Psychology of Programming

Thiago Rocha Silva (trsi@mmmi.sdu.dk)
Associate Professor

What does it mean to be a programming language?

(Shaw, 2021)

Myths

- Myths are narratives that help us understand our place in the world. They are intended to capture deep truths while providing insight.

> "A myth isn't simply a refutable claim; in its purest form it's a heroic narrative illuminating the human condition. It can establish a model for behavior and uphold social traditions. Myths have enduring appeal—people want to believe in them, notwithstanding contradictory evidence. This 'wish to believe' elevates a myth from a simple, objectively testable statement to a phenomenon whose appeal and persistence require interpretation." [Jaspan et al 2009]

Most cultures have myths

- Most of you grew up with myths, often different ones.

Myths commemorate our ancestors and origins
Myths teach us about the gods to explain our universe
Myths celebrate our heroes to inspire and unite us
Myths are a rich blend of fact, fiction and the fantastical,
religion, philosophy and history
Myths are told and retold, passed on through the generations

Witi Ihimaera

The software myths

- Our beliefs about software are rooted in an idealized view from a simpler era:

  Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.

  Software is made by (just) composing program modules.

- From this arise a number of myths.

The software myths

Professional programmers *create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.*

Software is made by (just) composing program modules.

se mythos	*se pragmos*
Programs are written by highly skilled professional programmers.	Vernacular software developers vastly outnumber professional developers. Professional developers now (mostly) do things other than write code.

The software myths

Professional Programmer

Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.

Software is made by (just) ***composing program modules****.*

The Code IS the Software

se mythos

Software is simply the symbolic program text.

se pragmos

Software systems are coalitions of many types of elements from many sources with sketchy specifications and unannounced behavior change.

The software myths

Professional Programmer

Mathematical Tractability

*Professional programmers create software by writing code in **sound programming languages** to satisfy a given formal specification and verify that the program is correct.*

Software is made by (just) composing program modules.

The Code IS the Software

se mythos

Soundness of programming languages is essential.

se pragmos

Task-specific expressiveness is more important than completeness or soundness.

The software myths

Professional Programmer

Mathematical Tractability

*Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is **correct**.*

Correctness

Software is made by (just) composing program modules.

The Code IS the Software

se mythos	*se pragmos*
Correctness of software is also essential.	Fitness for task usually matters more than absolute correctness.

The software myths

Professional Programmer

Mathematical Tractability

*Professional programmers create software by writing code in sound programming languages to satisfy a given **formal specification** and verify that the program is correct.*

Specifications

Correctness

Software is made by (just) composing program modules.

The Code IS the Software

se mythos

Thus formal specifications are also essential.

se pragmos

Much software is developed to discover what it should do, not to satisfy a prior specification.

The software myths

Professional Programmer

Mathematical Tractability

Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.

Specifications

Correctness

Software is made by (just) composing program modules.

The Code IS the Software

The Purity myths

The software myths

Professional Programmer

AI Revolution

Mathematical Tractability

Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.

Specifications

Correctness

Software is made by (just) composing program modules.

The Code IS the Software

se mythos

...a new one going around...
Artificial intelligence (actually machine learning) is so special that it will break normal software development.

se pragmos

Artificial intelligence has long been an incubator for disruptive programming ideas; the issues of current concern have variants in conventional software, where there are established ways to deal with them.

The software myths

Professional Programmer

AI Revolution

Mathematical Tractability

Professional programmers create software by writing code in sound programming languages to satisfy a given formal specification and verify that the program is correct.

Specifications

Correctness

Software is made by (just) composing program modules.

