Everything you ever wanted to know about *Formal Specification*

(but were afraid to ask)

Well… perhaps not *everything* :-)

This is an extract of a presentation made at Fujitsu UK, in 2014.

The presentation is available on Google Slides:
https://docs.google.com/presentation/d/1MuSfkShxcSz6BN7Aym-Gt_XeLYWHKD-ds32PZcgQTfE

Background

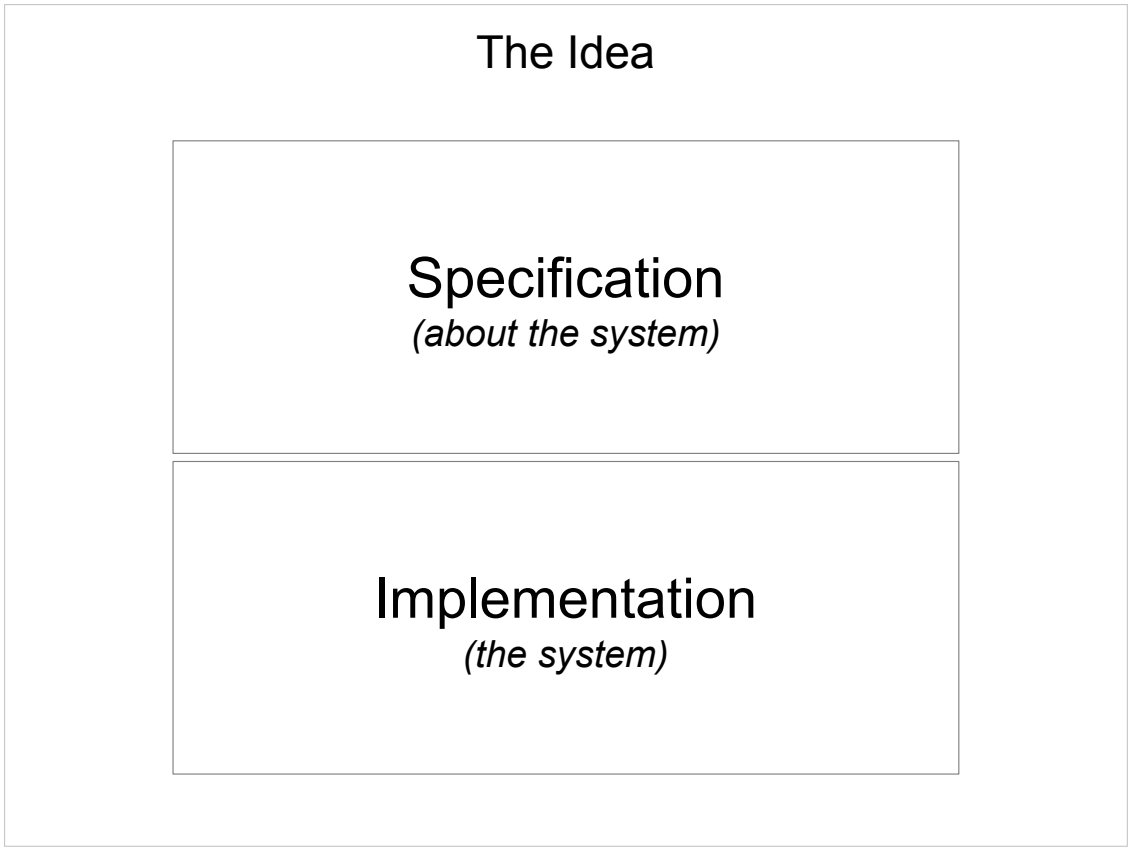
- *Vienna Development Method* (VDM)
- About design and construction of working systems
- Not about documentation or processes
- *VDMJ* open source tool
- *Overture* adds an Eclipse interface to *VDMJ*
- *VDM VSCode* extension available for VSCode
- Maintained by a group of industry/academics
- But there is a problem...

- VDM = *Vienna Development Method*, developed several decades ago by IBM Labs in Vienna.
- VDM is a method for capturing requirements, specifying a design that meets the requirements, and then producing code that conforms to the design.
  - *VDMJ* is an open source command line tool for working with VDM specifications.
  - *VDMTools* is another open source tool for working with VDM.
  - *Overture* is an Eclipse based tool that extends VDMJ, adding a GUI and other usability features.
  - *VDM VSCode* is a Visual Studio Code extension that uses VDMJ.
- The Overture group maintains the tools, mostly from academic institutions (from UK, NL, DK, PT, JP)
- So VDM sounds good, but there is a problem...

