


Function diagrams

Any flow-of-materials processing, customer processes or significant algorithmic behaviours can be captured with activity diagrams.

The Ideal Object model

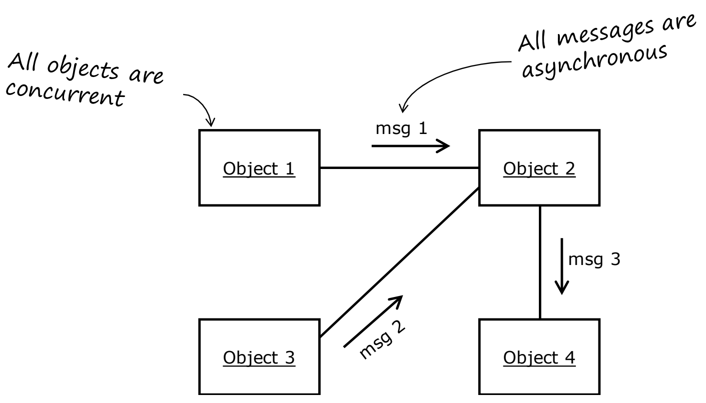
The purpose of the Ideal object model is to synthesize a design to meet the requirements of the system.



- Object oriented design
- Abstraction, coupling, cohesion, encapsulation
- Evaluation of concept

The focus is on demonstrating the design meets its requirements and is valid in terms of intrinsic qualities – that is, coupling, cohesion, encapsulation and abstraction. Because of this it is not necessary to build a full and complete set of executable diagrams. It is sufficient to capture the design; and to demonstrate that the design is correct.

The Ideal object model is a 'pure' design; it does not consider any implementation details.



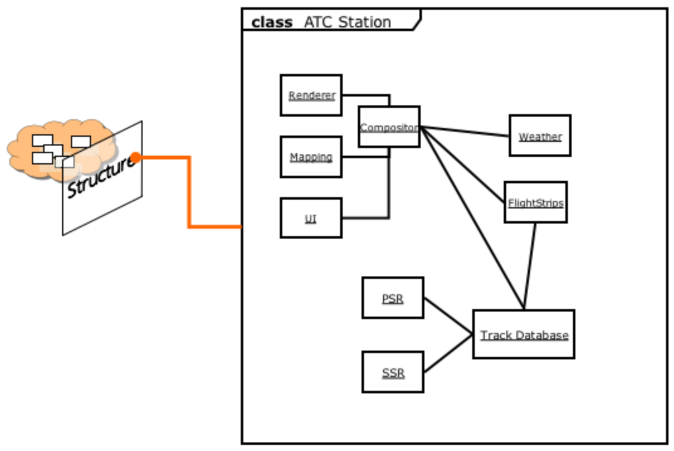
The Ideal model is a simplified abstraction of the 'real' solution:

- All objects are concurrent
- All processing is instantaneous
- All messages are asynchronous and instantaneous

This is the most general case, and does not necessarily represent any implementation reality. A direct implementation of the Ideal model would likely be very unresponsive and slow. If you can't get your design to work in this 'Ideal' environment it will be almost impossible to get it to work once you apply real-world constraints and limitations!

Structural diagrams

The Ideal object diagram shows the elements of the design and their organisation. It is not necessary to show types (classes) at this point.

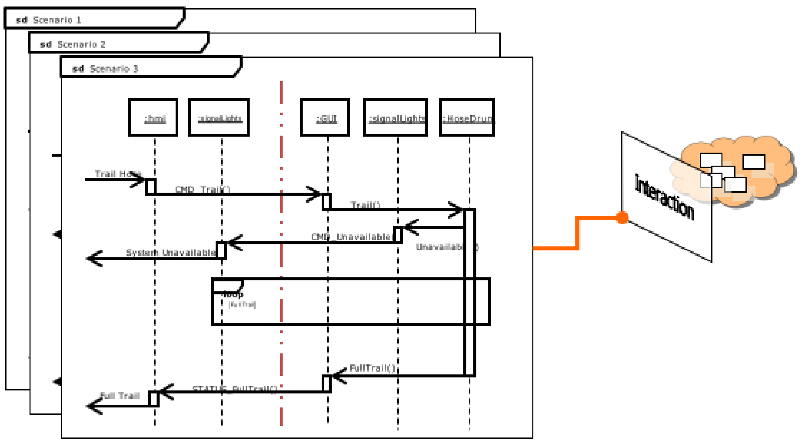


Ideal Object Diagram

Interaction diagrams

Ideal object sequence diagrams are used to verify the system operation by demonstrating the operation (interaction) of the design objects.

