Chapter 10: Semantics - An Engineering Model

10.1 The Zig

A zig-zag zigs and zags back and forth, a less steep route than climbing directly. I grew up with the zig-zags of Bournemouth sea front, shore to cliff top, up to a wide view across the bay. This book has zigged across data, information and knowledge, zagged back to ground those concepts in practical examples, zig-zagged up again through level change, and now is near the top - a final zig to outline an engineering model of semantics and then a zag to tease it out a little more. Looking up from the beach, the paths up hidden in a patchwork of green and yellow - gorse, broom and exposed outcrops golden sand - or they were before they graded the slope and brought in goats (vegetation management). But from the top, the line of the zig-zag became clear, how one zig lead up to the next zag - how distinctions between data, information and knowledge lead to knitworks, knitworks to level change, and now level change is used to explore semantics.

Start with some caveats. I'm not going solve all of semantics in a single chapter - too many philosophers have skin in the game, too many pedants would raise wearisome objections. My aim: when exchanging information though a computer, can we ensure the terms used at one end of the exchange mean the same thing at the other? For example, I know what an apple tastes like. When I go into Ludwig's fruit shop, I trust that he has the same knowledge as me, and hence will respond to "an apple, please" by presenting me with an apple. I also trust he is genuinely running a fruit shop, not a mad philosopher who insists on debating "is this really an apple?" And so when I order "three green apples" on Ludwig's online fruit shop, I expect his system to deliver three green apples. I do not expect limes - green but not apples - nor Pink Ladies (a red apple variety).

However, if the day I order on-line, Ludwig's camera shows Pink Ladies and russets, I might expect to receive russets, whose dappled skin is more green than red (Ludwig sells only by colour, not by variety). That is, I use the terms available to choose from the fruit actually on sale, not in any expectation that the terms are an exact specification of reality, but only that they differentiate the possibilities available in the context.

But, where I use natural language examples, it is not to make general claims about natural language semantics, it is to avoid tedious explanations of a particular business process. However, business, computing also steals natural language to cue people to the right interaction. The alternative is to force users take a training course on business terms. I know about apples - their size, their shape, their colour - and assume you do too. And, conversely, I know that when I use Ludwig's "Aircraft component store", I will need to have the right technical knowledge not to confuse an ISO 9300 certified supplier with one certified to EN 9300. [Exercise for reader: explain the difference - 10.1]. But tax self-assessment is another matter - natural language used in a technical context - am I domiciled and ordinarily resident? (see next chapter).

But I do have one major problem when discussing semantics - you and your philosophical assumptions, particularly the ones that you don't know you have. [Exercise to reader: list your philosophical assumptions about language, and star the ones you don't know you have.] This will lead to differences in the way you and I use language about language even when not explicitly comparing philosophies. The examples of this chapter partly explain my theory, and partly work as therapies for your starred assumptions. Or, while standing at the top of the zig-zag, will you gaze dreamily as the yachts disappear into the sunset, or will you closely observe how their hulls disappear before their sails? And then infer that the world is actually round, not flat?

10.2 A Theory of the Meaning of Terms

This is a theory about the meaning of **terms** used in a **business process** or **system** model**.** These terms are to be shared by computer systems animating the process or modelling the system. Terms roughly correspond to entities or attributes of the information model - they are the condensation points from the knowledge cloud. Words are used to represent terms, and, although they should be used consistently within a single system, independent systems may choose different words for the same terms, use the same words for different terms, or even divide up the knowledge cloud into a different - and probably incompatible - set of terms. To share information, we need a way to systematically establish what each term means, and then check that the systems sharing information have terms that mean the same thing.

When we talk about "terms having meaning", computers have no concept of meaning - meaning cannot be "in the computer's head". It can only be exhibited by the behaviour of the system that the computer is part of. Ludwig's fruit machine has a set of boxes labelled with fruit words and a camera with a colour chart. The meaning of the order "three green apples" is exhibited in the way it delivers the fruit to the customer, not in the labels on boxes or the colours on the chart.

