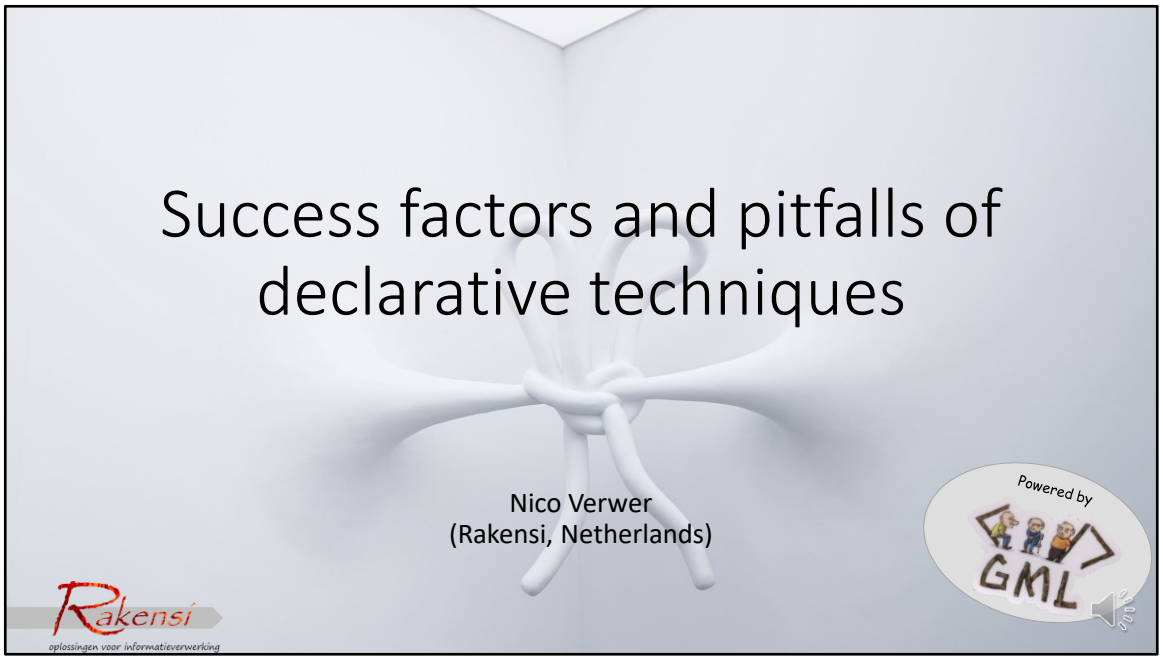
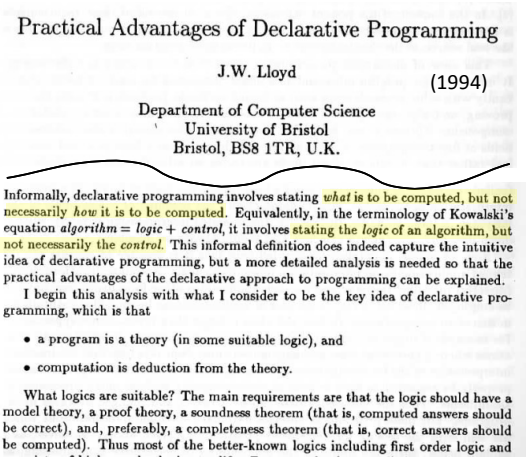


Declarative programming may not seem to be widespread, but it has been used for several decades in some areas of information technology. Examples are SQL for querying databases, and regular expressions and grammars for text analysis. More recently, domain-specific languages have been used to take advantage of declarative methods, with varying degrees of success. What can we learn from the successful applications of declarative programming? And perhaps more importantly, is there something that failed applications have in common? In this presentation we will look at what makes a declarative technique successful. We will also look at two pitfalls that the author has encountered many times: genericity and reification.

What is “Declarative”?



Practical Advantages of Declarative Programming

J.W. Lloyd (1994)

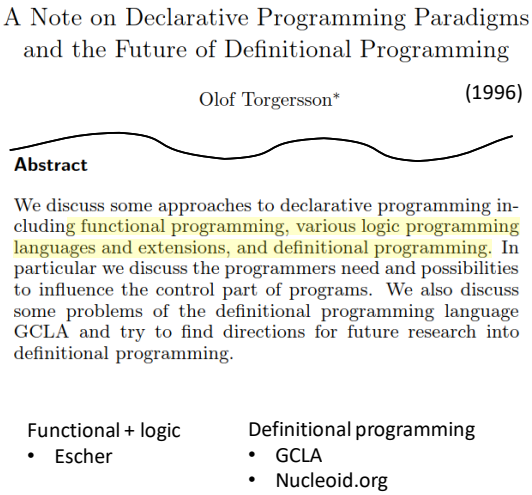
Department of Computer Science
University of Bristol
Bristol, BS8 1TR, U.K.