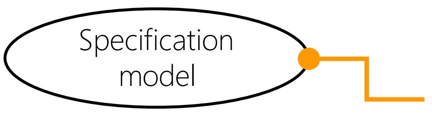
Note there will be multiple Ideal object sequence diagrams. These will be derived from the requirements sequence diagrams defined earlier.



Ideal Object Sequence Diagrams

The Specification model

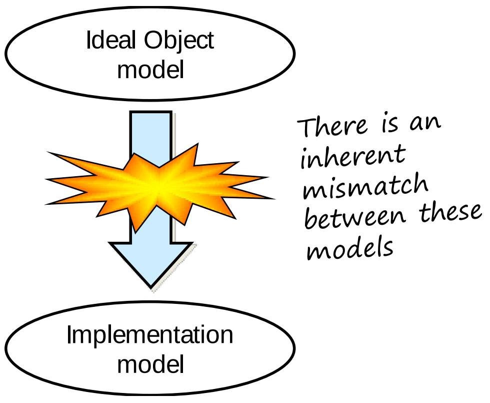
In the Ideal object model the aim is synthesise, and evaluate, an object design that has the capability to meet the customers' needs. The Ideal object model is an abstract design in that it consists of all concurrent objects, communicating via asynchronous messages (the simplest model to evaluate).



- Software design.
- Software structure.
- Software dynamic behaviour and function.
- Concurrency.

The Implementation model emphasises execution. In the Implementation model the design consists of sequential code, organised into concurrent units of execution.

Clearly, there is an inherent mismatch between these two models.



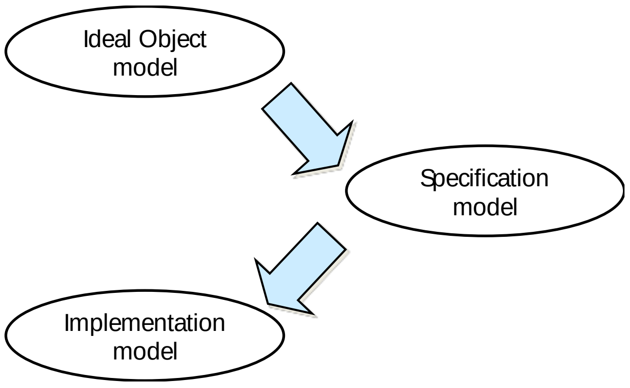
Going straight from the Ideal model to code can give rise to a range of verification and design issues, depending on the implementation environment chosen:

At one end of the scale, the implementation can be little or no resemblance to the Ideal design because of the scale and number of transformations required to convert the Ideal design to one that actually executes. In this case traceability from implementation back to design is a major issue.

At the other end of the scale an Implementation design that replicates the Ideal design could be highly inefficient and the system may not approach its performance requirements.

With such a potentially large gulf between the Ideal design and the implementation what is needed is a 'bridging' step.

The Specification model is an intermediate model to simplify the transition from analysis to implementation. The Specification model sits between the Ideal model and the Implementation model. The Specification model modifies the Ideal object model such that is can be transformed into an implementation model. The purpose of the Specification model is to make the transition between the Ideal design and the Implementation design as simple and as obvious as possible.



The Specification model uses the Ideal object model as its input. The reason is the Ideal object model has already been demonstrated to be a 'good' (that is, the least compromised!) working solution and is therefore an excellent base to work from.

In reality, the Specification model cannot be developed in isolation. There will be factors from the implementation environment – for example, the choice of programming language, operating system and/or hardware elements – that will constrain the design choices that can be made.