The theory proposed here uses four types of knitwork:

- A differential knitwork of terms covering the problem space;
- Subject knitworks, one for each term in the differential knitwork;
- Practical knitworks, one per subject knitwork, used to drive system behaviour;
- A semantic processor - the inferences that co-ordinate the use of other knitworks.

Semantics - the meaning of a term - involves the semantic processor jumping about between the first three knitworks, and is explained by how the separate knitworks work together: why the fruit machine delivers an apple, not an orange, and a green apple rather than a red one. Why, when Zane the Robot serves in the shop, we think he understands our words because he hands over the right fruit.

This theory is a mash-up from three different ideas about Meaning. It differs from one saying that the meaning of "triangle" is a reference to an abstract idea of a triangle. Such a theory looks to a subject knitwork that describes the aspects/attributes/properties of a triangle, but it does not situate it in a differential knitwork in which triangles differ from squares or circles, nor does it interpret it though a practical knitwork needed to draw a triangle.

This theory differs from a taxonomy of differential terms, in that it explicitly discusses subject knitworks in order to provide the "observables" about each subject. The differential knitwork embeds the classification process that leads from the "observables" - the properties of a subject - to the term used for the subject. This theory also provides the practical knitwork needed to exhibit the "observables", and thereby verify the classification process. However, it does not require an indefinite regression to a universal "thing" - which, as this universal root has no higher level differentia, must also be a "no thing". (Is that a Buddhist temple gong I hear?)

This theory differs from one that focuses solely on behaviour as providing meaning. While this theory uses practical knitworks to describe how the meaning of a term is grounded in system behaviour, it also links this knitwork to the differential and subject knitworks so that we know why a term is different to other terms and what "observables" are used to make that differentiation.

And, because it is a theory about systems embed in computers, it needs a mechanism to link the different knitworks together - the semantic processor (next chapter). This is itself another knitwork, but focused on inferences that link knitworks, rather than on particular facts about the world. It uses facts held by the differential, subject and practical knitworks in an intelligent way. But first, a little more detail on what constitutes each of the knitworks that the semantic processor links.

10.3 Three Linked Knitworks

The first step in modelling a business process or a system is to decide the condensation points for the fog of knowledge - the things that the computer system will talk about, the entities of the information model, the terms which will be used to enter data, the anchor points for actions. These are the elements of the **differential** knitwork. In general, modellers will not create terms for things which are out of scope of the model, and they will not differentiate two things if that differentiation makes no difference to the model. However, they will create differential terms where different courses of action follow, even though there is no such differentiation in natural language (again, see next chapter on whether a tax payer is resident, domiciled and/or ordinarily resident).

Once the concept space is split up, each term is elaborated by its own **subject** knitwork. The "observables" of a subject are just those elements which can be observed about it. In an entity/relationship model, this will include the attributes that describe an entity, its relationships to other entities and the actions that will be applied to it, or even, sometimes, the fact of its existence. The subject knitwork must be rich enough in observables that it can be classified correctly by the differential knitwork - it is the differential knitwork that embeds the classification procedure.

In practice, most computer models are sparse - they do not contain anything that is not useful to the purpose of the system. And they appear "natural" in that the elements of the model "obviously" reflect in the situation they are modelling, although in practice considerable thought is needed to make the model both obvious and adequate.

To canter through an example, in aircraft maintenance, a person doing the job must be qualified for it - must be trained and have gained key experiences. Thus we need to **differentiate** two entities: *ActualPerson* with attributes like name and personnel number (you can observe their name by talking to them) and *MaintenceQualifiedPerson*, an abstract definition with attributes including lists of *Courses* and *Experience* needed to count as qualified to do each job (observables of its **subject *knitwork***). An *ActualPerson* is identified as a *MaintenceQualifiedPerson* by a relationship between the two entities. Differentiating the two allows different lifecycles for the two entities - an *ActualPerson* may only meet the criteria for a few of the jobs listed, or, when a new qualification is created, an new subtype of *MaintenceQualifiedPerson* can ready and waiting for the first person to do the course. The maintenance system can be contrasted with the catering management system, which records the qualifications as part of each staff member record, and would not recognise *QualifiedPerson* as a distinct category of entity.

