Software, Engineering, Artefacts, Language

Software

The Merriam Webster dictionary defines software as: *the entire set of programs, procedures, and related documentation associated with a system and especially a computer system; specifically: computer programs.* This definition contains spurious complexity – and therefore serves as an excellent example of software (it could have been written by a software engineer).

The Oxford American Dictionary defines software as: *the programs and other operating information used by a computer.* This definition describes software as *information* used by a computer – which is technically correct (it could have been written by a mathematician).

I would argue that both definitions ignore a key aspect of software, and that the second definition is the better candidate for further elaboration: *the programs and other operating information used within a computer and in the interactions between systems and/or humans.*

Looking up the dictionary definitions of two core activities associated with the production of software highlights further issues:

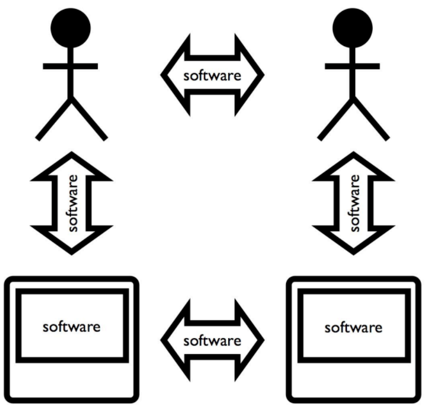
- to code: *express (a statement or communication) in an indirect [or euphemistic] way*
- to model: *devise a representation, especially a mathematical one of (a phenomenon or system)*

Considering that coding is commonly used as a synonym for programming, we do have a problem.

Observation: *Coding* can be viewed as having to deal with someone else's representation (program notation or otherwise). Coding happens when we work with third party implementation technologies (hardware or software) and when mapping to such technologies in a transformation or template language.

Observation: *Modelling* can be viewed as working with a direct representation of the purpose that humans associate with a system. Modelling is what happens when we capture knowledge in a domain specific notation that is grounded in established domain terminology.

Proposed definition: *Software* consists of all the models and code used within a computer and in the interactions between computers and humans.



Proposed goal: Software production techniques should promote modelling, and should aim to minimise the amount of code that humans are directly exposed to.

Engineering?

Most software projects try to deliver results via two simplistic transformations:

Stuff
➔ Solutions / products
➔ Monetised value

For most organisations "stuff" is the most accurate term to describe the state of their assets - there is no reliable or consistent terminology for their "stuff". To a significant degree business processes consist of series of hilarious chains of misunderstandings peppered with ad-hoc feedback loops that are needed so that at least some fraction of the "stuff" leads to revenue. Teams that are concerned about rigorous engineering sometimes resort to a set of three stages of transformation:

Stuff
➔ Data structures
➔ Solutions / products
➔ Monetised value

Most organisations have no concept of proper *terminology*, they have no idea how to represent and modularise deep *knowledge*, and no understanding of the journey it takes to implement and operate the full "stuff" transformation chain:

Stuff
➔ **Terminology**
(the Semantic Web standards don't provide the solution)
➔ Data structures
➔ **Information**
(data structures must be validated by sufficient numbers of instances)
➔ **Knowledge**
(concrete notations, abstraction, heuristics)
➔ Solutions / products
➔ Monetised value

The transformation process above involves many critical aspects: *linguistics, listening, validating by example, abstraction, collaboration, creative exploration, filtering, measuring, simulation, ...* Engineering mainly comes in at the transition from Knowledge ➔ Solutions / products. I cringe at the attempt of labelling the whole chain as "software engineering". In software production (and from what I've seen, things in classical engineering are often only marginally better), many of the really important steps are skipped or skimmed over, and the situation is not going to improve by placing more emphasis on the engineering aspect.

Observation: If software is information (models and code), then subject matter experts in various disciplines produce the vast majority of software, and software developers only produce a small fraction of software.

It is no coincidence that we have mature and highly scalable database technologies for handling operational data (including support for distributed transactions) and that we still rely on the arcane file abstraction (operating system level) for managing program specifications.

Observation: Distinguishing between "us" (software developers) and "them" (software users) is highly counter-productive. We are all "computer users", and all of us consume and produce information (software).

Proposed goal: All software changes should be as low-risk as a database transaction. Program specification changes are simply very-long-running transactions, and their duration is artificially inflated by the arcane mechanisms that we currently use to perform such transactions.

Artefacts

Commonalities that can be exploited to simplify the production and consumption of software:

- Humans interact by exchanging *artefacts* (or work products), and therefore all software must be modularised in terms of artefacts.
- Some software used by humans predates computers, and in many cases this software is recorded in more or less standardised notations – these notations provide an important stepping-stone.
- Humans have natural cognitive limitations that dictate sensible upper limits for the size of individual artefacts.

