Ch 4 First Floor - Unknitting Knowledge and Intelligence

4.1 Where do we Find Knowledge?

On the ground floor we found the Maths lecture rooms - facts being pouring into young heads. The Maths library was reached from the mezzanine - lots of facts written down, although if you take out a book, you might not understand it. No, for knowledge, look on the first floor where knowledge is performed in tutorials and small classes. A proof in mathematics may seem like a series of disconnected statements, but when performed, the ideas dance across the statements - the difference between analysing a poem one line at a time and hearing it read as a whole.

What is the difference between knowledge and intelligence? No - we must unask that question. Knowledge is found in facts used intelligently, and intelligence is seen in reasoning organised by knowledge. Knowledge and intelligence are intimately knitted together, leading to the unfortunate neologism "knitwork". Think of it as viewing the same concept from different directions (figure 4.1 [4.1]). For computer science, this implies that knowledge-based systems and artificial intelligence are essentially two ways of viewing the same problem, and if you are not doing both, then you are not doing either.

Or think about training a chatbot: I can read and digest a page of technical stuff every couple of minutes. Extrapolating from that, I might be able to read 200 pages a day, 1,000 a week, 40,000 a year - 25 years to read a million pages. In practice, to avoid brain melt down (and to get paid), my cumulative reading rates were at most a tenth of that and most of my time was spent working with what I read. That is, I have gained my knowledge both by absorbing facts and by doing something with them - I did not learn to add up by reading a million sums. Hence I suggest that knowledge comes from the intelligent reading of facts. By contrast, for chatbots, the training sets for large language models can be orders of magnitude larger, but still such models cannot tell the difference between truth and invention. This chapter starts an investigation of how a knowledge and intelligence might be knitted together for a computer.

To be precise, this book proposes four interlocking theses:

1. Data, information and knowledge are three levels of the same problem;
2. Knowledge and intelligence are aspects of a single facility - the Knitwork;
3. A Knitwork acts on several levels, with higher level knitworks building on lower level ones;
4. A Knitwork is anchored by the behaviour of the system it models.

These theses provide a framework for explaining information sharing - how the knowledge that people want to share is mapped to data exchanged by their computers. In this chapter I want to describe what a basic knitwork looks like - amplify point two above. This is not a general theory of how a brain works, but a sketch of how intelligent computation could be conceptualised.

Figure 4.1 Intelligence - a Front View

4.2 Before We Start

While ideas about human knowledge and intelligence inform the discussion of knitworks, this book is focused on information sharing, hence I do not go as far as providing a complete description of knitworks. I only hint at how knitworks are made intelligent by meta-knitworks operating under the control of meta-meta-knitworks, and so on. That remains too speculative, and is more than is needed for discussing working together. Here I keep to the simple stuff that computers do, and leave real intelligence to human wetware. Hence our starting point must be artificial ignorance - the world of conventional computing. But the aim of this chapter is to show that a knitwork is more than just a

complicated computer program.

However, it is best to start from a primitive program. Such a program consists of a set of variables which hold data values, and a series of instructions that act on those variables. Say that the program models a system, and that the facts of that system are the values of the variables at the start and at the end of program execution. Inferences predict how the model - and therefore the system - behaves, and they correspond to the algorithms that the program instructions implement.

In self-documenting programs, the variables are given names that hint at the information they reify, and comments describe what the algorithm intends to achieve at each step:

```
NewBalance := OldBalance - Withdrawal; /** Pay a debit
```

The progress of the computation is determined by the combination of the initial facts and the algorithm, but it is for the user to determine how the outputs - the finished facts - are interpreted in the real world. I can write the code:

```
Sean.BankBalance := 1,000,000;
```

but the bank manager takes no notice.

Such a primitive program does one thing, and one thing only. Usually, one reuses a program by setting the starting values of the variables from a series of inputs - we can view the variables as a data model overlaid by an information model. As a data model, it defines what data types are used, and as an information model, how those values are bound to the parameters (state vector) of system modelled. The algorithms provide a set of processing routes that are fixed in advance, although particular combinations of input values may pick out different processing routes - but the program does not rewrite itself as it goes along. Such primitive programs are essentially deterministic, even if the only practical way to predict the outcome of a large, complex program is to run it on a second identical computer and ensure that all the same inputs occur in the same order. Machine learning systems are deterministic in this sense, even though they may give different answers if trained on different data.

An alternative to this "do exactly what I tell you" approach is the "do exactly what I ask" used in basic AI systems - here the fixed computational route is replaced by a search for the right answer. The program consists of recipes that can be tried out, plus a "you've done it" criterion to identify when the program has found the answer. In the extreme, such a program can generate every possible result that follows from the starting data, though more typically algorithms are "optimised", limiting the number of solutions generated. For example, a program might stop at the first workable solution.

To use information intelligently, the system would choose the algorithm to match the data. For example, different sorting algorithms will be preferred depending on whether the data arrives all in one go (e.g. Quicksort), or in dribs and drabs (e.g. tree sort) [4.2]. A system that processes the information on the basis of knowledge would infer the relevance of how the data arrives and choose the algorithm accordingly - as if it had read the text book.