The **practical** knitwork is provided by the overall system that the computer is part of, and is designed to manifest the practical actions that follow from the thing being a particular **subject**. For example, if avionics box A54B needs a *MaintenanceQualifiedPerson.Level="3"* then the procedures of the maintenance organisation will check that the person allocated the job is actually so qualified. Indeed, the term *Level="3"* makes little sense without a list of the procedures that the person is allowed to perform.

And the term *Egg* refers to something that may be cooked, but is not something to be installed in an aircraft. The practical knitwork for aircraft maintenance comes with the meta-principle that anything that is not explicitly allowed should not be done. To a human, installing an egg in an aircraft is "obviously stupid", but "obviously stupid" is not a concept built into a computer. Nor is a *MaintenanceQualifiedPerson* allowed to boil an egg in the canteen - for that you need a catering qualification, or at least to have done a food hygiene course.

Hence, three knitworks, each of which contribute differently to the total picture needed to fully explain the meaning of a term.

10.4 Grounding Example - Colour Terms

In elaborating the need for three knitworks, I start by discussing colour terms, a subject unfamiliar to most readers except through the folk ontology of their native language. One might expect colour terms to have a reasonably consistent meaning, since human eyes all work in the same way [Exercise to the reader: list the caveats that apply, stop after two pages]. However, from noting phrases like "the wine dark sea" in Homer's *The Odyssey*, the British Prime Minister and Greek scholar William Gladstone prefigured a whole field of anthropology - the naming of colours.

To crudely summarise one five hundred page book on the subject [10.3], people went out across the world with colour charts and asked people what colours they had names for and to point to the colour on the chart (if you have a Munsell colour chart, do try this at home). Where people have equivalent colour terms, they would point to similar colour blocks, but with some variability in the exact shade they chose. Blue, for example, showed less variability than green. However, taken overall, the areas of the chart that people pick out as being typical of a colour are small compared to the areas not selected. That is, people mostly agree what basic colour terms mean.

BUT (a big but here) languages vary in the colours they name, ranging from only four in the Grue languages (dark, light, red and "grue", with "grue" covering both green and blue), to eleven basic colour terms (black, white, red, green, yellow, blue, brown, purple, pink, orange, grey) [10.4]. Or possibly twelve - studies on reactions times of Russian speakers indicate that for them, their different words for dark and light blue are basic colour terms [10.3, pp 75-107], perhaps like red and pink in English. Note also that pink - the name of a flower - has only been used as a colour term in English since the early mediaeval period, and the same is true in French for "rose", their word for pink. That is, there are groups of people - linguistic communities - that haven't felt the need for "pink", "orange" or "grey".

In terms of knitworks, if we start with a colour chart, the differential knitwork for a language divides it up into a number of distinct regions - usually between four and twelve. The subject knitwork for each region picks out a colour sample as the "characteristic colour" for the term and links it to a language specific colour word. The practical knitworks allow users of a language to name a colour seen, or pick out a paint or crayon associated to a colour word - the sea may be described as "blue", "bluey-green" or "wine dark".

While the anthropologists have put in considerable effort gathering evidence on the naming of colours, I have not seen discussion about why people need colour terms at all. One hint is that traditional Japanese colour terms were based on vegetable dyes but from the end of the 19th century

these have been replaced by the names from chemical dyes - including "orongu" and "pinku". My hypothesis is that to agree a colour term, users must have some consistent reference - for example, a flower (pink/rose) or fruit (orange), or a dye or a paint technology that produces consistent colours. In the case of manufactured colours, the ability to produce consistent colours connects to social drivers such as fashion and branding, so we get culture specific colours such as "pillar box red", "British racing green" or the brand purple of a chocolate bar wrapper. But, when we repaint a room, we start with a rough idea - a basic colour. We then look through a paint chart to find the exact shade we want, and then memorise its marketing name so we can pick out the right tin in the shop. My favourite marketing name was "witchcraft" [exercise for the reader - get a sample of friends to independently pick out which shade "witchcraft" might refer to].

