Literate Programming Revisited

Robert Bentall, 2019-09-24

Outline

- Motivation and context
- Knuth’s approach
- What I needed to produce
- Approach
- Tooling
- Worked examples…
- How well does it work?
- What’s next?

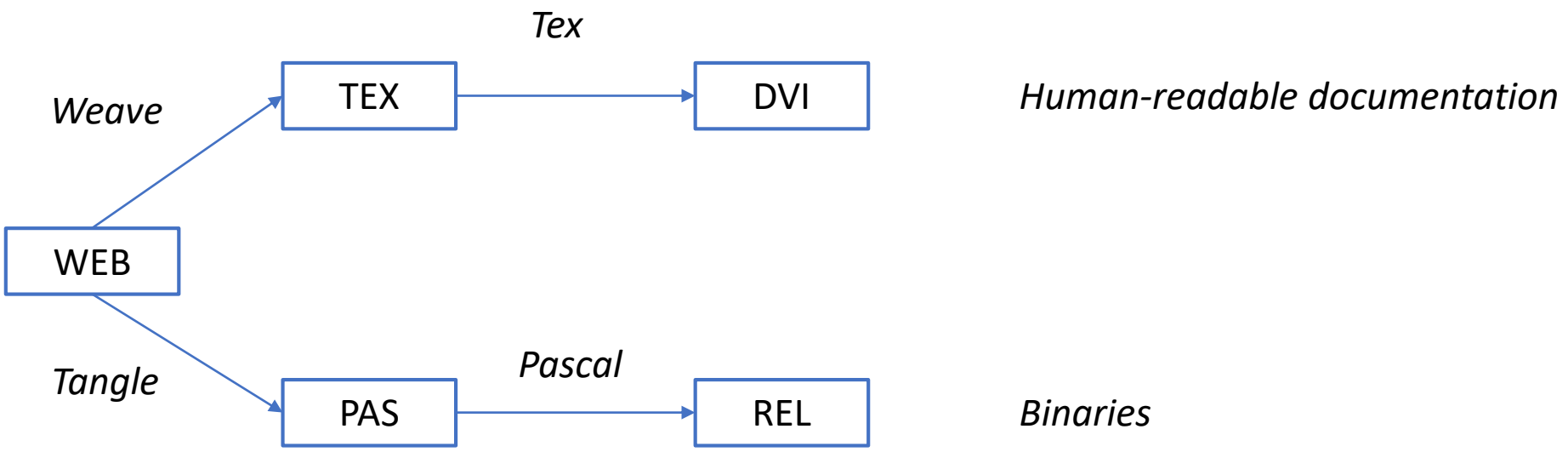
A practical problem – and some questions

- In my day job
  - We needed to produce MIL-498 std SDDs for an existing mature codebase
  - Our existing approach required a lot of work using a UML tool, and required maintaining two artefacts (the software and the model)
  - We were exploring the use of doxygen-based comments to provide content for an SDD on new developments
- For a long time I've been interested in
  - Which software artefacts to represent during construction (and why)
  - How to do this (prose, bubbles and lines, UML, formal methods, design templates…)