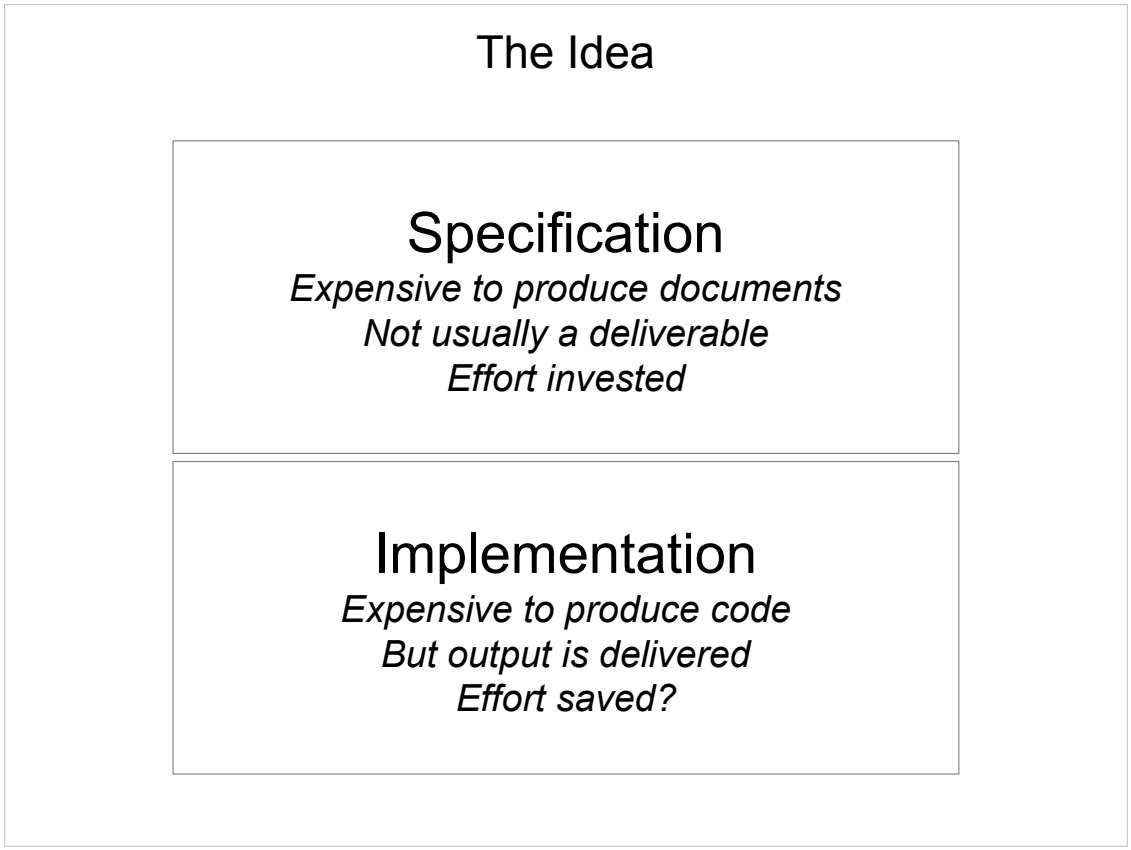
The Idea

The only thing harder to sell than formal methods.

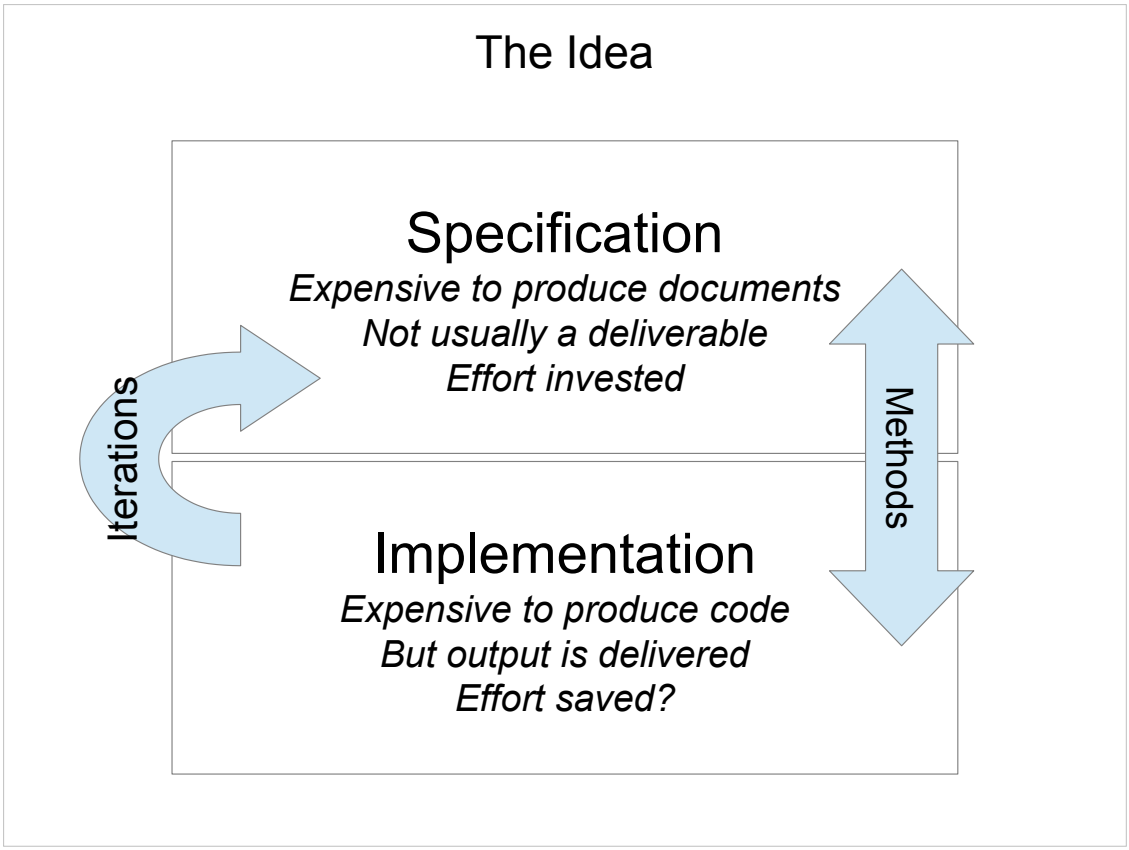
- VDM is a formal method.
- Formal methods are development methods with a sound mathematical basis (not just pretty diagrams and good intentions).
- Formal methods are usually thought of as:
  - *Very hard work (mathematical analysis, needing experts)*
  - *Very old fashioned (ie. irrelevant in today's agile, lean, emergent design world)*
  - *Not very practical (they don't scale to industrial sized systems)*
- The truth is more complicated. Formal methods are used routinely (and effectively) in some domains. Their use may be a regulatory requirement.
- We need to get over the idea that “formal” necessarily means difficult, irrelevant or time consuming.
- But received wisdom says that you can't easily persuade sceptics (still less, cynics) about the value of formal methods.
- Besides, this might be “lipstick on a pig” (industry is just too chaotic to use this effectively).
- But by rejecting this outright, we miss an opportunity...