And, of course, there are complications. Hair colours are "blonde", "brunette"or "ginger" rather than yellow, brown or orange (or red if you don't have orange as a colour term). And over complications - precise colour terms such as "Gainsboro" in the HTML colour space [10.5] or "light red" in the Java programming language [10.6] provide for consistency in programming code, but are not useful terms when talking to a user, especially as different screens may render them as different shades. One cannot expect users to name colours precisely and consistently, nor can one expect them to look up "your colour chart" rather than their own "internal reference". The best you can hope for is that they use basic colour terms, possibly with a qualifier such as "dark" or "pale". And also hope they are not colour blind.

10.5 Grounding Example - Part Number

A *PartNumber* is the number of a *Part*. Simple? Yes and no. When extending the STEP standard with PLCS [10.7], it took several months of meetings in order to agree what the variations on the concept actually meant and what they should be called. Although once the issues were combed through, the answer was fairly simple. The problem had lain in condensing the knowledge fog down to a few nucleation points so we could state the issues in consistent terms. But since you are unlikely to want to work through all of the detail, what follows is a cut down, bowdlerised version of the final agreement, using natural language cues rather than the actual technical terms of the standard.

Start from the conclusion - Part numbers are one sub-category of *ProductIdentifier*:

- *PartNumber* - the identifier for the **design** of a part;
- *PhysicalPartNumber* - the identifier of a **physical** part, consisting of the *PartNumber* from design and the production *SerialNumber*;
- *StockNumber* - the number of a **design specification**, usually in some public context;
- *CatalogueNumber* - also the number of a **design specification** but in a private context.

In aerospace, the *PartNumber* has three components:

- *Number* - the number allocated to the part **design**;
- *SupplierCode* - the identifier of the **organisation** that allocated the *Number*;
- *CodeAllocater* - the identifier of the **organisation** that allocated the *SupplierCode.*

The source for the aerospace numbering standard is ISO 21849 [10.8]. This defined three codes for *CodeAllocator:* CAGE (an identifier set by US government agencies), Dun and Bradstreet (the allocator of D-U-N-S number) and their European equivalent. Having these three codes defined in the numbering standard avoided an indefinite regression of naming contexts.

Getting into the details, manufacturers allocate their own part numbers in the design process, but they do so independently of each other, so the *PartNumber* must be put in the context of the supplier who allocated the number - the attribute *SupplierCode,* with *SupplierCode* unique in the context *CodeAllocator.*

In design, *PartNumbers* are unique - and if you are fitting a part to a product, uniqueness ensures that the right part is used. However, in moving from design to production, many physical parts are made to the same design, and so, once it is manufactured, each physical part is stamped with a serial number. Many maintenance procedures are based around checking parts for wear and fatigue and *PhysicalPartNumber* allows the tracking of each individual part. Exceptions include small items like rivets, and "indefinite" items like paint or sealant.

In conversation, the term "part number" is used during the design process to mean *PartNumber* but in production and support it can mean either *PartNumber* or *PhysicalPartNumber* depending on context - and the people involved generally know what the context is. Conversely, when that context is missing, say in a computer interface, the naming needs to be explicit.

A *StockNumber* identifies a range of interchangeable parts, which allows a large organization - such as an army - to buy the "same thing" from several different suppliers and use which ever is in stock. A *CatalogueNumber* has a similar function, but is used within the context of a single manufacturer, so that they can use which ever parts are available on the production line. For consistency in their own systems, *CatalogueNumbers* may have the same format as the manufacturer's internal part numbering system, and in the design world are treated as a "part number".

That only leaves *ProductIdentifier* - a generic concept partitioned into four classes as described above. The only people who explicitly use *ProductIdentifier* are the information modellers, who use it to create consistent design patterns across the different subcategories. This makes it much easier to integrate information from design, manufacture and support processes. It is also provides a starting point for deciding if additional subcategories are needed. For example, *SlotNumber* is a support concept used to specify a position where particular physical parts may be fitted - most cars have four *Slots* where tyres can be fitted (the wheel hubs), the tyres being specified a *StockNumber* which may point to the *PartNumbers* of several different tyre manufacturers. You will probably not care about the *PhysicalPartNumber* of each tyre, but it might be used by the manufacturer to collate failure reports, and so identify a below standard batch of tyres.