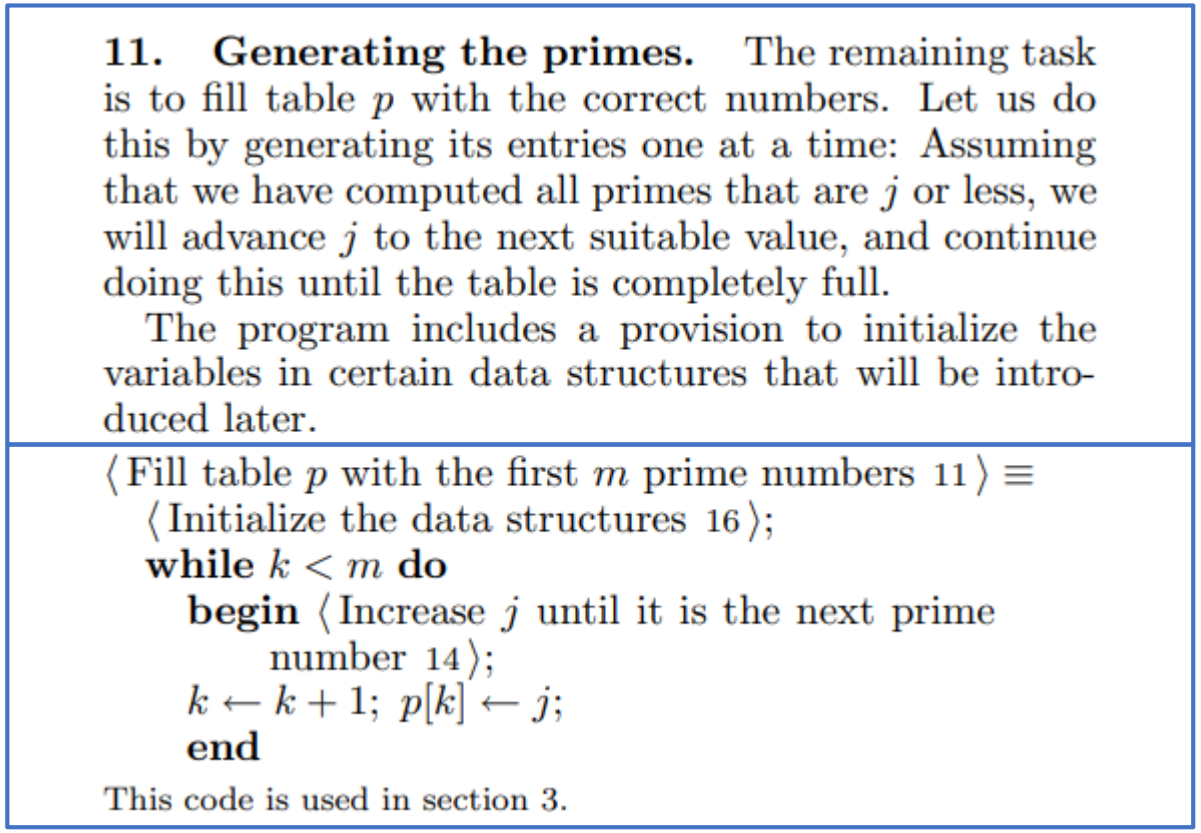
Some history

- Literate Programming (Knuth, 1984) published after c. 10 years of experimenting with programming styles
- An attempt to improve on Structured Programming
- He wanted to improve documentation of programs, to make them easier to read and understand (by a human).
- He saw the program as a work of literature to be read and understood by humans.

Knuth's tooling: WEB = Tangle + Weave



An example of rendered documentation



11. Generating the primes. The remaining task is to fill table p with the correct numbers. Let us do this by generating its entries one at a time: Assuming that we have computed all primes that are j or less, we will advance j to the next suitable value, and continue doing this until the table is completely full.

The program includes a provision to initialize the variables in certain data structures that will be introduced later.

⟨ Fill table p with the first m prime numbers 11 ⟩ ≡
  ⟨ Initialize the data structures 16 ⟩;
  while $k < m$ **do**
    begin ⟨ Increase j until it is the next prime number 14 ⟩;
    $k \leftarrow k + 1$; $p[k] \leftarrow j$;
    end

This code is used in section 3.

Human-readable documentation

⟨ThisFragmentName⟩ ≡
  ⟨OtherFragmentName⟩ ;
  Other Pascal source code…

From http://www.literateprogramming.com/knuthweb.pdf

Observations

- His style evolved significantly during the first year of use
- He loved using it
- It’s not really taken off as an approach

What did I need to deliver?

- Render a MIL-498 std SDD based on source code comments
  - Used on C, C#
  - Retrofitted onto an existing code base
  - Applied on new developments
- Have the approach accepted by the software team
- Have the output accepted by
  - other stakeholders in the business
  - the customer

Documentation use cases

- Small scale
  - Produce a paper for others/you to read
  - Toy programs
- Large scale
  - Manage complexity over longer construction cycles
  - Mixture of up-front design, prototyping, maturing components at different times and across multiple teams
  - Help reviewers focus on things that matter
  - Make it accessible to reviewers with varying levels of expertise
  - Provide data required by the contract (CDRL)

Developing the capability

- Proof-of-concept on a new development (a Java hobby project)
  - Identify candidate tools
  - Convince myself it was going to work
  - Produce a demonstrator
- Show and tell in the office
  - Explain the approach
  - Seek out feedback – identify possible problems
  - Get buy-in
- Start with something small and simple
- Keep discussing as a team