Similarly, if you talk to an aerodynamicist, you find that they intelligently choose between algorithms based on the problem modelled. The various computational codes are based on the same equations, but they give different answers for the same starting conditions, one giving a more

accurate prediction for take-off, another being better for high-level cruise. The question arises, can we incorporate that knowledge into the design of the system? Before starting to answer that, we will look for hints in a couple of existing computing techniques.

4.3 Knowledge Acquisition and Documentation Structuring (KADS)

A knowledge-based system aims to use the same knowledge that experts use when they solve a particular problem. Knowledge-based systems have been around for many years, and here I outline one methodology - KADS - for developing them. Like many methodologies from the 1990s, it is encyclopedic, a rich complex of ideas set out in detail - or equivalently, long, complicated and rather tedious. References to T&P [4.3] are to the course book used by an Open University MSc course. I am not here to advocate a particular KBS method - the interest here is how KADS can contribute to the idea of a Knitwork.

It's a methodology, so much of KADS is about the process of system development - the journey from wanting a system to having a detailed design to code from. Some of the elements come from conventional software development, for example, T&P illustrates project management using the PRINCE methodology [4.4]. But some of the suggested techniques were specially developed for working on knowledge. For example, in the systems analysis stage it describes the Card Sorting technique [4.5]: a number of concepts from the problem space are written down on cards; experts are asked to sort the cards into three or four piles; the analyst asks experts about the categories they have chosen - tries to determine which aspects of the problem are important and in what way.

What is of interest here is the construction plan for the KBS. Conceptually, a KADS KBS has four layers of expertise [4.3 Ch 4.3.1] - from the bottom up:

- Domain - the basic facts of the problem in the KADS Domain Modelling Language, which is basically yet another entity-relation data modelling language;
- Inference - the methods that can be applied to facts in the domain - data in the domain layer is connected to inferences using *inference roles*;
- Task - combining a series of inferences to achieve some end;
- Strategy - identifying which tasks can achieve the system goals.

A particular data element from the domain level might be used in a number of different roles, with an example given being the name of a bacterium, which may be an input to a diagnostic task or an output from it, depending on the inferences used. There is a library of inferences, such as *Instantiate, Generalise, Classify* and so on - fine grained steps. These are combined (at the next level) into generic tasks, such as *Diagnosis, Planning,* etc.

KADS is not a general method for representing knowledge, it is a method for analysing and automating specific tasks. KADS describes the strategy layer as choosing the right goals and configuring the tasks, but T&P is quite thin on detail on how this might work in practice, and acknowledges that in the simplest cases a rigid task structure will suffice. I infer that in the cases that it had been used, the KADS approach has been focused on well bounded problems - which is fair enough from the viewpoint of the person paying for the work. This raises an interesting question: if one could formalise the knowledge needed for the strategy layer, could one construct a KBS to construct KBSs? (Not answered here).

Note that KADS explicitly combines data and algorithms, with data being processed based on a strategy broken down into tasks further broken down to inferences. These are chosen from knowledge about the problem, and one of the specific ideas that KADS highlights is the way

experts pick up on particular facts to identify exceptions, and choose alternative strategies or tasks.

The second feature of interest for knitworks is the use of *inference role* to instantiate the relationship between a data element and an inference. Separating out the data elements from the algorithm enables a many-to-many relationship, in which the same data can be used in many different contexts, and the same inferences can be applied in many sources of data. This is also implicit in the inference patterns laid out by the traditional syllogism, and in the practice of computing on data types without regard for the information they represent. That is, there is a layer of knowledge which decides whether an operation is appropriate for the starting facts - whether, when three people share two elephants, they should own a notional 2/3 share each or physically chop them up into a bloody mess. More broadly, this leads to two questions: what inferences can be applied to this element of data (select inference)? and what data can be used with this inference (select data)?

That is, KADS gives us a way of talking about flexibility in the way facts and inferences are combined, but does not obviously extend to the next level: how do we use that flexibility in a flexible way. The next layer up from inference is generic task models [4.3 ch 12, p260 & 262]]. These are used to "initiate and drive the knowledge acquisition process", and it is also noted that KADS lacks a formal notation for specifying generic task models. That is, as originally conceived, KADS is a method in which humans provide the knitwork needed to build general reasoning techniques into something that can be applied to a particular problem.

The key take-away from this is that if we want to create a true knitwork, we need to recursively apply the concept of networks of knowledge and intelligence to itself - knitworks to reason, knitworks to reason how to reason, knitworks to... and so on. How many layers of knitwork does it take to infer that Epimenides' "Liar paradox" (All X are liars. I am an X) is not soluble? [4.6] I've no idea, though I am going to guess four or five.