In terms of the knitworks, there is the abstract definition of *ProductIdentifier* which is illustrated by the differential knitwork of *PartNumber*, *StockNumber* etc. These have their own subject knitworks both defining the concept and generating real world examples. For example, "*PartNumber* is the number of an aircraft part" (design context) and *PartNumber* = ("PI2345-101", "K0999", "CAGE").

The subject knitwork may also link into sector and company procedures. For example, configuration management [10.9] requires that any update to a part design that changes fit, form or function must be allocated a new *PartNumber.* However, in the aerospace sector, if a physical part is repaired it requires both a new *PartNumber* for the repair design and to retain its old *PhysicalPartNumber* identity for lifecycle tracking. And, just a reminder to the geeks who say "why not use a web URL", when you are under fire on a remote desert landing pad, you may not have web connectivity or a working bar code reader but you still need to get your helicopter in the air before the insurgents put too many holes in it (or you) [10.10].

The implication for data exchange is that the exchange standard must identify the different types of *ProductIdentifier* with appropriate descriptions to map them to the different uses, together with a formal description of the data elements involved. The programmers implementing the exchange must then map the terms used in the standard to those used within the company specific processes. That is, for data exchange:

"term in Organization A" *mapsTo* "Term In Standard" *mapsTo* "term in Organisation B"

The **users** (stress users, not IT geeks) must then check that the right identifier is being used in each process and that the process is still working correctly. For example, a maintenance procedure needs a *PhysicalPartNumber* not a *PartNumber* or a *StockNumber* if it is to record which particular part has been inspected for fatigue.

On a personal note, I found writing information standards hard work, both to find the fine cracks that indicate that a term has similar sounding but different usages, and then turning user resistance into acceptance of finer distinctions - this often involves making up new terms which don't trigger the baggage of existing terms and then laying out precise distinctions in the definitions.

10.6 Grounding Example - What is a System?

The first known usage of "system engineering" dates from 1950 [10.11], although it did not become a separate discipline until the 1990s. In the intermediate period, systems engineering occurred mainly in the context of developing embedded subsystems - radar systems, navigation systems, communications systems, and so on. In the 1970s, my job title was "radar systems engineer" and I modelled the performance of radars - behaviour modelling rather than building hardware. Commonalities in the methods - commonalities in the knowledge of design methods - condensed into the idea of "systems engineering" as a specific discipline. However, this understanding was not common when STEP was developed, so when *System* turns up in STEP (ISO 10303) as a subtype of *Product*, it refers to a functional model of an embedded system, as distinct to a physical model of the parts used.

In the Wikipedia entry on "Systems Engineering", the section "Systems engineering topics" includes a number of similar definitions of system, none of which is capable of differentiating *System* from *Product* [10.12]. For example, there are some *Products* that fall outside the definition of *System* - systems engineers don't usually design nails. Indeed, they don't even design bridges, which a structural engineer might regard as a system of beams and columns linked to enable the behaviour "not falling down". In some quarters, the overlaps between systems and general engineering practice have led to a land-grab by systems engineers. For example, general engineering disciplines such as change management have been claimed as being in the scope of systems engineering, rather than systems engineering being a user of change management. While some customer requests may lead to a change in what a system does (systems engineering change), changes can be made for other reasons, such pushing back the manufacturing schedule (production engineering) or using a different colour of paint (physical BOM).

In STEP, the situation is complicated by the way that the scope of *Product* has drifted over the past forty years. In 2012 there were some eighteen subtypes of *Product* including *Part, System* and *Slot*. In practice, one might view a *Product* as both a *Product* with a physical product structure, and as a *System* if it needs a separate product structure relating to behaviour. For example, a central heating

system has a boiler, pipes and radiators. In the physical product structure, these are listed as components of particular rooms, but a systems product structure acknowledges the way these components are linked by the flow of water, not merely by physical proximity, and a "systems test" will check that there are no leaks - no gaps between the physical components in one room and the next.