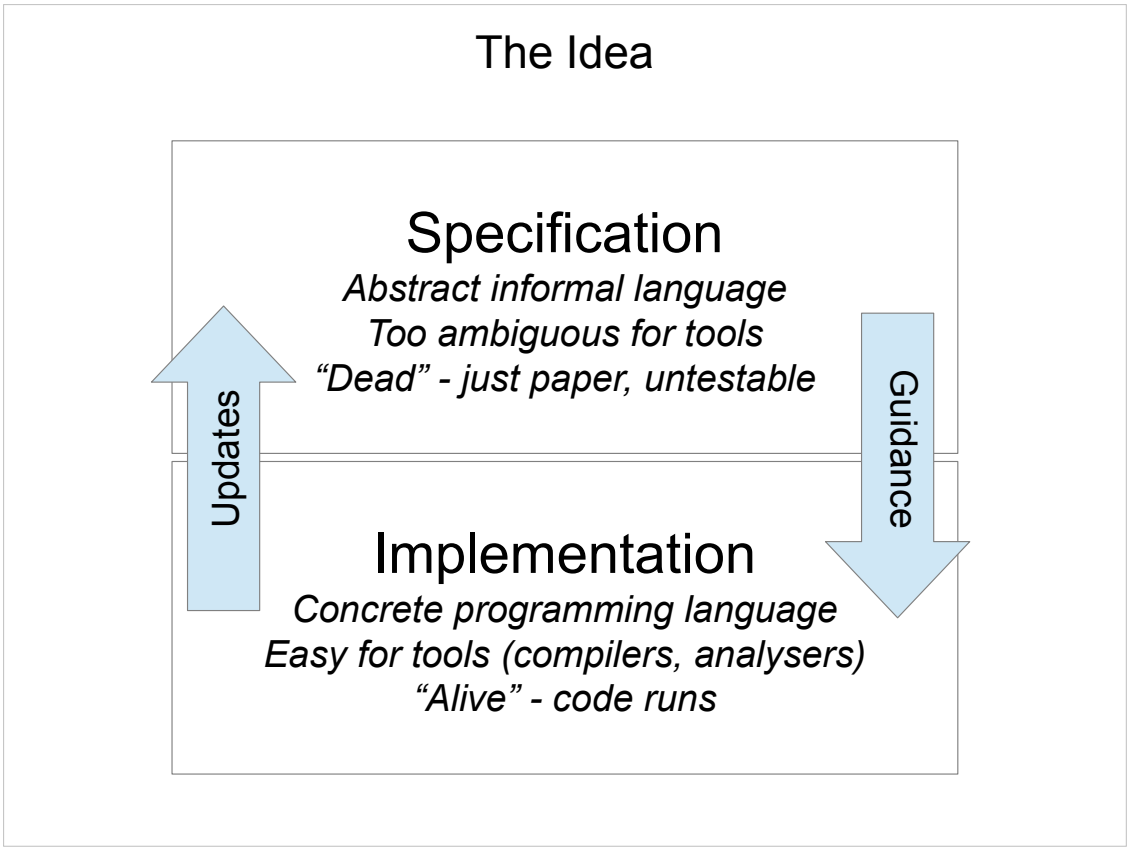
- Specifications are documents that describe the system to be developed – from its requirements, through architectural descriptions of the solution and its high and low level designs, to how the system will be tested.
- The implementation is composed of all the artifacts that have to be written in order to deliver a working system – essentially code and configuration data, running on properly configured machines.



- Specifications are not usually delivered to the end customer. Some, like requirements and interface specifications have to be, since they document an agreement between parties. But specifications that relate to the internal design of a system are usually not.
- Yet specifications are expensive documents to produce.
- The reason they are produced is that they should reduce the overall cost of the project, by allowing developers and stakeholders to see what is proposed before it is created, to guide the developers in creating what has been agreed, and in subsequent maintenance, to allow people to see the big picture when modifying the product or correcting bugs.