4.4 Thinking about OWL

OWL self describes as a knowledge representation language; "...Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things" [4.7] It uses description logic [4.8] to represent classes, individuals and their properties and relationships. Description logics build descriptions (facts) in formal structures of propositions that can then be analysed in a reasoner (inferences).

```
"The car    is                  blue"
<class element> <has property <property = colour>> <particular colour>
```

OWL also defines a subclass of a class and a subtype of relationship. An OWL reasoner makes inferences over a model written in an OWL dialect [4.23]. For example, it can infer that if a class has a particular property, then the subclass also has that property; or, for example, it can follow through a sequence of transitive relationships - if A is ancestor to B and B ancestor to C then infer A is ancestor to C. And essentially, that is all it does. It imposes no constraints on the meaning of its classes or relationships, and any interpretation of those classes or relationships is external to OWL. That is, in the terms of this book, OWL provides a data type representing sets and relationships between members of sets.

That might not sound very much, but if you have a spare afternoon, you might take yourself back a

hundred years to read a then standard text book on Logic [4.9]. You will discover that many pages discussing syllogisms can be reduced to half a dozen statements of Boolean Logic, which is not the half of what OWL can do. Early demonstrations of the Semantic Web often consisted of hand-crafted mash-ups - answering questions that require data from two or more different sources - they simply used OWL as a data modelling language. But with OWL descriptions of each site, one can use an AI reasoner to automatically search for queries to the individual sites which can be combined to give an answer based on linking data across different sites [See Ch 1 appendix &4.10]. Been there, done that, but the unresolved problem was that data is not information, hence this book. (I also help lobby for the Data Protection Act 1978, which was designed to prevent such mash-ups without the data subject's permission.)

That said, in practice, OWL seems to be used as a naive information modelling language. It provides formally defined semantics for the data model, but its approach to information semantics is to smear an informal verbal definition over the top of the data model - classes with natural language names and a "words refer to concepts" assumption.

It thereby provides an easy first step for small scale information models within a limited context. This is particularly unfortunate because OWL has been developed for "the Semantic Web", where users take their terms from pre-defined, open vocabularies, in order to enable interoperability across the whole world wide web. In theory, the vocabularies are linked to a context by being defined within their own namespace - that is, every class or property name is prefixed by the web address where the context is defined. But OWL does not come with a way of linking namespace to business process or system behaviour, or any of the contextual knowledge that avoids the flexibility and semantic drift of natural language - you cannot build firm foundations for a formal language on the jelly of natural language, especially when solidifying the jelly was the driver for creating the formal language.

This is not to say OWL is irrelevant to data and information modelling. Firstly, it provides relationship subtyping as a basic construct, something that has to be shoehorned in to conventional modelling languages such as EXPRESS [4.11]. Secondly, it can give clear semantics to hierarchical reference data, since every subclass of a reference data class is - by virtue of it being a subset - necessarily a member of the parent data class. Thirdly - and this suggestion has particularly annoyed Semantic Web zealots [4.12] - it provides a route to formally defining the semantics of conventional data models, for example by making "Entity" in an EER model into a subclass of *thing*.

Taking about information modelling and OWL, JC3IEDM [4.13] is a standard for information interoperability between NATO forces. I was involved in a project in which its large Entity-Relation data model was automatically converted into OWL. The resulting model was not an efficient rewriting and at the time was too large for the reasoner of choice. Moreover, that conversion did not include the extensive reference data library that comes with standard. Of more significance here is that writing the automatic conversion was a small task when compared to the considerable effort that has been put into the JC3IEDM information model and into ensuring that the different NATO forces mean the same thing by the data elements. Moreover, this information model started from military terminology which has been refined over many years in an environment where misunderstandings risk lives. And yet, even starting from military terminology, aligning the terms used between different NATO forces has been a subject for regular meetings over many years. The semantics of its information model do not assume that words simply name concepts, but that they are exhibited in how armies act on the information JC3IEDM provides - and conversely, knowledge representation is more than just a data model written using a vocabulary taken from natural language [4.24].

4.5 Inferences

One of the ways knowledge is exhibited is in the intelligent use of facts. Although it is intelligent to apply logic to come to a conclusion, logical inference itself uses facts unintelligently - it is a relentless application of mechanical deduction (ch 2). Mathematicians, by contrast, like to tell stories. Admittedly, the stories that they tell are not easily understood by non-mathematicians. You can see that when engineers try to tell mathematical stories - they give you a torrent of bare facts, but rarely say anything interesting on the way [4.14]. Mathematicians' stories are about imagination - and who would imagine that a rule about numbers is also a story about triangles, but we remember Pythagoras's theorem because of this surprising connection.

I was inspired to do a maths degree when I learned that solving simultaneous equations, drawing lines to find their intersection and inverting a matrix were three ways of looking at the same problem. When I hear a pigeon clap its wings to take off, I think back to my fluid dynamics course, and remember how a vortex gives lift to a wing - of Kelvin's Circulation Theorem and the Joukowski Transformation [4.15]. And it was the imagination and ideas exhibited by the proofs of pure mathematics that shaped my choice of courses (and that differential equations are boring). For a logician, every line of a proof is a new theorem, but for a mathematician, a theorem in an interesting and often surprising result. A number of my algebra lectures started by proving that certain axioms were equivalent - that what were the end results of one formulation were the starting points for another. North Whitehead and Russell used logic to prove arithmetic [4.16], while Gödel used numbers to show there are questions that logic cannot answer. From which I infer that the proper model for intelligent inference is mathematics rather than logic.