That is, there is an abstract concept of a *System*, but viewed in isolation, the concept has not been fully condensed from the cloud of knowledge. Some practical examples of systems - and of things that are not systems - have not been fully reflected in some abstract definitions; "System" has not been fully situated in a differential knitwork. In a differential structure, one might identify a system as a view on a product which has its own functional product structure separately from the physical product structure or that used by maintenance planning. Thus for a bridge, the function of beams and columns is intimately related to their position in the bridge - the physical and functional product structures coincide - but for central heating, the siting for radiators and pipe work is quite flexible, and moving a radiator often has little effect on the working of the whole system.

Or perhaps, when I design a system, I may start from an abstract, functional design, and later realise that design in physical equipment - I know for my hi-fi system I want an amplifier, speakers and CD player (functional elements) and then move on to buy the actual components including the cables to connect them (physical elements). By contrast, when I build a bookcase, I buy uprights and shelves (physical elements) and then when constructing it, I space the shelves (as functional elements) to fit the heights of my books.

10.7 Ludwig's Fruit Machine Revisited

Although Ludwig's fruit machine is a physical product, we are not concerned with it as a metal and plastic object, but as a system that sells fruit - we take a systems view, but here we only consider just a few system elements: buttons, fruit, and colours.

Buttons are both physical products with a part number and also systems in their own right. In the fruit machine they are components for the order subsystem: in the *FruitSelection* subsystem they are labelled "apple", "orange", "pear" and "banana", while "red", "green, "yellow" and "brown" appear on the *ColourSelection* subsystem. We can abstract each of the *FruitSelection* and *ColourSelection* subsystems to a memory element which takes the values zero to four - at the beginning of each purchase cycle, the value zero means that no button has been pressed, and then the memory element is set when a button pressed, say 1 for "apple", etc.

Each button subsystem has an associated practical knitwork. The *FruitSelection* subsystem sets the value of the entity *Fruit*, while the *ColourSelection* subsystem sets a value for the attribute *Colour*. The practical knitwork for colour also includes the camera subsystem, which generates one of five colour assessments - red, green, yellow, brown or "ugh?" (i.e. any other colour such as blue or black). Meanwhile, the practical knitwork for *FruitSelection* consists of a grabber that transfers fruit from a box to the delivery drawer, based on the information mapping from *FruitSelection* to a box containing fruit. The operating procedures of the system are there to ensure that the mapping is consistent with the fruit - i.e. to get Ludwig to put the fruit in the right box.

The generic knitwork for fruit includes knowledge about how to store fruit and how to dispose of it when it goes off. However, we are most interested in the differential knitwork for classifying fruit

using each fruit's specific subject knitwork. The knitwork for apples includes details such as their being round and having a core, while the one for bananas includes "curved", and the colour changes that occur as they ripen. Shape is enough to differentiate a banana from an apple, but colour and skin texture are used when differentiating an apple from an orange.

The link from the subject knitwork for apple to the differential knitwork for fruit takes us up a level. That is, there is knowledge specific to a type of fruit such as an apple, and general knowledge about what to look for when comparing one type of fruit with another. This involves a level change, as discussed in the previous chapters. One could create entirely independent knitworks for each type of fruit, but then knowledge of, say, the role of pips would have to be researched afresh for each fruit. Indeed, we would note that Pink Ladies, Granny Smiths and russets look and taste different, and ask the question whether there is any common category - call it "apple" - which we could use to consolidate our knowledge and read across from one variety to another?

Such abstractions and level changes are not within the boundaries of the physical machine, but are needed within the boundaries of the system as a whole - they are found in the operating procedures which require Ludwig to put the fruit in the right box and label the buttons correctly. That is, the physical system of the vending machine is a mechanism inside the business system of selling fruit. Consequently the knowledge of its users and operators is inside the boundary of the total system, and the system must be designed within the constraints of that knowledge. Alan from England might guess that a machine in Turin would label a pear as "pera", but might be confused if it is labelled "birne" - German being Ludwig's native language.

One might think that genetics might come within the subject knitworks both for fruit (the general class) and specific types of fruit, such as apples or bananas. But Alan does not carry a gene sequencing machine when shopping for apples - semantics always occurs within the bounds of the system interpreting each term. Ludwig does not sell tomatoes (usage indicates it is a vegetable), but will happily sell strawberries, though a botanist might quibble that they are really "aggregate accessory fruits". Such quibbles are only useful in a larger (semantic) context.