Informally, declarative programming involves stating *what* is to be computed, but not necessarily *how* it is to be computed. Equivalently, in the terminology of Kowalski's equation *algorithm* = *logic* + *control*, it involves stating the *logic* of an algorithm, but not necessarily the *control*. This informal definition does indeed capture the intuitive idea of declarative programming, but a more detailed analysis is needed so that the practical advantages of the declarative approach to programming can be explained.

I begin this analysis with what I consider to be the key idea of declarative programming, which is that

- a program is a theory (in some suitable logic), and
- computation is deduction from the theory.

What logics are suitable? The main requirements are that the logic should have a model theory, a proof theory, a soundness theorem (that is, computed answers should be correct), and, preferably, a completeness theorem (that is, correct answers should be computed). Thus most of the better-known logics including first order logic and



A Note on Declarative Programming Paradigms and the Future of Definitional Programming

Olof Torgersson* (1996)

Abstract

We discuss some approaches to declarative programming including functional programming, various logic programming languages and extensions, and definitional programming. In particular we discuss the programmers need and possibilities to influence the control part of programs. We also discuss some problems of the definitional programming language GCLA and try to find directions for future research into definitional programming.

Functional + logic
- Escher

Definitional programming
- GCLA
- Nucleoid.org

Before we continue, let’s briefly restate what “declarative” is. Here are two papers from the 1990’s.

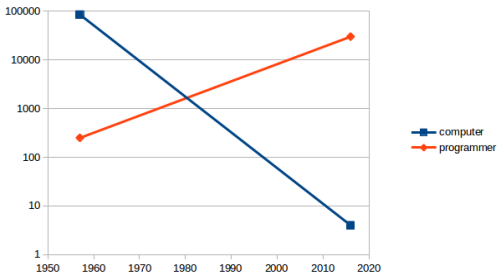
“Declarative programming involves stating _what_ is to be computed, but not necessarily _how_ it is to be computed.”

Functional programming, logic programming, definitional programming (more general).

Nucleoid is an open source (Apache 2.0), a runtime environment that allows declarative programming written in ES6 (JavaScript) syntax. Since statements are declarative, the runtime provides logical integrity, plasticity and persistency as hiding technical details.

Complexity in software

- Programmer *effort* is proportional to *complexity*^{n} where $n > 1$
  - Halstead metrics
  - Boehm, COCOMO
- Boehm predicted that software costs would overwhelm hardware costs.
  - 1973, “Ada - The Project : The DoD High Order Language Working Group”
- Moore’s switch
  - Steven Pemberton (2019)
  - Compared to the cost of a programmer, a computer is almost free.



Relative costs of computers and programmers, 1957-now (from Steven Pemberton: Declarative vs. Procedural)

In an important 1973 report entitled "Ada - The Project : The DoD High Order Language Working Group" to the Defense Advanced Research Projects Agency (DARPA),[7] Boehm predicted that software costs would overwhelm hardware costs. DARPA had expected him to predict that hardware would remain the biggest problem, encouraging them to invest in even larger computers. The report inspired a change of direction in computing.

Essential vs. Accidental Complexity

- Frederick P. Brooks, *No Silver Bullet* (1986)
  - "we cannot expect ever to see two-fold gains every two years" in software development, as there is in hardware development
- Essential complexity:
  - caused by the characteristics of the problem to be solved and cannot be reduced.
- Accidental complexity:
  - problems which engineers create and can fix
  - difficulties due to the chosen software engineering tools
  - difficulties arising from the technical solution
  - lack of using the domain to frame the form of the solution

No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the *essence*; *accidental* tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Does it have to be hard?—Essential difficulties

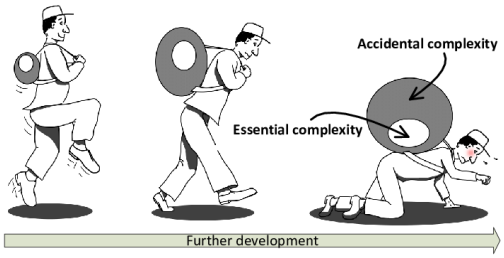
Fred Brooks, author of "The Mythical Man-Month" also observed that software costs were overwhelming hardware costs.
Software development is hard, because of its complexity. But there are two kinds of complexity: Essential an Accidental.