The Code IS the Software

The world has evolved, the old myths no longer suffice

The Professional Programmer myth

se mythos

- *Professional developers* are:
  - trained professionals
  - with math and logic skills
  - who will spend time learning languages and their models
- They write code to specifications or APIs.

se pragmos

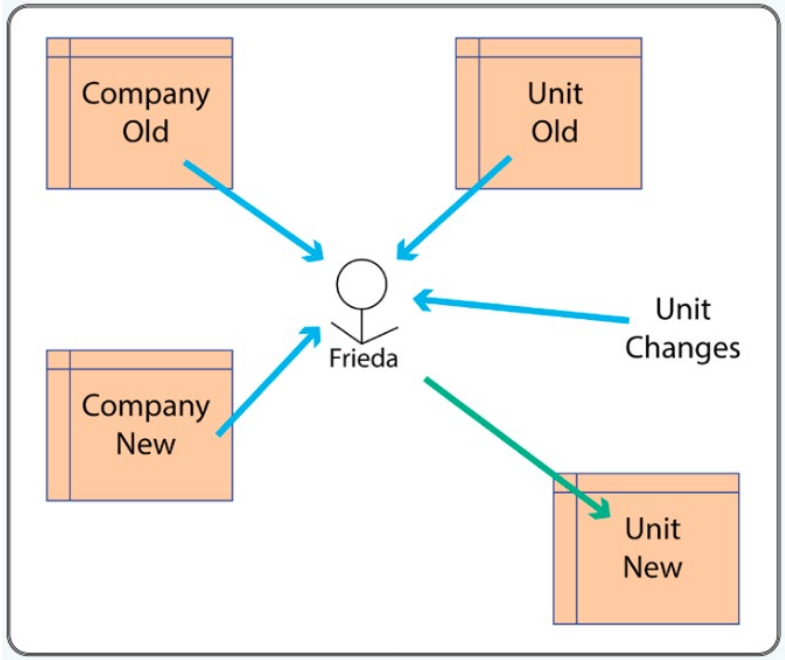
- *Vernacular developers*:
  - create software for own goals
  - start with idea, not a spec
  - may not think of themselves as “doing programming”
  - may not have/want extensive training in programming
- They *vastly* outnumber professional programmers.

Vernacular developers

- Many people without professional software training create software for their own goals.
- These vernacular developers outnumber professional developers, by 5-10x.
- They think in the concepts of their own domains, not software concepts.
- They often develop software by trial and error, refining their goals as they go.
- They write scripts, macros, sets of formulas and constraints; they use graphical tools, exploratory programming, follow-me scripting.

Frieda's department budget

- Frieda handles her department's budget spreadsheets. Each year she must adapt the company's new budget to her department's needs.
- She must remember last year's changes, discover how the company version changed, and adapt the new version to her department.
- She does this by re-discovering last year's changes and by trial and error.



(Burnett & Myers, 2014)