10.8 Some Unwarranted Speculation

So far the theory has accounted for the terms with which semantics are exchanged; the knitworks which provide the formal structures needed to go from a term to its meaning and the mechanisms by these structures are exhibited - the systems models and business processes. That leaves us in need of one further explanation - why do we want to create such mechanisms?

Imagine you are a small, fury creature that lives in colonies, that forages above ground but has a burrow to hide away in, and is a favourite snack of creatures further up the food chain, such as foxes, snakes or eagles. If you spot a fox, your best bet is run down into your burrow, as they can't get to you there. Snakes can follow you down the burrow but are slower, and you just have to be vigilant to scamper out of their way. The best way to avoid being eaten by an eagle is to stay stock still, because they drop faster than you can run away and they are very good at motion detection. Watching for potential predators takes concentration and gets in the way of foraging, but as you are a member of a colony, you can set look-outs, and agree a convention - squeek for a fox, squeeeeeeek for a snake, and sque-sque-sque for an eagle. That way you will know when a predator is nearby, and what is the best strategy for avoiding it.

That is, I suggest that the motivation for creating a term is to say something about the world in which the system operates and invoke the appropriate practical response. If Alan wants to buy green apples, the fruit machine must differentiate apples from pears, oranges and bananas, and differentiate green apples from red ones (and should not try to sell brown ones). The shop does not sell clothes or computers, so needs neither the vocabulary or the knitworks for them - the machine does not need the cost of the extra buttons, nor the compute power to tell the difference between an apple and a computer. And as it doesn't sell grapes, blueberries or blackberries, it can get away with a restricted colour vocabulary.

Or, it takes about 25% of our energy to run our brains, so we want to make our language as efficient as possible - we are only going to invent terms where they convey enough to make a practical difference. It is just that we are complex creatures, who need a vast range of words for practical things - not just apple, orange, and banana, but also spade, shovel, and large-hadron-collider (though not many need to differentiate a spade from a shovel).

This model of semantics is designed to take the meaning of meaning out from the dingy lecture theatres of philosophy and into the bright lights of engineering labs. This is not yet a practical mechanism, a numerical recipe that one can take away to cook up a cool, universal app. But it is a stepping stone from the current islands of integration to, say, a single Internet of Things - where, assuming it has something to say, your cooker can talk to your fridge, even though they are made by different companies. Or rather, if your fridge needs to talk to your cooker, we need to provide them with an appropriate vocabulary linked to the actions they can undertake. Coolant loss has no meaning for a cooker, nor "Grill on" to a fridge.

10.9 Conclusions

The most obvious conclusion is that Dr Johnson, Noah Webster, and William and Robert Chambers are essentially liars, as are most editors of dictionaries. The meaning of a word is not given by a scatter of near synonyms. Rather, it is a cue into a subspace of knitworks, defined as much by what it is different from as what it directly refers to, and also by the behaviour of systems that act in response to it. Dictionaries, in which the editors have sought out and compiled commonly used alternatives for each term, are the precursors of large language models.

But, to return to the driving principle behind this book, the function of data exchange is to share knowledge. Not to teach knowledge - not to expand the knitwork of the receiver. Rather, to instantiate generic knowledge to a particular situation, a situation known by the sender and knowledge needed by the recipient. When I use a web site, I do not use it to learn how to buy something, rather, I share my knowledge of what I want to buy with the seller, and also my banking details, which they then use to share the price with me and subsequently the fact of my purchase with my bank. But this only makes sense in a world where I know what a 100mm coach screw is, where the seller has the product to sell and that I trust it meets my specification. And where they know how to ship it and to bill me. And also where the bank knows what to do with the message element "£5 please".

If one examines Ludwig's fruit machine, all the knowledge - knowledge about fruit types and their colour, and about money - is abstracted away to labels on buttons and into the operating procedures on how put the right fruit in the right box. When we buy fruit from Ludwig in person, all that knowledge is "in his head". And we must trust that that knowledge in Ludwig's head is somehow

embodied in the machine - a trust which is reinforced every time it gives us the fruit we asked for - every we time we validate the term "apple" with our teeth. (Fortunately we were out of the country when Zane loaded the machine with wax display apples).