The Specification model focuses on the software detailed design. The aim is to produce a coherent, consistent set of models that fully describe the software. This is sometimes referred to as an executable model.

The emphasis of the Specification model is the practical realities of software design – behaviour, interfacing, concurrency and inter-object communications. The aim is to produce a specific design representation that can be transformed (mapped) to many possible implementations

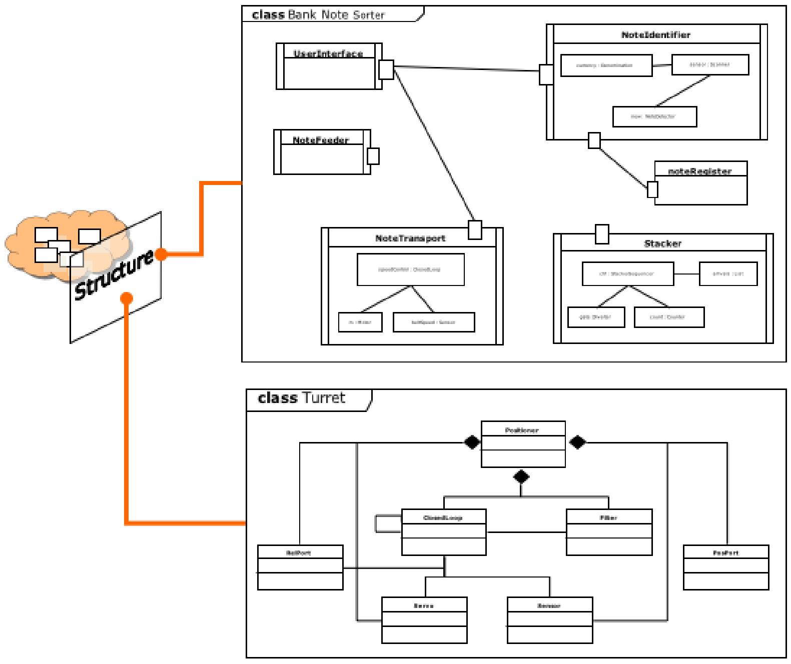
The Specification model should be defined independent of programming language, RTOS etc. This is why it is sometimes referred to as the Platform Independent Model (PIM).
This is the second stage in the design process where the developer can apply their engineering knowledge. However, in the case of the Specification model the engineer must apply what they know happens (or works) in the real world. In the Ideal object model practical realities can (and should) be ignored. Now these practical realities must be applied; and their effects on the design evaluated.

Structural diagrams

The Composite structure diagram emphasises architecture – component hierarchy, interfaces and interconnection.

The Class diagram defines the object types. The class diagram communicates:

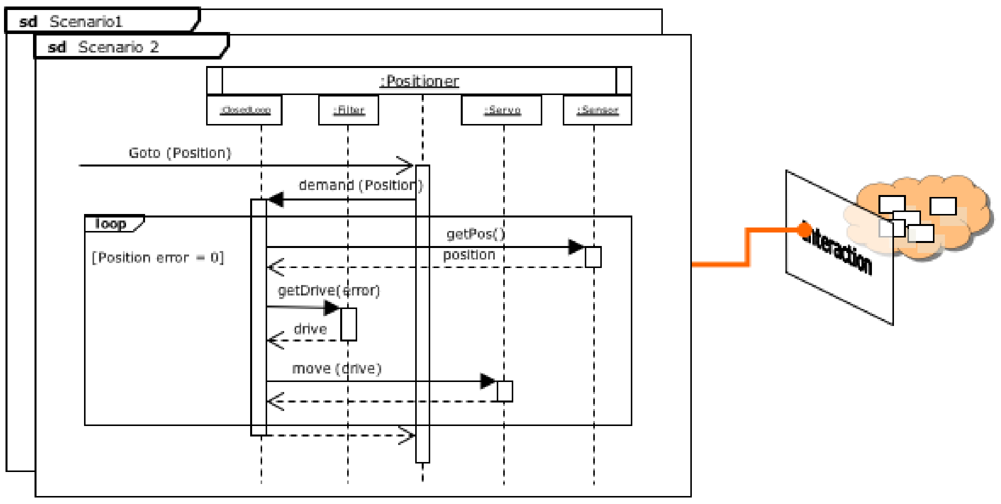
- The data content and interface of each object type
- Object visibility
- Object structural relationships (composition)