So, starting from mathematics, what are (some of) the types of inference that are available?

- Logic - basic operations on the values zero and one;
- Algebra - games using whole numbers and the discrete operations on them;
- Continuous Analysis - using real and complex numbers for the fine detail;
- Pattern Recognition - matching a signal against a pattern to filter out noise;
- Bayesian Belief Networks - working back from effects to probable causes.

The intelligence behind each inference is working out what kind of mathematics is needed to solve a particular problem. (Non-mathematicians may wish to spend a couple of months browsing Wikipedia to discover what these are different branches of mathematics are, each with their own distinctive methods.)

Let's take a worked example from a riddle: "A man shoots a bear, walks a mile South, a mile East, and on walking a mile North, finds he is back at the spot where he shot the bear - what colour is the bear?". Hints come from non-Euclidean Geometries, where parallels may converge or cross, and Lie Algebra, a mathematics for walking in squares on curved surfaces. Now think about how a co-ordinate square gets squished into a geometric triangle - and we are on a sphere at the North Pole and so the bear must be white (there are no bears at the South Pole). (Note to future generations - this was written at a time when the North Pole was still covered in ice).

That is, we need to use the facts of the situation to identify the type of inference (mathematics) we want to use - inference based on knowledge. Fortunately, at least for the problem of data integration, most software is not written by mathematicians. Most software uses simple discrete maths - is WithdrawalAmount more than BankBalance? - or maps to a discrete value, as in

can-I-get-to (fuel level, fuel consumption, distance to destination) -> Yes/No

There are problems of more complexity - does the vibration spectrum on the propeller shaft warn of impending failure of the ship's propulsion - but the engineers who pose such questions know a bigger range of mathematical techniques than the average programmer. That is, the problem of choosing the right inference comes down to characterising the (mathematical) properties of the context - which is, in practice, creating a business process model or a systems engineering model - either systems analysis or systems engineering (the word "systems" meaning different things to different professions).

4.6 The Facts

The initial temptation is to describe a fact as a fragment of an information model :

The car	is painted	red
[entity]	[property]	[reference data value]

This has the advantage that a fact is an explicit combination of information elements together with their representation. BUT

Test Case: What is the number of a *Part* in a product? Sounds like a simple enough question. But Product Identifiers are more complicated than a simple *Part.part_number.* Many information models follow simplifying assumptions that come from a siloed approach - where a particular business process is considered in isolation. As a result, in software there are many variations on *Part_number,* as are needed by different industries and their processes.

In one study - unfortunately not separately published - I found a dozen or more variations on the product identifier model. A visual inspection of the models showed a number of recurring patterns and from these patterns it was possible to automatically generate code to translate from one model to another. The results were, of course, limited to cases where the source data model contained all the elements needed by another. It should also be possible using description logic to automatically infer what elements of one pattern are the equivalents in another, but reasoners are designed to work one enquiry at a time, and don't apply the result from one inference chain to the next set of data. That is, when there are a million parts to process, description logics may prove orders of magnitude slower that a fixed bit of translator code.

And a further complication: although every physical part may be marked with a product identifier, not every part of a product can be so marked (two senses of "part"). For example, a fuel tank in the bowels of a ship may physically consist of bulkheads, the ship's hull and the decks above. The concept "fuel tank" only exists in the fuel systems view, not the physical view used in manufacture. Each wall of the tank will be built into one or another section of the ship, but the tank exists only when the sections are welded together. That is, in order to pick out the physical parts that need repair in "corrosion in Fuel Tank 2", you will need cross-references from the product structure of the fuel systems view to the as-built product structure. Moreover, that product support problem traverses at least three distinct information systems - product design (ISO 10303-242), in-service support (ISO 10303-239) and Integrated Vehicle Health Management (MIMOSA-CBM). Intelligent data model traversers are an interesting problem, but not one I had the budget to solve. Moreover, traversing these structures may also involve traversing company and organizational boundaries, and hence a fight through security and IPR issues. Extensive intelligent processing is needed to find

simple facts, which brings us back to "a fact is a view on an information model", rather than a simple extraction of a few data elements.

And this only the complications of finding a product identifier: Consider the "simple" case of the fact that states that a fuel tank is spherical, written as a fragment of the product data model based on an extended version of ISO 10303 [4.17]:

> **Fuel Tank** [product] **is**
> [-> product version -> mechanical view -> formation -> property (property type = shape) -> property_iteration -> property_representation (representation type = CSG)) ->]
> **Sphere** [CSG primitive].

That is, the fact "Fuel Tank is a sphere" is far too complicated to explain in less three chapters. For the hyper-geeks among you, it is a traversal through a series of different entities which separate out both different functional views and properties within the synchronic view of the model, and also different points of progress in the diachronic view, as described in chapter 3.

So, while we may modify our starting point to say that a *fact is a view of of an information model*, considerable intelligence may be needed to pick out the salient fact across a multitude systems which hold the data in multiple formats.