Reducing complexity

- Essential complexity cannot be reduced
- Reduce accidental complexity, which tends to grow over time

- Fools ignore complexity.
  Pragmatists suffer it.
  Some can avoid it.
  Geniuses remove it.
  - *Epigrams on Programming*, Alan Perlis, first recipient of the Turing Award

- Declarative techniques reduce accidental complexity by specifying *what*, not *how*



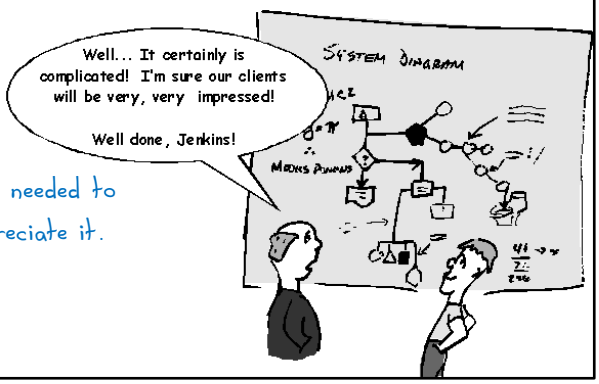
Grewe, Axel et. al. (2017). *Automotive Software Product Line Architecture Evolution: Extracting, Designing and Managing Architectural Concepts*.

Why is it so difficult to reduce complexity? Why are declarative techniques not used everywhere?

Engineers like complexity

- It gives a sense of *doing* something
- Everyone thinks that they are very clever
- They see themselves as software engineers, not domain experts
- Why has elegance found so little following? That is the reality of it. Elegance has the disadvantage, if that's what it is, that hard work is needed to achieve it and a good education to appreciate it.

  - Edsger W. Dijkstra



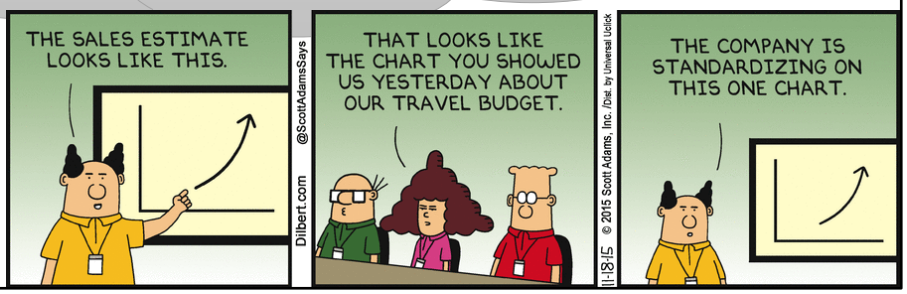
The most difficult is to understand the essential complexity thoroughly. Understanding and creating accidental complexity is easy.

One source of accidental complexity is lack of using the domain to frame the form of the solution.

Picture source: http://antipatterns.com/briefing/sld015.htm

The manager's way to reduce effort

- Making software has been, and will always be difficult
- I have no way to do anything about that
- Better make the software *generic*, so we have to make it only once
  (and the rest is just configuration)



Managers ask more often to make everything generic. The idea is that something generic "only needs to be configured".
Sometimes I suggest that the most generic software is a C compiler, and that the C stands for 'configuration'.

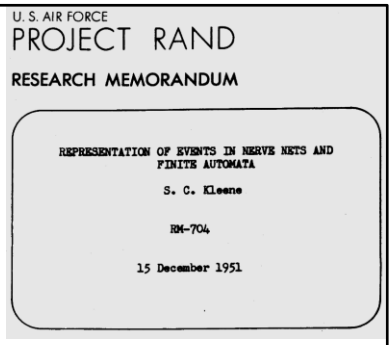
Success factors of declarative techniques

- Reducing accidental complexity

- More Edsger Dijkstra quotes:
  - Simplicity is prerequisite for reliability.
  - About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.
  - The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.