Observation: To date software producers have neglected the role of artefacts as natural units of work, and as a mechanism for defining the boundaries of areas of knowledge.

Methodologies for Enterprise Architecture Management, Project Management, and Software Development and related tools are often sold and deployed as silver bullets, with the expectation that the artefact templates associated with the methodology provide an effortless path to organisational enlightenment.

Real value is not created by off-the-shelf methodologies, but by consciously focusing on the specific needs of the organisational context, and by automating the conversion of pure domain knowledge (business level decisions) into valuable products. In many organisations very little analysis is needed to quantify a business case for replacing significant parts of requirements specification and application software development activities with a set of custom-tailored artefact definitions, and appropriate transformation tools. Software engineers should never encode decisions that may change once or more each year (such as workflows, business rules, or pricing strategies). All such decisions must be changeable directly in artefacts created by subject matter experts and decision makers. To get started, the term artefact requires a practical definition:

Proposed definition: An *artefact* is a container of information that

- is created by a specific actor (human or a system)
- is consumed by at least one actor (human or system)
- represents a natural unit of work (for the creating and consuming actors)
- may contain links to other artefacts
- has a state and a lifecycle

Work product templates associated with a methodology or a supporting tool usually conform to this basic definition of *artefact*. However, in order to be suitable as a front end for automation and transformation programs, artefacts must fulfil a number of further requirements:

Proposed definition: A *software artefact* is an artefact that meets the following requirements:

- It is created with the help of a software program that enforces specific instantiation semantics (quality related constraints)
- The information contained in a software artefact can be easily processed by software programs (in particular transformation languages)
- Referential integrity between software artefacts is preserved at all times with the help of a software program (otherwise the necessary level of completeness and consistency is neither adequate for automated processing nor for domain experts making business decisions based on artefact content)
- No circular links between software artefacts are allowed at any time (a prerequisite for true modularity and maintainability of artefacts)
- The lifecycle of a software artefact is described in a state machine (allowing artefact completeness and quality assurance steps to be incorporated into the artefact definition)

- The events consumed and produced by the artefact state machine are available for processing in software programs (allowing transformations to be triggered at all necessary points in time to keep artefacts in sync with all derived artefacts)

Tellingly, none of the most commonly encountered artefacts (emails, text documents in various technological formats, UML models, database rows, source code, in-memory objects) meet all of the above criteria. Fundamental practices and patterns for software production must directly relate to the production of software artefacts. Software quality criteria must relate to the needs of software artefact producers and consumers. Since most software producers are not software engineers, it is invalid to assume that most software artefacts are produced with the help of the typical tools that are bundled in so-called integrated development environments (IDEs).

Language

Proposed goal: The set of software artefacts of any non-trivial configurable system include artefacts that are templates for the production of further artefacts. Software producers need to agree on:

- a practical notation for decorating any software artefact with instantiation semantics, such that the result is a template for software artefacts produced by down-stream roles in the value chain.
- a software program that acts as the reference implementation for software artefact instantiation semantics.

Note that the goal above is based on the practical needs encountered in the work with numerous software intensive businesses. The proposal allows for unlimited recursive definitions of software artefact templates. Attempts by programming language designers and software tool vendors to over-simplify reality by insisting on a limited number of instantiation levels have failed.

It is instructive to watch discussions of subject matter experts in various disciplines, and to take note of the symbols, illustrations, and terms that are recorded on flip-charts and white boards. Communication of deep knowledge between subject matter experts almost always requires the use of specialised terminology and symbols.

The discipline of mathematics is a good example of a mature domain for which a rich set of domain-specific symbols and notations has evolved over hundreds of years, and continues to be used. Basic mathematical concepts such as sets, induction, graphs, groups, algebras etc. provide an excellent foundation for the specification of software artefacts.

It strikes me as odd that software engineers and computer scientists have in many cases reinvented new names and symbols for well-established mathematical concepts. Why do we need “inheritance”, when all we are talking about amounts to supersets and subsets? Why do we need “classes” and “types” and “objects” and “instances” when the classification concept can be articulated in terms of ordered pairs? …

Observation: The artefacts that subject matter experts produce when not shackled to a software engineering “methodology” tend to be neither specifications in a general purpose programming language nor do they tend to be long-winded stories expressed in natural language.

Subject matter experts tend to rely heavily on domain specific jargon and related notational short cuts. In most disciplines the domain specific jargon and notation is nowhere near as standardised as the mathematical language, and often the jargon contains many terms that are only familiar to a specific team or organisation.

Proposed definition: *Software artefact design* is the emerging discipline of recording useful domain specific jargon and nudging the jargon into a shape where ambiguities are resolved, and where the artefacts articulated in the jargon meet the proposed definition of a software artefact.

Related material:

http://www.slideshare.net/jornbettin & http://www.industrialized-software.org/kiss-initiative