4.7 Knitting Things Together

4.7.1 Indirection and Flexibility

Indirection occurs when one entity is used as a proxy for another, but the thing pointed to is only determined when the user asks for the entity. For example, one key variation on Product Identifier is Stock Number: the physical resistors in an electric circuit are identified by their original manufacturer's part number, however, resistors from multiple manufacturers with the same characteristics and quality will be grouped together under stock number. If a manufacturing Bill-of-materials lists resistors by stock number, this allows a manufacturing line to use any of the resistors currently in stock, and to change suppliers if one cannot meet delivery dates.

A stock number points indirectly to multiple part numbers, but is only bound to a particular part number only when an individual part is needed. In software, a pointer can be used to represent this flexibility. And this type of flexibility is implied by "facts used intelligently" and "inference based on knowledge". Just as we have stock parts, we also use stock inferences - we use a column of figures to add up accounts, hours worked, trees planted and so on - the same method applied in many situations. Or, we use the same fact - "Osaka is in Japan" - to plan a holiday, identify where a film is set, or simply fill in a crossword puzzle.

That leads to four questions:

- What combinations of facts and inferences are possible?
- Which of those combinations match the problem at hand?
- How do we validate that we have chosen the right combination of facts and inferences?
- Where does the knowledge lie to answer the above questions?

The next section (4.8) evades the final question with the answer "in a knitwork of knitworks", while this section only tackles the first of these questions, as this provides sufficient apparatus for the rest

of the book. For the answer to the middle questions, I will wave in the general direction of "subject matter experts have worked the topic through".

4.7.2 Data and Formal Methods

The term "formal methods" refers to a way of specifying software using logic, usually at the level of a subroutine/procedure. One example is the Vienna Development Method [4.19], which uses the Logic of Partial Functions (LPF), as discussed in chapter 2. The specification for a routine/procedure consists of a set of pre-conditions - assertions that must be true for the procedure to work correctly - and post-conditions, assertions that should be true about the outputs. For example, for a function for N! (read as "N factorial", being the product 1*2*3*...*N) :

```
integer: Factorial(input integer N)
pre-   N > 0             /** undefined for negative numbers
post- Factorial = N!    /** ! is the operator for factorial
```

Basically, if the pre-conditions are not True, then the procedure might not work for the inputs, and if the post-conditions compute to False, then the code is wrong. One might imagine that such a practice would be universal across the software industry, but mention it to a programmer, and while they may not run off with the screaming abdabs, the response is functionally equivalent - formal methods are too difficult. In most cases this is because programmers literally do not know what they are doing. Writing software is a design problem - you don't exactly know how to get from the requirement to the solution, or even if it is possible, but writing software is the way you find the path connecting them. Often, writing a formal specification is only practical once you've written the software - but then it counts as documentation, which is not encouraged as it cost time and money.

In VDM, the conditions are framed in terms of the parameters of the routine/procedure and the data types used to represent them - any consideration of information will be in the mind of the programmer, as computer chips only operate on data. Consequently, a formal specification does not *mean* anything. It is a formal statement about data types and the values they take. Two procedures may be written for entirely different purposes but can end up with the same specification: that is, procedures written from the point of view of the different information they act on turn out to be identical at the data level. And, in fact, such revelations can be quite helpful - to solve this new problem, we can use this known method simply by changing the interpretation of the data. In KADS terms, we bind the facts to an inference using an *inference role.*

So, at the data level, one could define a "data signature" for each inference - a fragment of a data model required as input or the target of output - and then match the signature to the facts - a fact seen as a fragment of a data model for the whole program. That is, given a set of signatures for inferences, we can search through our data - as described by our data model - to find subsets (facts) that can be inputs to our inferences. Perhaps, even, we can use some intelligent processing to find a transformation that can match the inference data signature with that of the facts we have. In business process terms, this is equivalent to reading through a manufacturer's catalogue to see if any of their products meet the specification for a component we use - *Stock Number* is simply a memory that such a match was found.

However, at this point we have an issue: just because you can make an inference on a data type, it does not imply that the inference is appropriate to the context. You can divide the number two by three to give a fraction, but how does this help sharing out two elephants between three people?

Actually, this is a bad example, because counting elephants should be done using natural numbers, for which the answer to "Two divided by three" is "zero remainder 2", rather than 0.6666 - and a portion of elephant meat can no longer be counted as an elephant. But you get the point.

To summarise, we can conceive of the knitwork at the data level, matching inferences to facts, but we need a better model of the task being performed to decide whether any particular chain of inferences is appropriate. And that involves understanding the situation in which the inferences are used, which brings us back to knowledge about the process or system that is the focus of our interest.

4.7.3 Knowledge and Knitworks

At this point I will return to an example I used in "Aircraft as a System-of-systems: A business process approach"[4.18]. That illustrated in detail what can summarised as: there is a duality between the design for a product and the process of designing it; the product design exhibits knowledge recorded by a series of engineers, while the process brings together the right knowledge and in the right order, enabling each engineer to build on the designs of engineers earlier in the process. In my book, a whole chapter is taken up with describing the design of a simple lever. That lever is used to transmit the force of an actuator to move a flap on a wing. The lever's mechanical design is discussed, along with the work of stress analysis and manufacturing planning.