Regular Expressions

- Invented by Stephen Cole Kleene in 1951
- Describe *what* a neural net (in 1951) or a state machine does, not *how*
  - Omit start, end, and other state labels
- https://youtu.be/528Jc3q86F8

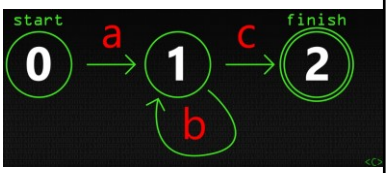

U. S. AIR FORCE
PROJECT RAND
RESEARCH MEMORANDUM

REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA

S. C. Kleene

RM-704

15 December 1951



ab*c

This Computerphile video about the history of regular expressions: https://youtu.be/528Jc3q86F8

Domain Specific Languages

- SQL
  - relational algebra
  - operations on the underlying data model (what, not how)
- XSLT
  - pattern matching on the input to drive computation
  - operations on the underlying data model (what, not how)
- Xforms
  - interactive view on data model
  - operations on the underlying data model (what, not how)
- LINQ, GraphViz, Vega

- DSLs do not by definition reduce complexity, effort, cost

Declarative languages are often domain specific – operate on a specific data model.
Not all DSLs are declarative!
This slide lists some DSLs that I think are successful declarative programming languages.

https://vega.github.io/vega/ - Vega – A Visualization Grammar

Making a DSL is easy, making a useful one is not.

Pitfalls of declarative techniques

- Genericity
  - Not being specific, widely applicable (configurable)
  - Complex technical meta-model
  - When configuration becomes Turing complete…
- Reification
  - Making an abstract concept into an explicit data object
  - Objects that only represent something else
  - Data Transfer Object (DTO); represents data storage, not a domain entity (but DTOs were better than entity beans in EJB < 3.0)

Both genericity and reification can be useful. They can also be dangerous.

EJB is a prime example of accidental complexity. Many ‘frameworks’ start out useful and grow into bloated swiss army knives.
https://playingintheworldgame.com/2013/10/28/the-ultimate-swiss-army-knife-1880-version/

Aside: Enterprise software

- Conway's Law: "Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."

enterprisey

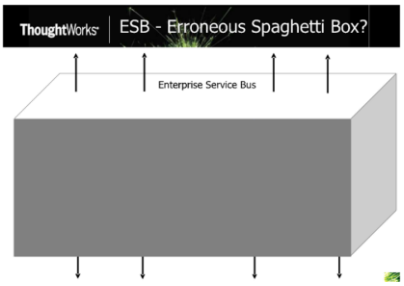
Enterprisey is anything that is clearly part of the enterprise culture.

A culture where buzzy words are often used to talk about simple concepts but they do it that way so they can look smart inside that culture and impress outsiders.

The term enterprisey is used to be explicit on the criticism of such behavior.

PC guy: this is so cool, how did you improved this software?
mac guy: easy, I've added a delay of 3 seconds to every action and now users are feeling it's enterprisey

small business owner: I'm tired of presenting good projects and getting rejected, what's wrong with these people?
colleague: they only want their bloated, hyperexpensive projects approved because they're trapped in their enterprisey ness. The're surely saying one to each other: "hey, nobody got fired for hiring IBM"



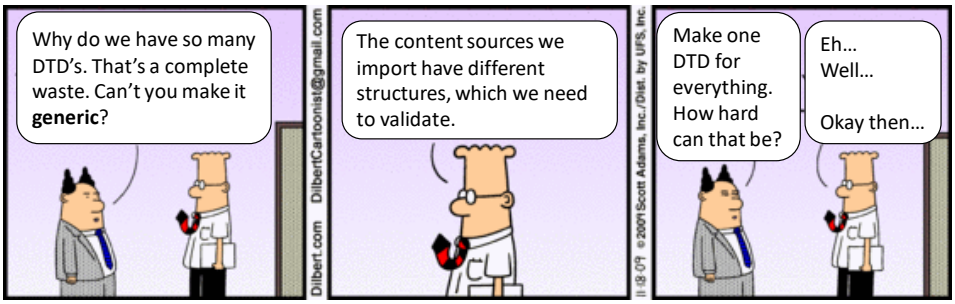
Does My Bus Look Big in This? Jim Webber & Martin Fowler

Enterprise software is difficult. Consultants make it seem easy by using lots of buzzwords that would solve the difficult parts.
Does your software have exponential blockchain user experience ; IoT SAAS SPA API ; agile cloud computing algorithm mashup ; full-stack mobile-first responsive design ; disruptive serverless microservice toolchain ; crypto ecosystem vapourware ; actionable holistic business-IT alignment convergence ; multichannel devops microblogging web 2.0 engine framework.