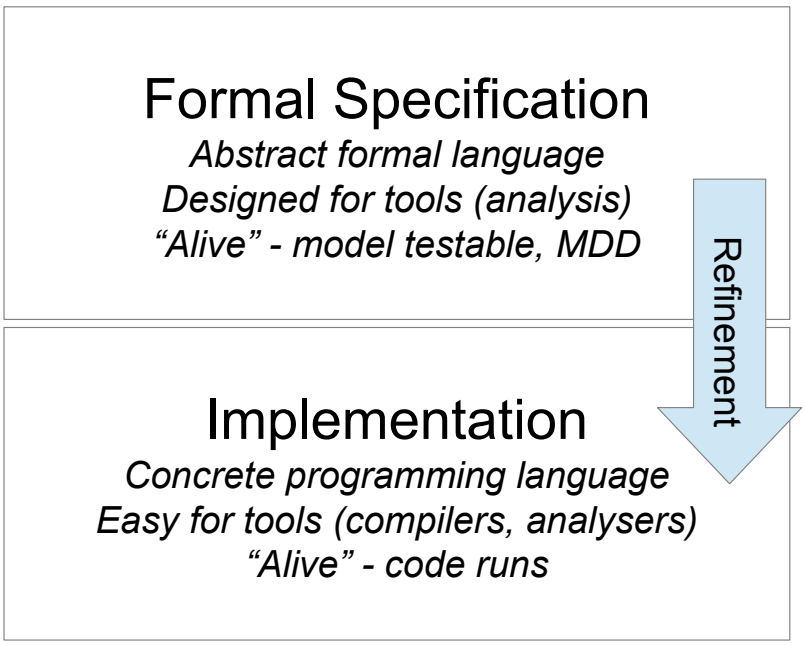


- Development methods – waterfall, spiral, iterative, agile etc. - modify the relationship between specifications and implementation, effectively moving effort between the two.
- Waterfall or “V” style methods depend heavily on up-front specification production before implementation.
- Agile and iterative methods try to reduce the initial cost of specification by involving developers directly with stakeholders to discuss requirements, leading to rapid development cycles that are repeated. Written specifications can be produced, but to document what was done rather than to describe what will be done up front in detail.
- There are many methods that take a middle path.

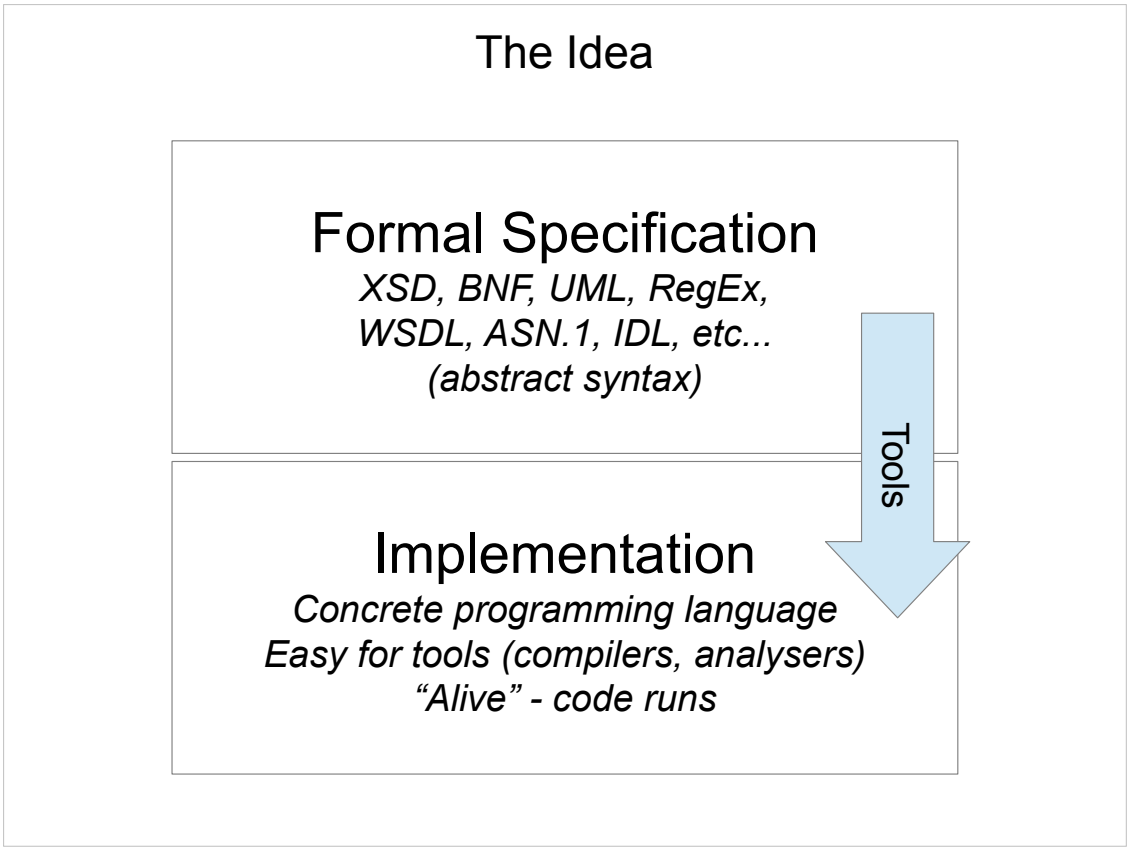


- Specifications (requirements, interfaces, designs) are usually written in informal language – English, with diagrams and tables.
- The informality of the language used (ambiguities) means that tools cannot process the specification to produce anything useful, nor can we perform any analysis of the specification (for example, to see whether a design meets a requirement for certain).
- Such specifications are “dead”. You can only check them by hand.
- By contrast, languages used for the implementation are formal. They are designed to be understood by tools like compilers and interpreters, and because of their lack of ambiguity, other tools can analyse them – static analysis tools: *FindBugs*, *CheckStyle* etc.
- The implementation is “alive” because it can be executed and tested to see whether it meets its requirements.
- Therefore problems tend to be discovered late, when the system becomes alive towards the end of implementation rather than during specification. This can be expensive.
- Since the specification is dead, it's hard to tell whether it has been updated to match the implementation. Over time, informal specs always become inaccurate; eventually, they become too expensive to correct.