Here, it is convenient to focus down to just the shape. The lever is a single, closed solid, with critical measurements designed to meet the control systems model of the lever - an outline drawing showing the distances between pivot points, and detailing the amount of movement and the forces involved. The design engineer uses a CAD package to detail the geometry for a physical part, using also knowledge about material strength, stiffness and manufacturability. The CAD model is used directly to calculate the volume of the lever, and thence predict its weight (basic rule of aircraft design - don't make the plane too heavy). The CAD model is also used to create a finite element mesh for stress analysis, where the forces to be applied to it are used to calculate any deformation of the lever and highlight potential points of failure. The shape model is also used to generate a machine tool path to cut the part from a block of aluminium. From the data point of view, we have considered a particular data type - a 3D solid - and identified the computational operations we can do on it - volume, finite element mesh, machining.

From the process viewpoint, the control systems engineer takes requirements from aerodynamics to define a functional specification of the lever. The functional specification is turned into a physical shape by the design engineer. The stress engineer verifies the design, and the production planner generates the manufacturing plan, including the machine tool program. From the process viewpoint, we know what knowledge each engineer brings, but the computer geeks need to ensure that the outputs of the software from one stage are in the form needed as the inputs for the next.

The past fifty years has seen a steady progress in mechanical design, not only with design software replacing pen and paper, but also with workflow software sending completed work on to the next stage. Some thirty years ago, expert systems were being designed to automate specific parts of a design process, however these seem to have fallen out of favour. Part of the reason is likely to be that specialised software is expensive to maintain, particularly where it extends general purpose tools such as geometric design. Moreover, the task of designing a aircraft is so vast that even the biggest integrated package contributes only a small percentage of the knowledge needed. And

besides, by the time you have designed a couple of planes, the technology will have changed, changing the design process, and leading to a rewrite of the knowledge. Consequently, much of the design knowledge lies in the wetware of the engineers, and software is still only a tool to record and animate that knowledge.

Return to the detail. There is not a single knitwork encompassing the whole lever problem, but a series of smaller knitworks - control systems, design, stress, manufacture - but with common points of contact: the critical dimensions of the lever is the contact point between the control systems design and mechanical design; 3D shape links mechanical design to the finite element model and manufacturing planning. At a finer level of detail, the feature "flange" is a ridge on the lever's base-plate, and such a feature is a nexus of design knowledge: a flange is a stiffening feature for a stress analysis and a boundary feature of a machining pocket. Indeed, a useful definition of a feature is a subset of a model in one knitwork that has functional significance in a separate knitwork - here each knitwork is specific to a particular design discipline.

The tricky step is having the intelligence to know which features of each view are the jumping off points for the next step in the process. These features define the requirements for data exchange between the tools one engineer uses and those of another. And that knowledge rests to a large extent on experience - knowledge gained from designing the previous aircraft. In part, that is the experience of trying something and it not working.

...

Parenthetically, this raises an import question about how this knowledge spreads. There is a strand in engineering knowledge that is realised in amusing stories in the bar after the conference - stories about the unexpected consequences of an engineering decision - the sort of stories you can get sued for publishing. More broadly, the social interactions of the community - the tea machine, the chance meeting in a corridor, the squash ladder - are the opportunities for knowledge sharing [4.20]. Why does it take several thousand engineers several years to design a modern airliner or jet fighter? Because it involves a huge amount of knowledge, including knowledge about how to deploy that knowledge, and knowledge about how engineers acquire the knowledge to work together. All I can achieve in this book is hint at the complexities involved.

4.7.4 Intention in Knitworks

There are four elements [4.21] to explaining why a lump of engineering looks the way it does: what it is made of, how that material is shaped, how it functions, and what it is for. In the lever, manufacturing knowledge identifies the right material (BTW, material is often taken for granted here, but imagine making the lever out of jelly), the CAD model gives the shape. The finite element model - a combination of shape and material - sets the limits to the conditions for its function And the control systems model describes how the material and the shape convert the actuator's thrust into the movement of the flap.

The thing for which we don't have a good engineering model is "what is the lever for?" We can say that the lever is there to connect the actuator to the flap, but all we are doing there is invoking a wider knitwork. Purpose is about the intentions of the user - about a human - rather than about the thing itself. When we say the "the lever is there to connect the actuator to the flap" we are somewhere in a chain of intentionality - connect the actuator to the flap to control the lift on the wing so that the aircraft operates efficiently so ... I can have a cheap holiday. This is not obviously

an engineering issue except that...

Engineering is fundamentally a way of converting a requirement (I want to do something) to a specification (and this is how). If I want to build a wall or to do a smash-and-grab job, a brick can fulfil either requirement. However, the brick wasn't made with the latter purpose in mind - it has been "repurposed". Consequently, the term "design intent" is better when one wishes to distinguish this from any other purpose.