Instead of focusing on understanding essential complexity, the essential complexity is hidden under a layer of accidental complexity.
An ESB looks on the outside like "everything is easy".

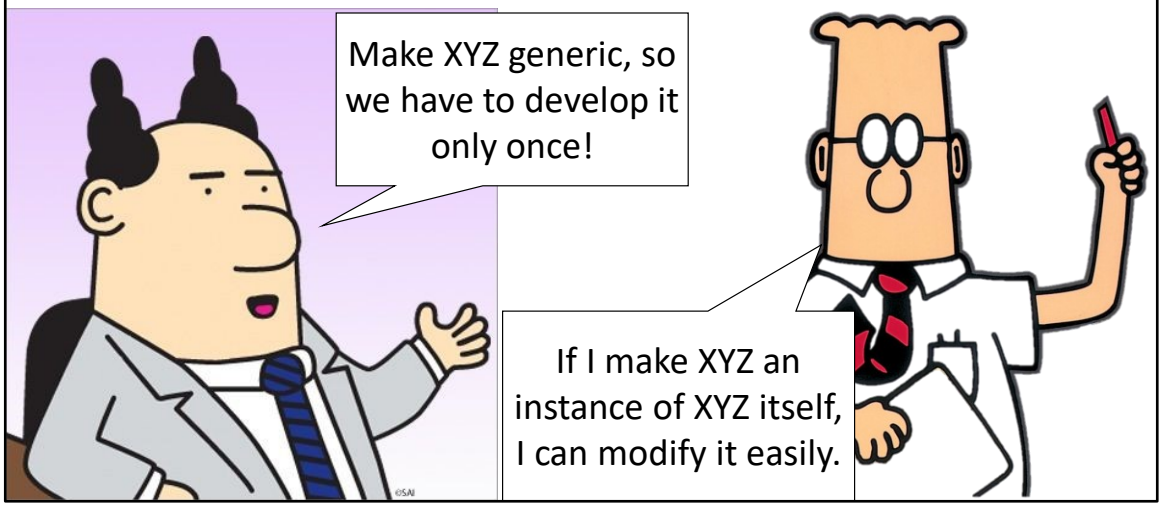
Publishing from multiple content sources

- Every content source has its own DTD (~ 2005).



Around 2005, I worked for a company that had many content sources.
At the time, there was this hype about integrating many content sources in a flexible (generic) way, and publishing to many channels (paper, CD-ROM, Web, …)

Making a generic <XYZ>



Note on the following slides:

Code samples, identifiers, and programmer or company names are either the products of the author's imagination or used in a fictitious manner. Any resemblance to actual software developers, living or dead, or actual program code is purely coincidental.

Reification: the “Thing DTD”

- A “thing” is something with “things”.

```
<!DOCTYPE element [
   <!ELEMENT element ( attribute* , content? )>
   <!ATTLIST element name CDATA #REQUIRED>
   <!ELEMENT attribute EMPTY>
   <!ATTLIST attribute name CDATA #REQUIRED>
   <!ATTLIST attribute value CDATA #REQUIRED>
   <!ELEMENT content (#PCDATA | element)* >
]>
```

```
<element name=“page">
  <attribute name="type" value=“slide"/>
  <content>
    <element name="title"><content>The “thing” DTD</content></element>
    The most generic DTD is also the most useless.
  </content>
</element>
```

One size fits all → most generic!

Easy to translate from and into any original format!

No more validation errors!

Any XML document, regardless of the DTD or schema, can be converted to an XML document using the “Thing DTD”, and back.
What’s not to like?

This is Genericity by Reification: Describe X in X. Call it a meta-model.