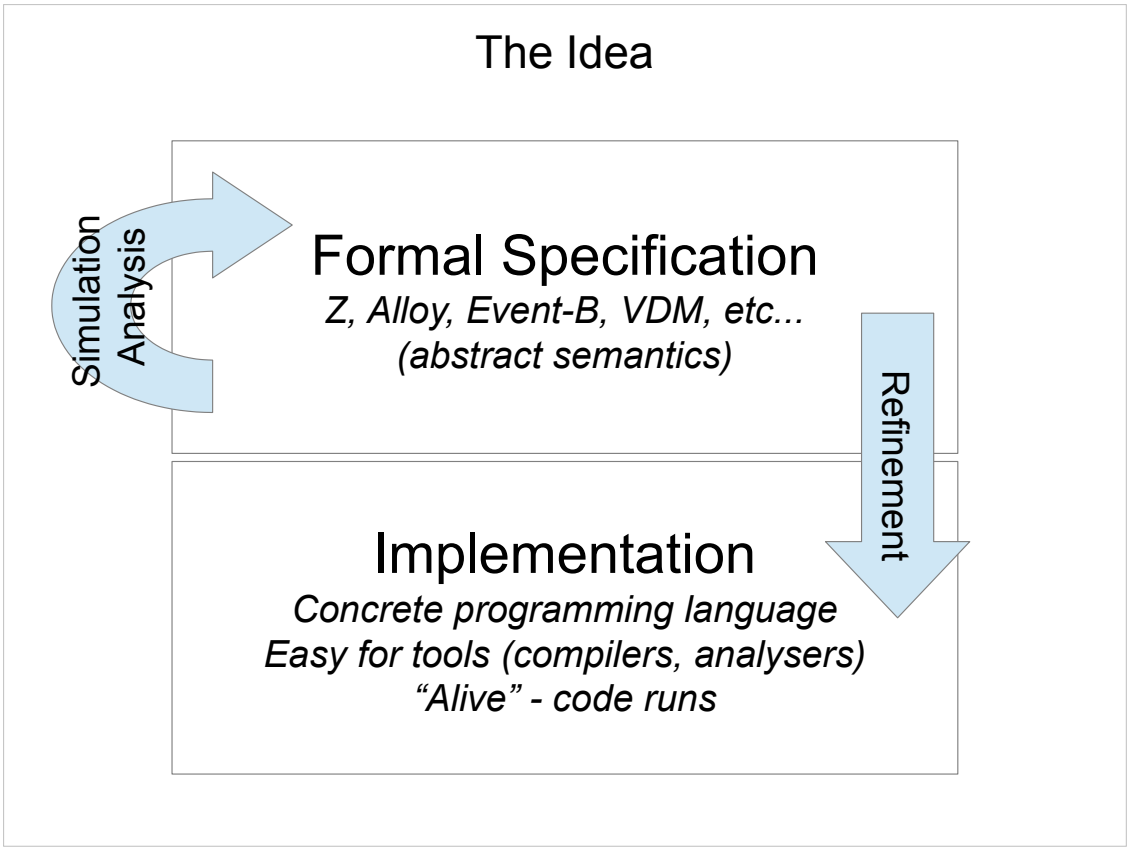
The Idea



- The idea of *Formal Specification* is to use formal languages during specification as well as implementation.
- This complements the informal description rather than replacing it.
- The formality in the specification means that it becomes "alive", like the implementation. A formal specification can be analysed by tools, and simulations of the system's functionality can be made to check it meets requirements.
- This is a type of Model Driven Development (MDD).
- Knowing that a design is sound before the implementation gets too far considerably reduces the risk in the project.
- A formal specification is typically very abstract, focussing on the important functional aspects of the system, so it is very small compared to the implementation. Producing it need not be onerous.
- The implementation is guided by the specification as before (it is a refinement of it), but we know that the design works before the implementation starts, so the specification is less likely to need to be updated afterwards.



- This isn't such a new idea. We already use several formal languages routinely when writing specifications.
- These languages are precise (unambiguous), concise, and typically tools allow us to gain benefit directly from the resulting specification – like generating code, verifying XML documents, generating parsers or AST classes, etc.
- So we use these languages because of the *business benefit*.
- But these languages are usually syntactic rather than semantic. For example, WSDL and XSD may describe all the structures needed to make a SOAP request, but it does not define what that request really does (for example, how "add item to basket" affects the state).
- UML is an exception: it attempts to define semantics via activity diagrams, state charts and sequence diagrams. But the notation itself is only semi-formal, typically wrapping informal English in formally defined diagrams. So tools cannot generally do much with UML.



- But there are many formal languages that do define semantics.
- Typically, languages describe a system in terms of a collection of data types with constraints (invariants), state data of those types with constraints, and functions or operations which manipulate the state with pre- and postcondition constraints.