Interaction diagrams

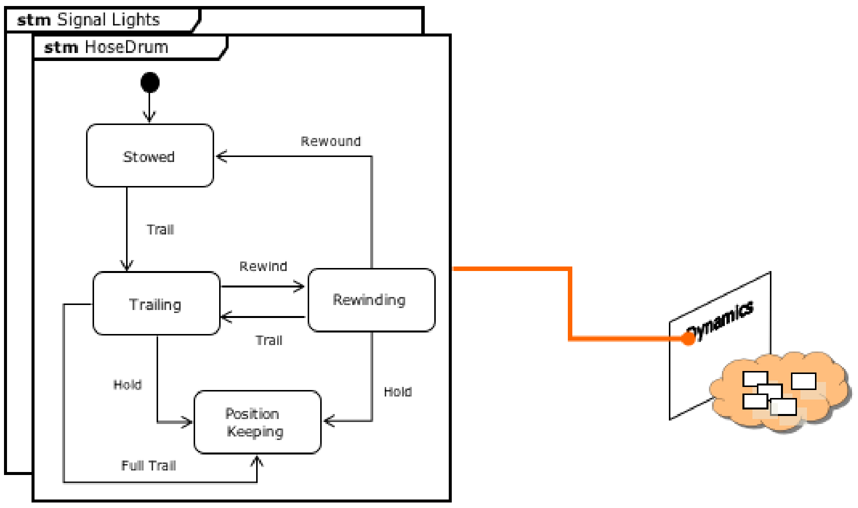
The Specification sequence diagrams show how the software design meets the customer requirements.

The Specification sequence diagrams are a refinement of the Ideal object sequence diagrams, taking into account changes required because of composition choices, concurrency design decisions and factors driven by the other design forces in the system.



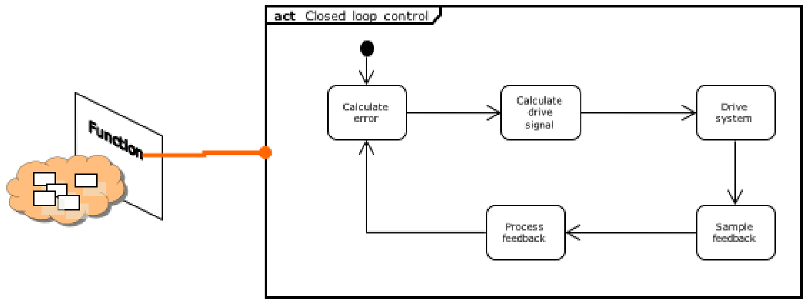
Dynamics diagrams

State machine diagrams show how the behaviour of objects change with time (known as 'reactive' objects). Only reactive objects require state machines defined.



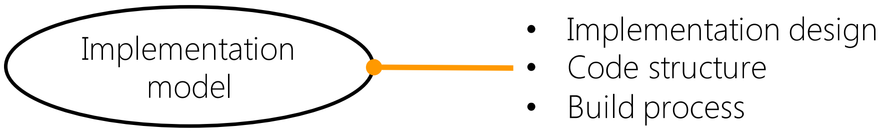
Function diagrams

Activity diagrams are used to define any *significant* algorithmic behaviour in the system



The Implementation model

The Implementation Model represents the software that will be coded. The model is hardware-, platform- and language-specific. It represents one possible implementation of the Specification model.