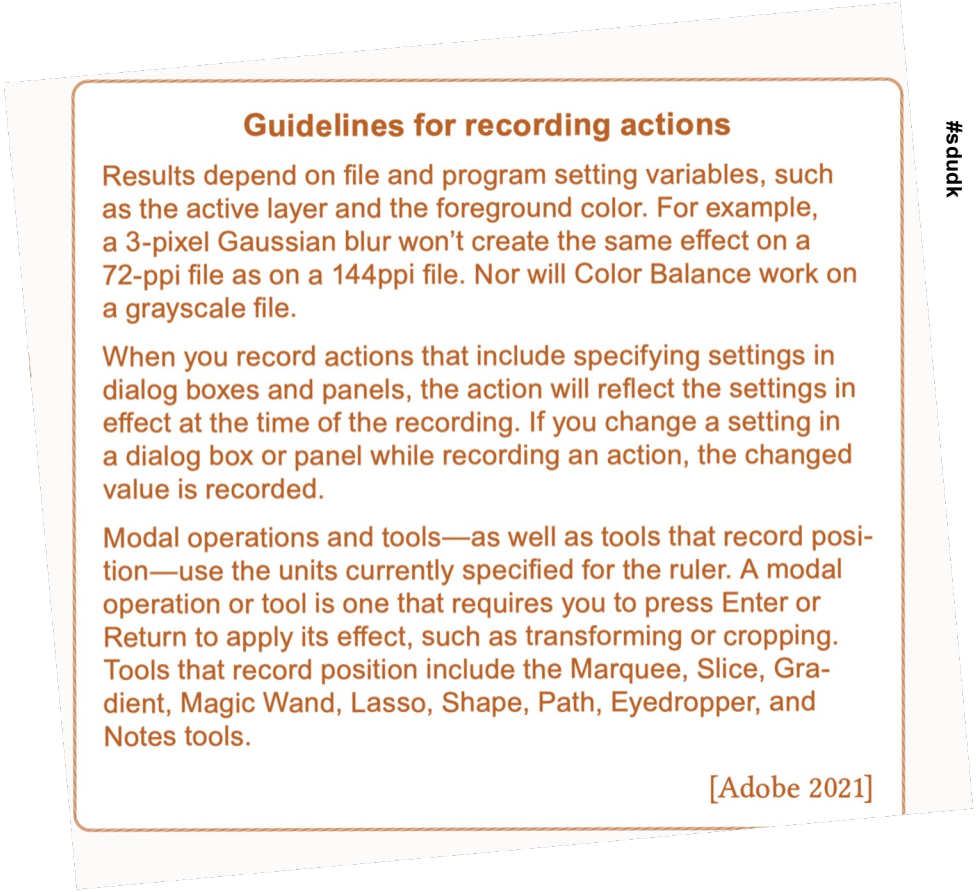
Scientific data analysis

- Scientists create software to help them understand phenomena.
- The code involves lots of scripting and data handling, plus calls to specialized libraries.
- They develop incrementally with spot checks at every step, often with visualizations.
- This exploratory development style develops the code and confidence in the code in parallel.

> When people use Jupyter notebooks or the RStudio IDE to develop data analysis pipelines, a common workflow is:
>
> 1. Load the data and visualize it.
> 2. That looks OK, so filter the data and check there aren't any NAs ("not available", or result not returned).
> 3. That looks OK, so replace the check for NAs with a summarization step and visualize the groups.
> 4. Those groups look OK, so remove the visualization, add a pivoting step, and check the row and column totals.
>
> …and so on…
>
> [Wilson 2021]

Photoshop scripts

- Photoshop allows definition by example of “actions” for future replay.
- But current context is captured in unexpected and unhelpful ways.
- These scope issues would benefit from programming language concepts.



Guidelines for recording actions

Results depend on file and program setting variables, such as the active layer and the foreground color. For example, a 3-pixel Gaussian blur won’t create the same effect on a 72-ppi file as on a 144ppi file. Nor will Color Balance work on a grayscale file.

When you record actions that include specifying settings in dialog boxes and panels, the action will reflect the settings in effect at the time of the recording. If you change a setting in a dialog box or panel while recording an action, the changed value is recorded.

Modal operations and tools—as well as tools that record position—use the units currently specified for the ruler. A modal operation or tool is one that requires you to press Enter or Return to apply its effect, such as transforming or cropping. Tools that record position include the Marquee, Slice, Gradient, Magic Wand, Lasso, Shape, Path, Eyedropper, and Notes tools.

[Adobe 2021]

The The Code IS the Software myth

se mythos

- Big programs are closed systems composed from smaller code modules that interact via procedure calls.
- Data is incidental.
- Behaviour depends only on code plus explicitly invoked libraries and system calls, which change only with the knowledge of the programmer.

se pragmos

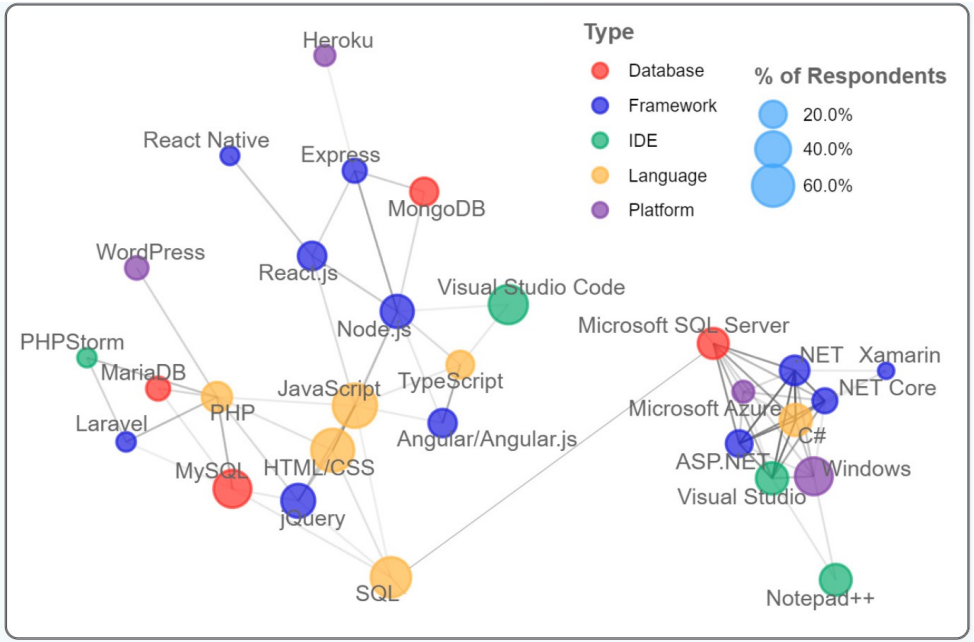
- Modern software is coalitions of many types of components and supporting tools.
- Data is significant.
- Writing code is a small part of professional developers' jobs.
- The vastly more numerous vernacular programmers mostly use tools other than general-purpose languages.