MIL-498 Software Design Description

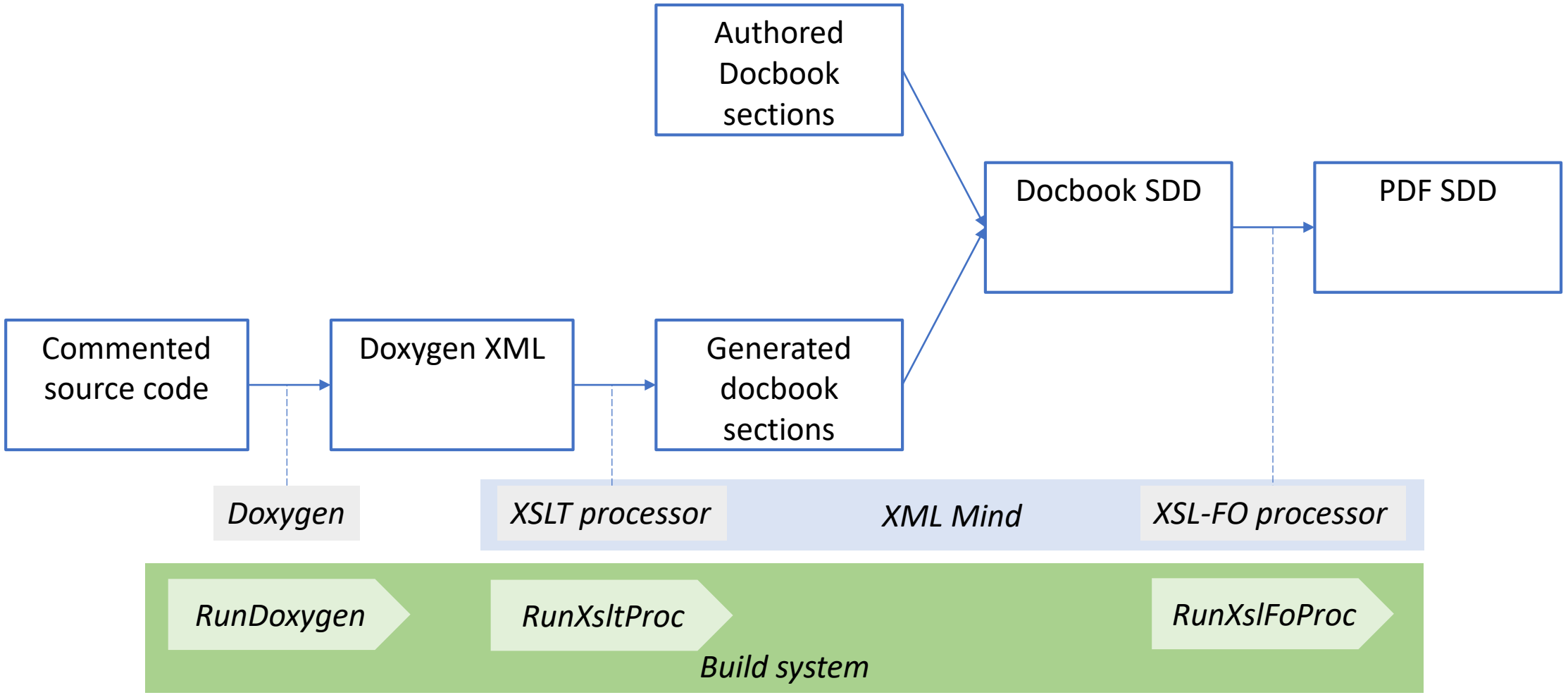
1. Scope
2. Referenced Documents
3. CSCI-wide design decisions
4. CSCI architectural design
    1. CSCI components
    2. Concept of execution
    3. Interface design
5. CSCI detailed design
6. Requirements traceability
7. Notes

See https://kkovacs.eu/free-project-management-template-mil-std-498

Tooling

- Doxygen - for turning comments into documentation
  - http://www.doxygen.nl/
- Docbook schema – to produce high-quality published documents
  - http://www.sagehill.net/docbookxsl/
- XmlMind – xml WYSIWYG editor
  - https://www.xmlmind.com/xmleditor/
- PlantUML – for rendering UML diagrams
  - http://plantuml.com/
- XSLT processor – for transforming XML
  - Contained in XmlMind
- Print formatter – for rendering XML to PDF
  - Contained in XmlMind
- Build system
  - Pick the one you like…. Ant/MSBuild/Maven/Jenkins