Information is the mapping from the realisation (data) to knowledge (the system model). But in making that mapping, we are pointing to the knowledge that is represented by the data - we are narrowing the use of that representation in view of the design intent of the system. Perhaps it would be better to reverse the definition: information as the mapping from the knowledge back to a data representation in a specific context (a design intent) - the information content of a message is seven bits because there are 100 elements of knowledge that can be identified, and that is because we intend to react differently in a 100 situations.

Or, start with two sets of axioms that differ only the the strings representing the terms - say "Petrol" in one, "Beer" in another. The expansions of the axioms will be identical after replacing the strings of one set with the strings of the other - the calculation "will we get to the next service station?" might look the same as "can we drink the pub dry?" It is only when a mapping is made from things in the real world to the terms in the model that the set of axioms can be called knowledge. Naming a data item "beer" to represent a quantity of petrol is technically correct but deeply unhelpful. This is not to say that OWL - or any other "ontology" language - cannot be used to represent knowledge, but that it is methodologically incomplete - its individual terms cannot fully represent design intent. The data only represents knowledge when *we know the context* and know the information mapping from context to data.

Historically, the process of systems analysis has been about the abstraction of context into facts and the inferences that could be animated as a computer program. Read any systems analysis manual, and intention and/or rationale are recorded under background documentation - it is never present in the data itself, even if it is used in the choice of inferences. While there are AI models of intention these are not part of the software mainstream[4.22].

4.7.5 Knitworks and Humans

Now, suddenly, an abstract discussion about data and knowledge needs to be enfleshed in humans and their abilities. We are told that typically a human can only keep seven things in working memory at a time. An expert works by narrowing their focus to a particular set of facts and inferences: The design engineer translates the requirement for the lever into a shape, using some basic rules about strength, stiffness and manufacturing. The manufacturing planner uses the shape as a starting point for a manufacturing plan, bringing in considerations of machine tools and costs. The stress engineer takes the shape, ignores what the manufacturing engineer and the aerodynamicist think, and focuses on the complications of predicting stress concentrations. People working together, not by sharing out a job equally, but by specialising in one part of it.

So now the job of "information sharing" focuses on the range of knitworks required and their points of contact. A knitwork becomes a proxy for an expert and their range of knowledge - facts and techniques. But the focus should not be just on the data - the numbers and letters that represent the

facts - but also on the chains of intention, which move the focus from the abstraction of data to the fleshy reality of humans in society and the "intentions" of their wetware. Information still does not exist, except as an "intention", a context in which knowledge is mapped to data.

Data interchange works because the humans involved are intending to work together. That is, we ensure interchange works by assuring that we are working on the same system in the same way - that at the point we exchange information, we are referring to the same fragment of knowledge - say, the same element in the system state vector. The technical problem is to work out the inverse mapping, and show that the way the knowledge fragment is reified is equivalent in both the sender's and the receiver's system - that when I tell you that I have 20 litres in a 40 litre tank, that it matches with your system view that says the tank is half full.

4.8 Knitworks of Knitworks

The aim of this book is to make information sharing work, and the concept of knitwork need only be elaborated as far as needed to inform that goal. But as far as the aims stated - facts used intelligently, inference based on knowledge - we've not gotten past a complicated computer program. I need to move up a level to make the case.

What has been described is a base-level knitwork - a network of facts and inferences which contribute to aspects of our models of systems or business processes. But if we look only at this level, our software processes the facts with a fixed set of algorithms, our employees follow fixed procedures - no broader intelligence or knowledge needed (our employees show these characteristics when they leave for better paid jobs). However, from this base level we can infer a second level of knitwork, the one that worked out how to build the base level. This is seen in action as the business process develops - as the card reader supplements the cash register, or the sales system updates the inventory - or even more radically as the business goes on-line and the middle managers are made redundant.

And we could go even further - the knitwork of knitwork of knitworks, where engineers devise new methods for developing systems, or business analysts techniques for business process re-engineering. Or even up another level, where mathematicians develop new mathematics to solve new series of problems. Or up again...?

Suffice it for our current treatment to stick to the base level. We share information, and that requires shared knowledge - a description of the system model or business process that the sharers are working on - which is mapped to data in their computers. Information names the mapping elements, so that any data exchanged can be put in the right part of the system/process model, and so allows the person using the model to invoke the right inferences from the information - I can use this series of numbers to reconstruct a solid, or that sequence to show my bank balance is in credit. The base level is a knitwork in so far as it benefits from the next level of knitwork, that of the skilled practitioner who validates the use of the base-level knitwork. I look at the graphs my junior has produced, frown for a minute, then ask them to rerun the simulation, checking they put in the right starting values (they hadn't).

4.9 Summing Up Backwards

I started with the idea that Knowledge and Intelligence knit together into knitworks. But working the idea through, it seems that the goal is to break apart a context - the universe in which a system or process operates - into a set of more narrowly focussed knitworks, but which can connect together though fragments of common interest.