Professional developers…

- Software is not just closed systems with known, specified parts.
- Systems have scripts, huge datasets, code in multiple languages, real-time datafeeds, automatic updates, distributed processing,…
- Components are composed with scripts, generation tools, dynamic selection from under-specified, third-party sources.
- These are *coalitions*, not systems.

Technology ecosystems

- Software development relies on ecosystems with platforms, frameworks, IDEs, and databases as well as languages.
- These cluster around applications.
- Most the languages are DSLs, not general-purpose languages. Most have not been rigorously designed.



Tools used by StackOverflow community for web development and Microsoft technologies.

Programming-in-the-large

- Almost half a century ago DeRemer and Kron recognized that configuration of modules into systems is not supported by traditional languages and offered a provides-requires language.
- This has evolved into notations for system configurations and software architectures, but the integration with programming languages remains tenuous.

[Freeman 2021]

Vernacular developers…

… may not have/want extensive training in traditional languages

… use many tools and notations: spreadsheets, data schemas, markup, visual web tools, constraint systems, domain-specific languages, scripting, scientific libraries, graphical notations

… may not have static symbolic text for their programs

… need support that matches their domain models

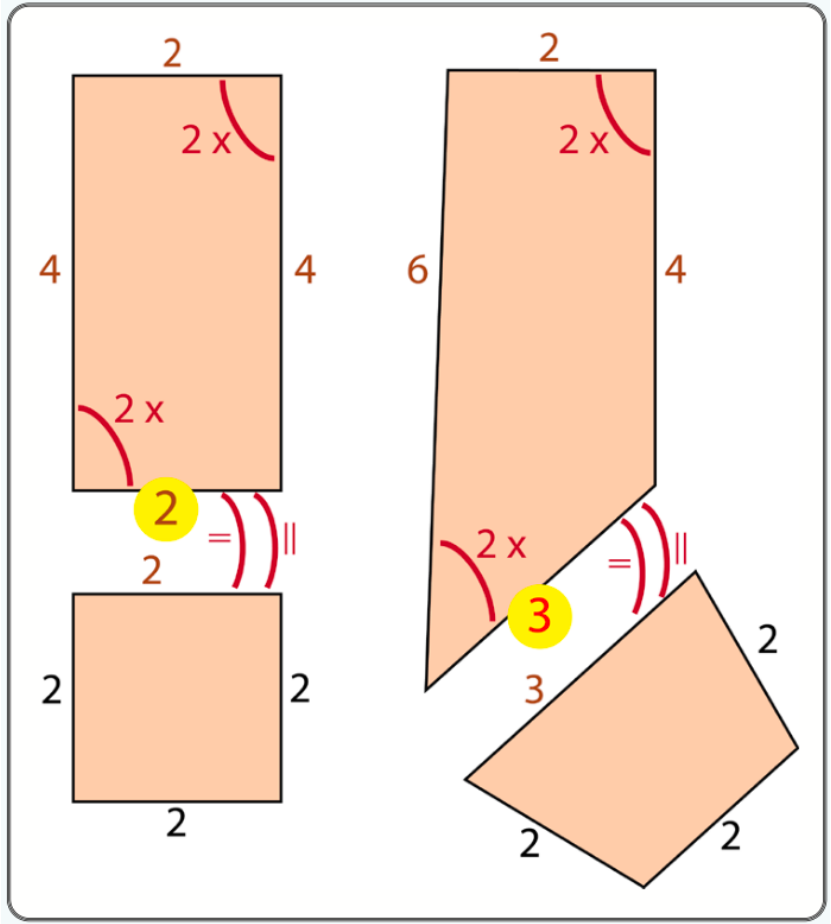
… need quality language support for their notations and tools