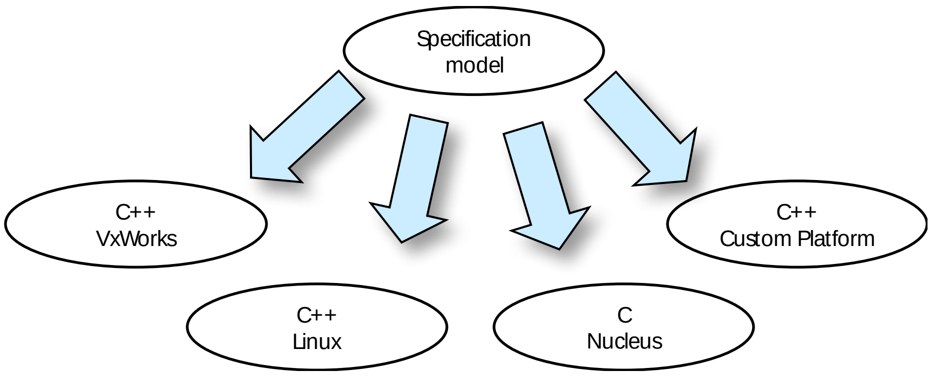


The Specification model is platform-independent – it could be implemented in many different ways on different combinations of hardware, Operating System (OS) and programming language

UML is not directly executable. That is, it does not have a direct mapping to any particular computer language (or have its own virtual machine). It must be translated, or transformed, from the Platform-Independent Model (the Specification model) onto a particular implementation. The Implementation transform rules map the Specification model onto one particular implementation.

Some features in the language can be translated very simply. Other features require an intermediate model – the Specification model is translated first to an intermediate representation, then the intermediate model is translated to programming constructs. The complexity of this intermediate model is somewhat dependent on the target language – languages that support object-oriented features, for example, require a simpler intermediate model than those that do not (for example, C)

The transformation rules should allow one particular Specification model to be implemented on a number of implementations.



The Specification model cannot be implemented on just any platform. The target platform must support a necessary set of features (in terms of hardware support, interrupt handling, concurrency, any other operating system features). Without such features, the Specification design cannot be implemented.

In a practical design, knowledge of the intended target platform will influence the design decisions you make in the Specification model. That is, the design is not an open-loop process: there is feed-forward from the design onto the target implementation and feed-back from the implementation into the design.

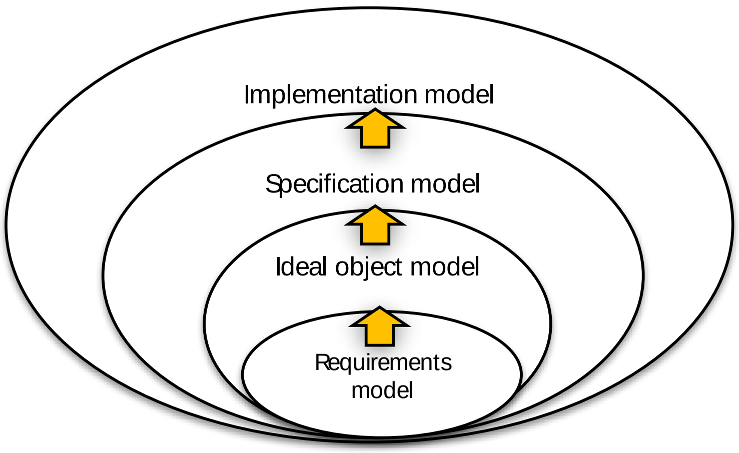
Very commonly the Implementation model is omitted. It can be argued that the most effective model to describe what software is supposed to do is source code itself, not a diagrammatic model

Organising multiple models

Once we have multiple models for our system we have to consider how to organise, develop and grow these models. There are two core approaches, which we shall call the *Evolutionary* and *Adaptive* approaches.

Evolutionary modelling

In the Evolutionary approach there is only ever one model of the system. The model (diagrams) is grown and developed as more is learned about the system under development.



At the start of the project an Analysis model is built to understand the problem. This model is then 'enriched' with design elements. Finally the design elements are transformed to form an implementation model which can be implemented.