Generic databases

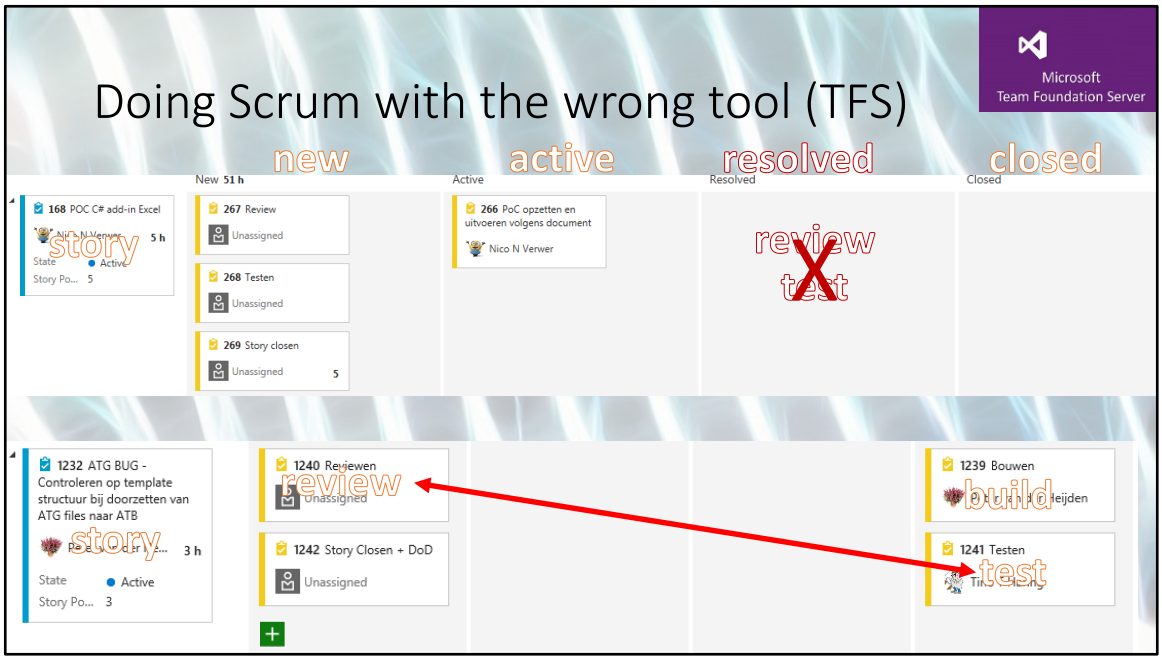
- Database schemes are hard to change because only the DBA is authorized to do that
- Make the database within the database (in 2020 speak: Software Defined Database)

Only one database schema for all storage!

- Bonus: Also store code in the DB and bypass the cumbersome deployment process

I have seen databases used in this way more than once.

Reification is often used in linked data, usually because of perceived limitations of RDF and other representations.



The tools that software developers use also provide opportunities to make them more generic using reification.

TFS does not support the usual scrum board columns “review” and “test”, and “resolved” can only be used for bugs, not stories.
To be fair, TFS allows for some customizations, but I never met a TFS admin who knew (how to do) that.

So we split up the phases of a story, and have *tasks* for ‘active’, ‘review’, ‘test’, ‘close’. These become activities, not states.

This can create inconsistencies, as in the second example. It also splits the history of a user story across tasks.

Reification with XSLT

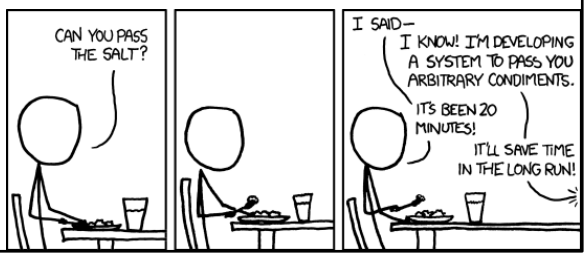
```
<xsl:template match="dcterms.creator">
   <creator-id function="lookup-organisation">
     <param name="organisation">
       <xsl:value-of select="." />
     </param>
   </creator-id>
 </xsl:template>
```

```
<xsl:template match="/*[@function]">
<xsl:choose>
    <xsl:when test="@function = 'lookup-organisation'">
      <xsl:variable name="org"
        select="param[@name = 'organisatie']"
as="xs:string"/>
      <xsl:variable name="org-id" select="… $org …"
as="xs:string"/>
    </xsl:when>
…
```

This makes a function call into an XML element in the 'analysis phase', in order to execute the function in a later 'enhancement phase'.
This was supposed to lead to "separation of concerns".
Just calling a function in XSLT is not very interesting for software developers. Anyone can do *that*.

Conclusion

- Genericity may seem to reduce effort
- Reification may seem to provide flexibility
- Often they lead to more accidental complexity
- Declarative techniques *can* provide reduced effort and flexibility by removing accidental complexity
- Specify *what*, not *how*, but first ask "*why?*"



Picture source: https://xkcd.com/974/