The Idea

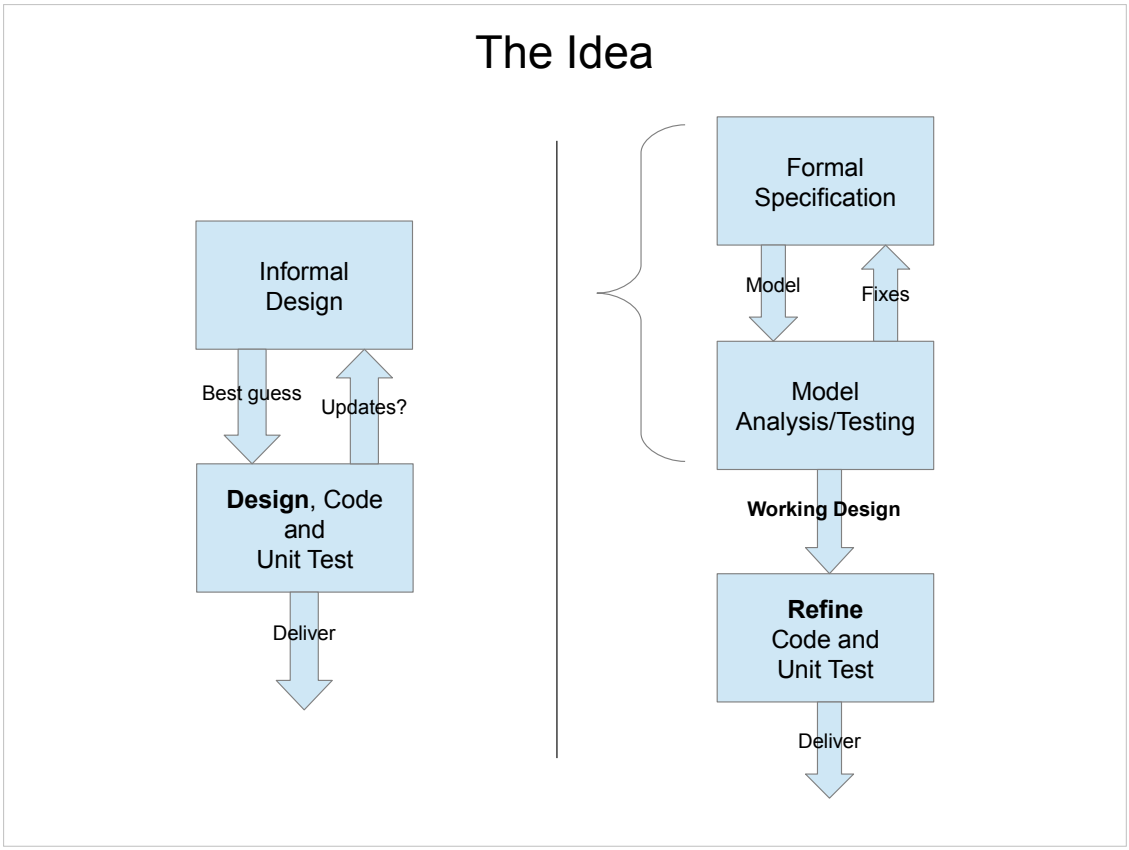
So why is this useful?

- As before, language is precise and concise
- Universal model, allows analysis and simulations
- Tools can test the specification exhaustively
- Can gain very high confidence in design

But isn't this writing the system twice?

- A formal model is very abstract
- A fraction of the size of the implementation
- No class or code constructs, just abstract operations

- Like XSD, WSDL etc, formal languages are precise and concise. Tools allow a variety of analyses, from full formal consistency (ie. proved never to violate any constraints) to simple use case simulations.
- Heavyweight analysis does involve mathematics. But this is rarely required outside critical industries (it is laborious rather than difficult).
- Lightweight use benefits from the precision of the notation and tool simulations to build high confidence in the design, without the need for heavyweight analysis.
- Combinatorial tests can be used to automatically check vast numbers of scenarios (in between ad-hoc tests and formal proof). These are very good for picking up “corner cases”.
- Tools measure model test coverage, so you know that the test suite covers all of the functionality of the model.
- Formal models are typically very abstract. For example, they may not attempt to define the class hierarchy of the code, instead describing abstract operations like, “add item to basket”.



- So formal specification gives us a different style of working.
- Informal specifications are “best guesses”, because the designer has no certain way to know whether what they specify is complete, consistent or meets the requirement.
- Those informal design specifications are further “designed” during the coding phase to find out whether it is possible to realise them, and if so, whether they work and are complete.
- Formal specifications are tested before coding starts, but at an abstract level. Tests or analysis of the model is then performed to show that the model is complete, consistent and meets the requirement.
- The final coding stage proceeds on the basis of a working design. The coding job is to refine the abstract design to add code level details and (say) object oriented structure.
- You would expect comparatively little update to the model after coding as design errors are unlikely and the model has no coding details.

The Idea

But how can you refine the model accurately?