Web mashups

- Web mashups combine data from different sources to visualize in a browser.
- In 2008 election there was concern about delays at polling places.
- This mashup displayed wait times based on crowdsourced reports in stylized form on SMS, Twitter, telephone.

AutoCAD

- AutoCAD's core is a sophisticated parametric constraint system for graphically defining 3D models.
- The constraints may be formulas that may interact with other constraints.
- Toolsets for specific professions include standard parts, generators for 3D elements, analysis tools.
- Writing the constraints is very much a programming activity.

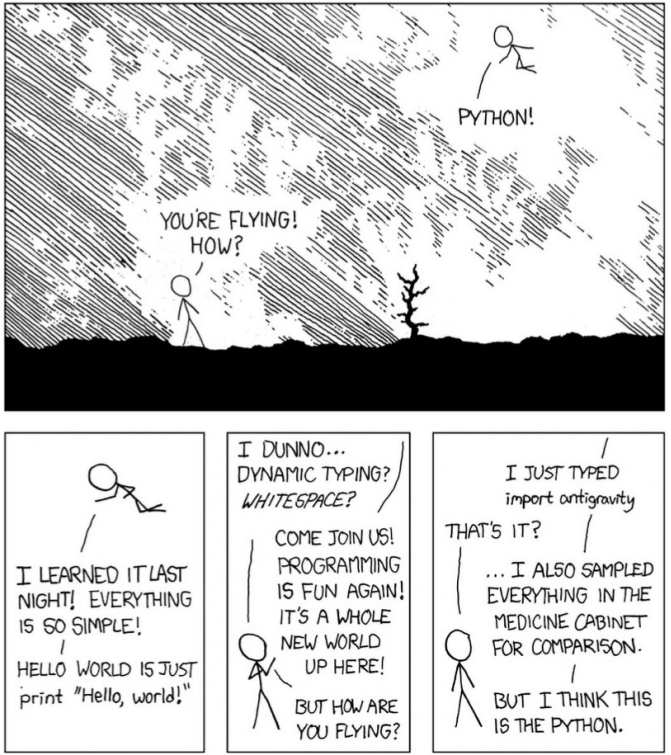


Simply changing the yellow 2 to 3 dramatically changes the figure.

Low-code development platforms

"The shortage of highly skilled…. Software developers has given rise to a new generation of low-code software development platforms [that] enable the development and deployment of fully functional applications using mainly visual abstractions and interfaces and requiring little or no procedural code."

- Gartner sees 65% of applications developed this way in a few years.
- These tools need excellent language support to mitigate the obvious risks.



The Purity myths

- Mainstream traditional programming languages research focuses on *symbolic notations* with *precise specifications* and well-defined semantics that support provably correct solutions to *well-specified* problems.
- This has given rise to a trio of myths, that

Mathematical Tractability of programming languages and

Correctness of software are essential, and hence that

Specifications, preferably formal, are also essential.

The Mathematical Tractability myth

se mythos

- Languages are general, rich, sound, complete, and have rich abstraction mechanisms.
- Reasoning uses formal logics.
- Using the languages requires good math and logic skills, so it's assumed programmers have these skills.