A prerequisite of this sharing is that we have the same system of differential terms, the subject knowledge about each term is similar, and the systems that embeds the terms exhibit the appropriate behaviours in response to those terms. If, for a moment, we restrict our attention to the computer systems involved, and further restrict our attention to the informations models (as a proxy for knowledge) then we can exchange information between systems if the information models "look the same" in situations of interest, and the operations of the software in response are consistent with expectations. The strongest criterion for "look the same" is that they are isomorphic (maths word) although weaker criteria may apply - but the technicalities of "look the same" need not concern us here. Moreover, the criterion applies to the information model not the data model which reifies the information model - otherwise we'd never exchange between similar applications as each application uses its own, private data model.

The cheap and cheerful test of information model compatibility is to send data from one system to another, then send it back again, and check that nothing has changed. A more expensive test of knowledge sharing is to exchange data about a representative situation, and see if the consequences of continuing the process are unchanged. Say I tell my robot Zanetta to buy three green apples - does she get same outcome as I would if I went to Ludwig's fruit shop in person, or used his fruit machine, or his on-line fruit store? [Exercise to reader: identify a full set of tests to cover both necessary and sufficient conditions for model compatibility.]

I think I have now outlined an engineering model of semantics. The next chapter will exercise the theory a bit further. But for now, raise a toast to Dr Johnson, the inventor of the Large Language Model.

Notes, References and Hints.

10.1 When I did this check on-line in early May, the AI summary confused EN 9300 Long Term Archiving, with ISO 9300 on sonic nozzles - I commented on this on LinkedIn. Repeating the search a week later (27/05/2025) gave no such confusion, although, confusingly enough, the sonic nozzles standard is referenced as BSI EN ISO 9300. But note also (27/06/2025) that my son pays for an AI upgrade, which got the distinctions right.

10.2 One section I chose to omit covered pattern recognition and its relationship to differentiation. Both recognition and differentiation are based on correlating features of the thing to be recognised/classified with "an ideal" and scoring the result - convolution with a matched filter if you want the technicalities. In a recognition process, we trust the filter and say bingo if the score crosses the recognition threshold. In a differentiation process we run parallel recognition filters, and choose the one with the highest score. A formal decision process quantises such algorithms to logic rather than use the arithmetic of convolution. If we could reflect on the details of out own thought processes, we would probably find that we use both recognition and differentiation, but prioritise the lowest cost (leap to a conclusion) except when we don't trust the result.

10.3 Maclaury, R.E., Paramei, G.V., Dedrick, D. Eds "Anthropology of Colour", John Benjamins Amsterdam/Philadelphia 2007, ISBN 978 90 272 3243 4 (Yellow to brown banana)

10.4 Wikipedia "basic colour" terms https://en.wikipedia.org/wiki/Basic_Color_Terms accessed 28/3/2025 - saving you £110 if I referred you to 10.3

10.5 "Color names" [sic] https://htmlcolorcodes.com/color-names/ accessed 27/05/2025 - Gainsboro is a shade of grey.

10.6 "Colors in Java" [sic] https://teaching.csse.uwa.edu.au/units/CITS1001/colorinfo.html, accessed 19/02/2025

10.7 PLCS - "Product Lifecycle Support, ISO 10303-239 https://www.iso.org/standard/78832.html is the canonical reference (Very Brown Banana) but better to search for a user-friendly introduction.

10.8 ISO 21849 "Aircraft and space — Industrial data — Product identification and traceability" - this book is based on the 2006 version as I'm not paying £160 for the 2022 version. (Brown Banana)

10.9 See Wikipedia or SAE International "Configuration Management Standard EIA649C" https://www.sae.org/standards/content/eia649c/ accessed 27/05/2025 (Brown Banana)

10.10 Based on a private conversation with an RAF Squadron Leader.

10.11 SEBoK "A Brief History of Systems Engineering" https://sebokwiki.org/wiki/A_Brief_History_of_Systems_Engineering 7/02/2025

10.12 Wikipedia "Systems Engineering" https://en.wikipedia.org/wiki/Systems_engineering 8/04/2025