- The full VDM method would use mathematics:
  - Laborious and time consuming
  - So this is rarely done!
- But a lot can be done by comparing:
  - Diagnostic trace from model and code
  - State data and output from model and code
- Pre- and postconditions, invariants => assertions
- Model simulations become integration tests

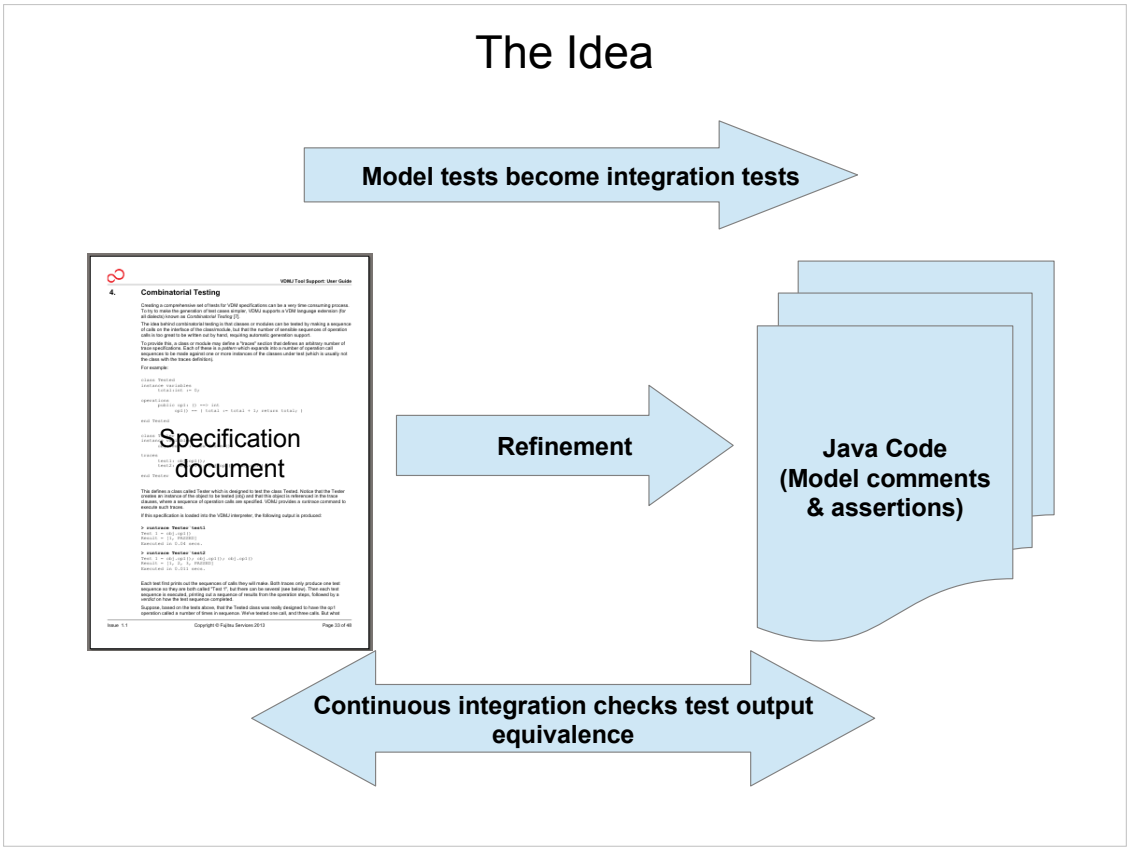
- It is important that the refinement of the formal model into code does not change the design!
- In the full VDM method, this is done mathematically (called data reification and operation decomposition). This is extremely laborious without tool support and is rarely used. But strong confidence can still be gained by informal use of model simulations.
- A model simulation and the code should produce “the same” diagnostic trace when performing the same operations. For example, the code could have a special *log4j* category for such diagnostics.
- A model simulation and code should produce “the same” output data or changes in state data when performing the same operations.
- Assertions can be used to represent (some) preconditions, postconditions and state/type invariants in the code, so violations are more obvious.
- Model tests of the design become integration tests of the implementation. It may be possible to automate this.
- This is a lot more likely to produce working designs and matching code than the informal process!

The Idea

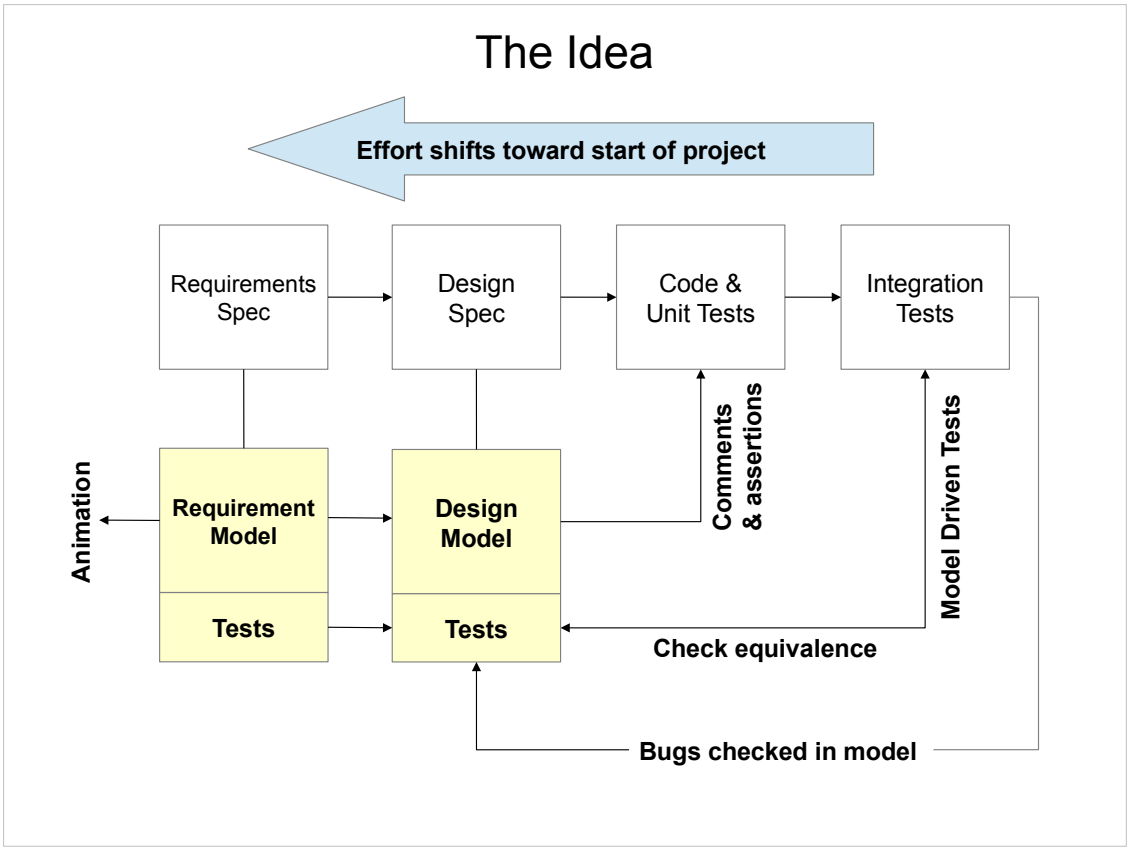
But when is this really effective?