In later chapters we consider different viewpoints on these ideas. We return to ontologies (in the computing sense) to consider the duality between a classification tree and the corresponding decision procedure - we ground the ontology in what its users find important to differentiate (trigger warning for Structuralism). Subsequently we will try to formalise the idea of *level change* as moving between two knitworks through a common nexus - rather as the shape of a lever is the nexus for connecting mechanical design with system design, stress analysis and manufacturing planning. The difference in level between a concrete situation and its expression in language is used to ground language - semantics is grounded in how words differentiate things in specific situations and the level change to class terms that provide generic descriptions of the situation.

But our immediate next step is to consolidate the idea that information is a mapping from data to knowledge by looking at Product Data Management (PDM). PDM ignores the way data represents individual pots of knowledge in order to link the pots together in the processes where the knowledge is used - as in the Maths library, where the librarians are there to help the students find the right books, rather than explain their contents.

Notes and References

4.1 Apologies to Hofstadter, D.R "Gödel, Escher, Bach: an Eternal Golden Braid" Basic Books, 1979, ISBN 978-0-465-02656-2

4.2 See, for example, Knuth, D.E. "The Art of Computing Programming", particularly Vol 3, "Sorting and Searching", (2nd ed. 1998) Addison-Wesley Professional. ISBN 978-0-201-89685-5

4.3 Tansley, D.S.W., Hayball, C.C. "Knowledge-Based Systems Analysis and Design: A KADS Developer's Handbook", Prentice Hall Europe, 1993 ISBN 0-13-515479-0

4.4 For Prince, T&H cite the Information Resources Centre "Projects in Controlled Environments - PRINCE", 1991 - web search brings up Prince2.com "PRINCE2 Methodology" https://www.prince2.com/uk/prince2-methodology Accessed 11/1/2024

4.5 Wikipedia "Card sorting" https://en.wikipedia.org/wiki/Card_sorting Accessed 24/1/24

4.6 Wikipedia "Epimenides paradox" https://en.wikipedia.org/wiki/Epimenides_paradox Accessed 24/1/24

4.7 OWL 2 Web Ontology Language Primer (Second Edition) W3C Recommendation 11 December 2012 This version: http://www.w3.org/TR/2012/REC-owl2-primer-20121211/ Accessed 25/05/2023 (Brown Banana)

4.8. Wikipedia "Description Logic" https://en.wikipedia.org/wiki/Description_logic Accessed24/1/24

4.9 The source for this was a small text book on logic from the Studium of the Dominican Friary in

Oxford, the spare afternoon was during the novice's holiday at a cottage in Devon.

4.10. Halevi/Levi - source to be dug out once my daughter collects her impedimenta that she didn't take to Osaka.

4.11 This comes from the "inverted attribute" style used when modelling ISO 10303-239 (Product Life Cycle Support (PLCS)) - an essay in its own right.

4.12 Thread on Ontolog Forum c. 2010 see https://ontologforum.com/index.php/WikiHomePage

4.13 Start with Wikipedia "JC3IEDM" https://en.wikipedia.org/wiki/JC3IEDM Accessed 24/1/24 The standard itself is brown banana - earlier versions (version 3 I think) were an easier entry point than the later versions that I worked with (v4). Even the Wikipedia page is Yellow/browning banana.

4.14 My particular gripe here arose from an overlong presentation on an array antennas, which ended up declaring victory at the Cramer-Rao lower bound. In proving there was no better unbiased estimator, it said nothing interesting about the mathematics while ignoring the poor assumptions about its application, particularly about its model of terrain reflections and spatial autocorrelation.

4.15 My notes from the fluid mechanics course, c. 1973

4.16 A good starting point is Doxiadis A. Papadimitriou C.H. "Logicomix: An Epic Search for Truth" 2009 Bloomsbury N.Y, ISBN 0-7475-9720-0 - see also its Wikipedia entry.

4.17 The original suggestions for extending ISO 10303 for data management purposes date from a paper I wrote for PDT Europe, c. 2002, but a more extended treatment is given in 4.18

4.18 Barker S. "Aircraft as a System of Systems: A Business Process Perspective" 2018 SAE International ISBN-10 076809402

4.19 Jones C.B, "Systematic Software Development Using VDM" 1990 Prentice Hall International ISBN-10 0138807337 - see also Wikipedia "Vienna Development Method"

4.20 At various knowledge management seminars in the early years of this century, the Tea Machine was rated as one of the top locations for knowledge sharing.

4.21 This is a reading of Aristotle "Metaphysics" Book V ch 2 - in, e.g. "The Basic Works of Aristotle" (McKeon, R ed) Random House, NY 1941

4.22 Wikipedia "Belief–desire–intention software model", https://en.wikipedia.org/wiki/Belief-desire-intention_software_model Accesses 19/1/26 BTW: *Rationale* provides a clue to intention, but this is a wide field and includes *meeting the "-ilities"* such as manufacturability, recyclability. While very relevant here, this would be too big a digression for the present purpose.

4.23 There are multiple dialects of OWL, which trade generality for decidability - a choice between representing Epimenides paradox or ensuring every question can be answered.

4.24 In the navy, the command to open fire is not "Fire" but "Shoot", as "Fire" sounds the alarm that the ship is on fire.