The strengths of the Evolutionary approach are:

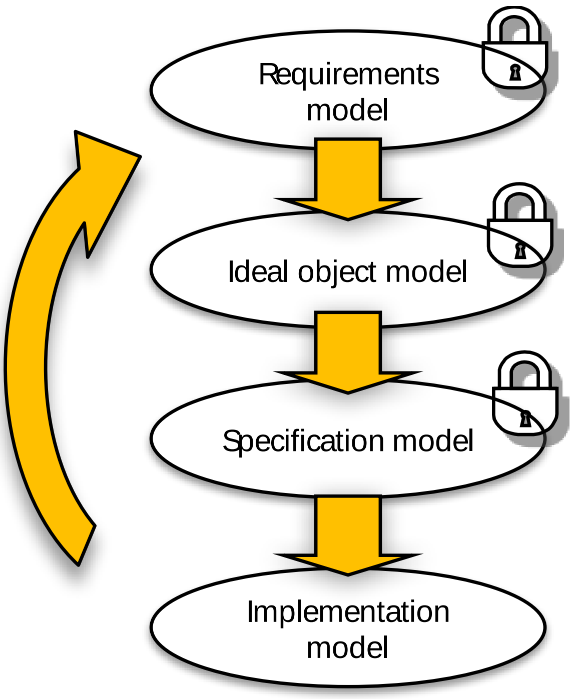
- There is only one model to maintain
- The model directly matches the current configuration of the software.

The major weakness of the Evolutionary approach is that transformation rules, design justification and evaluations are often lost (since they must be maintained independently of the model). This can make it extremely difficult for engineers joining a project to understand the background and rationale behind the design, since all they see is the a 'snapshot' of the design as it currently is.

Use the Evolutionary modelling approach when the Problem Domain and Solution Domain are very close. That is, when the problem can be expressed in concepts and language very similar to implementation language. For example, a network packet sniffer.

Adaptive modelling

In the Adaptive modelling approach, each model stands separate from all the others. Once a model has been created it is fixed and the next model created (adapted) from it using the transformation rules. The model is only then changed when new information comes to light (for example, a new project iteration)



The strengths and weaknesses of the Adaptive approach are broadly the opposite of the Evolutionary approach:

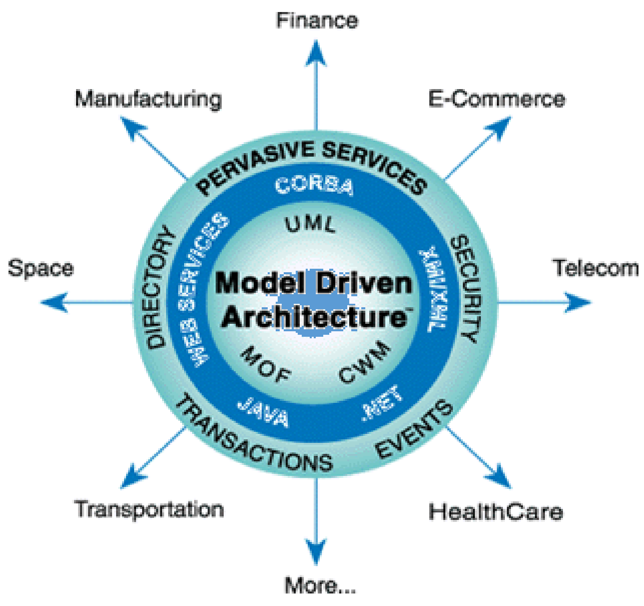
As a separate model exists at each level it is easy to see (and trace) the development of the system. Readers can refer to analysis or early design models to gain an understanding of the system.

There are now four models to maintain. Each change in requirements will impact all four models. It is therefore significant effort to keep all the models in sync.

Favour the Adaptive approach where the concepts of the problem domain are significantly different to the solution domain. For example, in an Air Traffic Control system, concepts such as Aircraft and Corridor are valid; yet have no relationship to how the concepts will be implemented.

How does this strategy fit with MDA?

The Object Management Group (OMG) Model-Driven Architecture is a modelling framework plus a set of tools and technologies design to provide vendor-independent model-driven development of software.