- Not everywhere!
- Often only applied to subsystems, not the whole
- Typically areas with “a lot of design”
- Or areas where bugs would be serious
- Or semantic interfaces with other systems
- Often areas that (otherwise) become “dark code”
- But reverse engineering models is difficult.

- It is important to remember that formal specification will not be cost effective in all circumstances, and it should only be applied where appropriate.
- It is typically effective in subsystems rather than being universal.
- It is very good at keeping control of complexity, and so suited to subsystems with complex designs that are difficult to describe (or for which description would otherwise be missing).
- It is also good for reducing the risk of errors, so benefits areas that are sensitive (eg. financial, safety critical or regulatory areas).
- And it helps to communicate interface semantics with external parties. They may initially object to the unfamiliar language, but experience is that ultimately the clarity is appreciated.
- Areas that would have benefited often become the “dark code” in the system, where few (if any) people understand the code, and no one dares to change it, even though the function of the code is essential.
- Reverse engineering models from code is notoriously difficult. So this is not a way to rescue a “ball of mud”. It is a way to prevent balls of mud from forming.



- The formal specification can be embedded in a normal design document with (informal) supporting descriptive text.
- Tools can read the specification directly from the original document (eg. from an design, written in doc, docx or ODF format)
- Model tests can either be embedded in the document (perhaps an appendix) or held as separate files.
- The design is therefore a *testable artifact*, just like the code, and tests of the design would be executed in continuous integration.
- Model tests are roughly equivalent to integration tests of the real system. Model tests can therefore be translated into integration tests (semi-automatically).
- Output from integration tests (eg. via diagnostic logs) can be compared for equivalence with the output from the same tests run against the specification model.
- Therefore the design and code are *locked together*. Divergence is identified by test comparison failures.



- Typically, two models are important:
  - A requirement model: a very abstract description of the external behaviour of the system (typically just if-then-else logic)
  - A design model: an abstract description of the internal behaviour of the system (some implementation details, but not code)
- The Design model is a refinement of the Requirement model – the tests of the one, when translated by the abstraction, should produce the same behaviour in the other.
- Requirement models can be used to show animations to customers very early in the project. The same animations can be used by integration test, acting as test oracles, defining the system's functionality.
- Design model specification details should be copied into code (as comments and assertions) to aid traceability. The code is a refinement of the design.
- In maintenance, bugs are fixed/tested in the model first, then moved to the code. So the design model is a dynamic part of the system and keeps track with code over time, even during maintenance.
- Coding is less “creative” and faster. Design is slower. Overall about the same effort, but quality is consistently much higher (see industry studies) and maintenance cost is reduced.

So why *VDM?*

One of the easiest languages to learn

- Lots of formal languages are very unusual or use special techniques
- VDM-SL is a familiar "model based" language – data types, state data, functions
- Simple ASCII text, not "mathematical squiggles"

One of the longest established (30+ years)

- VDM isn't new and untried
- Lots of industrial use, not an academic toy

Has reasonable tool support

- "VDMTools" (Kyushu Uni., Japan, open source)
- "VDMJ" (command line tool, open source)
- "Overture" (based on VDMJ, open source)
- "VDM VSCode" (based on VDMJ, open source)

Has an active user/teaching community

- VDM is very popular in Japan, taught in Aarhus and Newcastle universities
- Active EU research and tool development community
- Consultancy support/training available

Where can I find out more?

Background:

- Wikipedia: https://en.wikipedia.org/wiki/Vienna_Development_Method
- Overture site: https://www.overturetool.org/publications/books/

The Tools (free):

- *VDM VSCode*: https://github.com/overturetool/vdm-vscode/wiki/
- *VDMJ*: https://github.com/nickbattle/vdmj
- *Overture*: https://www.overturetool.org/
- *VDMTools*: https://github.com/vdmtools/vdmtools

The Wikis:

- *VDMJ:* https://github.com/nickbattle/vdmj/wiki/
- *VDM VSCode:* https://github.com/overturetool/vdm-vscode/wiki/

The Documentation:

- https://github.com/overturetool/overture/tree/master/documentation
- https://github.com/nickbattle/vdmj/tree/master/vdmj/documentation