Workflow


Authored Docbook sections
Docbook SDD
PDF SDD
Commented source code
Doxygen XML
Generated docbook sections
Doxygen
XSLT processor
XML Mind
XSL-FO processor
RunDoxygen
RunXsltProc
RunXslFoProc
Build system

Mapping the toolchain onto the SDD

1. Scope
2. Referenced Documents
3. CSCI-wide design decisions

Manually written Docbook sections

4. CSCI architectural design
    1. CSCI components
    2. Concept of execution
    3. Interface design
5. CSCI detailed design
6. Requirements traceability

Generated Docbook sections

7. Notes

Manually written Docbook section

Show & tell

CSCI architectural design

Section	SDD Interpretation	Doxygen elements	Application
CSCI components	Structure of “software units”	Package/namespace structure Classes/modules Functions/methods Includes/using relationships	Nested docbook sections to represent structure of the application Comments as required
Concept of execution	Behaviour of “software units”	Call relationships Called-by relationships Pre/Post conditions Invariants In-body documentation	Tabular representation of call graphs/called by graphs Tabular representations of contracts Occasional UML diagrams (where useful)
Interfaces	Interface characteristics of “software units”	API definition Method signatures	Sections only where needed. Call out to IDD

CSCI detailed design

- Use as a reference manual
- Exhaustive list of all software elements in the CSCI
- Generate completely from Doxygen \brief comments and source code structure

Traceability

- Demonstrate that your design implements the requirements
- Make sure no requirements are missed
- Make sure you have no redundant code
- All code should trace back (ultimately) to a software / interface requirement or design decision
- Can include all of this in section (6), or alternatively include in section 4.1

Maturity

Maturity scale	Class 1	Class 2	Class 3	Class 4
Rationale summarised	✓	✓	✓	✓
Data members defined	✓	✓	✓	✓
Functions defined	✓	✓	✓	✗
Contracts defined	✓	✓	✗	✗
Requirements traced	✗	✓	✗	✗
Implementation designed	✗	✓	✗	✗
Implementation completed	✗	✗	✗	✗

Commenting example

```
/**
  \brief This class provides all information about an annotation needed by the merge utility.

  \par Concept of Execution
  \parblock
  Pellentesque sapien ex, malesuada quis sapien vel, fringilla auctor purus. Pellentesque et sodales leo. Orci varius natoque
  penatibus et magnis dis parturient montes, nascetur ridiculus mus.

  IMAGE_REF:My image:./AnnotationsFilterModuleStructure.png
  \endparblock

  \invariant Some stuff about the class that must always be true

  \req RAB_REQ_001

  \req RAB_REQ_002

  \rationalesummarised     Yes
  \datamembersdefined      Yes
  \functionsdefined        Yes
  \contractsdefined        No
  \requirementstraced      No
  \implementationdesigned  No
  \implementationcompleted No
*/
```

Doxygen features

- \parblock – define a named block
- \xref aliases (\req, \maturity)
- \pre, \post, \invariant
- \image doesn’t work with xml output

Pros and cons…

- Rendered documentation looks good, if you are prepared to put in the effort
  - Stylesheets
  - Commenting styles
  - Rendering customisation
  - Coaching and supporting your colleagues
- Once you've got an approach working, it's easy to keep to the commenting style within your code
- Creating the stylesheets to go from doxygen xml to docbook is time consuming – doxygen has some odd quirks, the xml schema is non-trivial
- It can be hard to get over the preconception that you are "just documenting what's there", rather than building something in a robust way

What's next?

- There is a need for tooling to document aspects of software *not* captured directly by source code
- It needs to integrate seamlessly with both source code and with requirements management tools
- It needs to play well with a modern software engineering tool chain (command-line driven, lightweight, well-defined interfaces for getting data in and out)
- We want to eliminate redundant effort (don't waste time duplicating something in UML that can be rendered from the source code directly)
- Allow integration of different approaches (formal methods, prose, bubbles and lines, UML, data flow…)
- Support automated analysis – where appropriate
- Not tied to a particular language or vendor
- Easy to use for the engineer

Thank you!

- Any questions?