UML is one of the core enabling technologies; along with the Meta Object Facility (MOF) – a template for providing meta-data and meta-data interchange for tools and applications – and the Common Warehouse Metamodel (CWM) – designed for large scale database and data retrieval activities.

Also included with the MDA framework are middleware technologies like CORBA or Web Services, to enable the interchange of data across networks and disparate platforms.

A word of warning should be raised here: MDA is targeted at the IT application market and not the embedded arena. The core technologies and concepts have little relevance in (current) embedded systems.

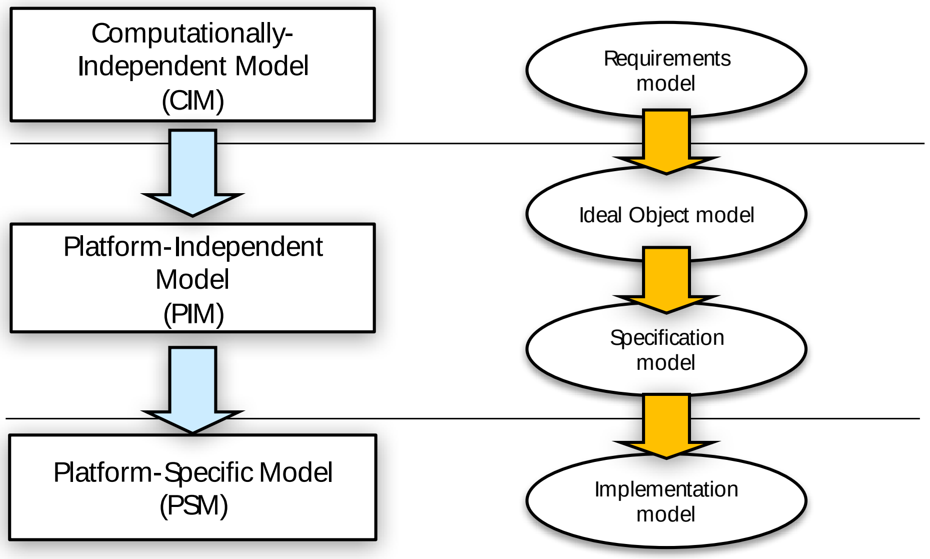
MDA's modelling framework consists of three models:

Computationally-Independent Model (CIM). The CIM is a (non-executable) model, often thought of as a Business Analysis model. The CIM focuses solely on the problem domain, its concepts and relationships.

Platform-Independent Model (PIM). The PIM is a design specification model. The PIM describes a synthesised solution in a language- and target-independent form, that may be transformed into one (or more) target implementations.

Platform-Specific Model (PSM). The PSM is a transformation of the PIM onto a particular target platform, language and other technologies.

MDA also specifies transformations between each of the models.


Computationally-Independent Model (CIM)
Platform-Independent Model (PIM)
Platform-Specific Model (PSM)
Requirements model
Ideal Object model
Specification model
Implementation model

The MDA models largely align with the modelling strategy described here. The major differences are in the Platform Independent Model, where we recommend two separate models (Ideal and Specification).

Summary

In modern software design we don't have the luxury of being able to build complete, complex systems which fail, and then learn from the results of our failures. To succeed we must *fail earlier* when the costs and consequences of doing so are minimised.

The key to this is modelling.

Building models allows us to explore unfamiliar problem domains, explore and innovate with new solutions and evaluate designs for correctness and quality before committing to expensive implementations.

Building effective models in languages like UML requires us to understand two important and fundamental concepts:

- How UML diagrams form orthogonal views into the design of a system; and that one diagram is not sufficient to completely define all aspects of a system's behaviour
- That models are built for different purposes; to ask different *questions.*

Having a strategy for organising your models and transforming between them is the key to effective modelling.

FEABHAS
EMBEDDING SOFTWARE COMPETENCE

Contact Us

Feabhas Limited

15-17 Lotmead Business Park,

Wanborough,

Swindon,

SN4 0UY

UK

Email: info@feabhas.com

Phone: +44 (0) 1793 792909

Website: www.feabhas.com