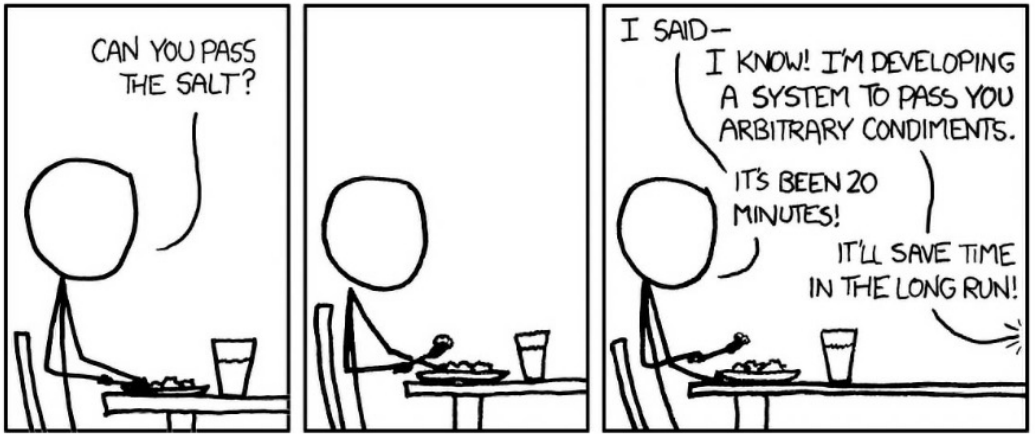
se pragmos

- A minority of professional programmers studied – let alone retain – formal math.
- Languages support the code in modules, but not high-level system abstractions.
- Domain-specific languages trade generality for power.

A senior developer, and Don Syme[1], on math requirements

> *"Over several years, I've tried and failed to learn functional programming because that means learning a big pile of applied category theory. Despite claims that I can just use the parts I do understand, I find that reusing common libraries, like web serving, entails understanding the advanced stuff because it's revealed in their interfaces."*

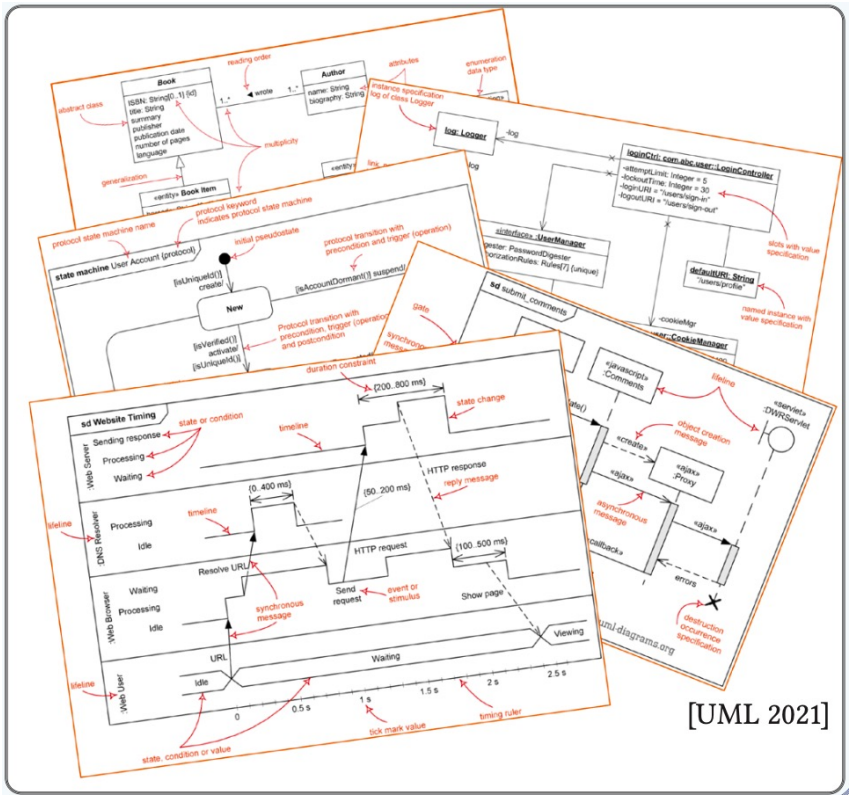
If you add [feature], *"programmers uninterested in [higher math] are disempowered. I don't want F# to be the kind of language where the most empowered person in the discord chat is the category theorist."*



[1](Syme, 2021)

Unified Modeling Language

- The need for mathematical tractability extends beyond the module boundary to the properties of the system as a whole.
- Each notation in the UML suite had a purpose and formal basis, but the suite was incomplete, didn't handle interactions between languages, and lacked connections to the code.

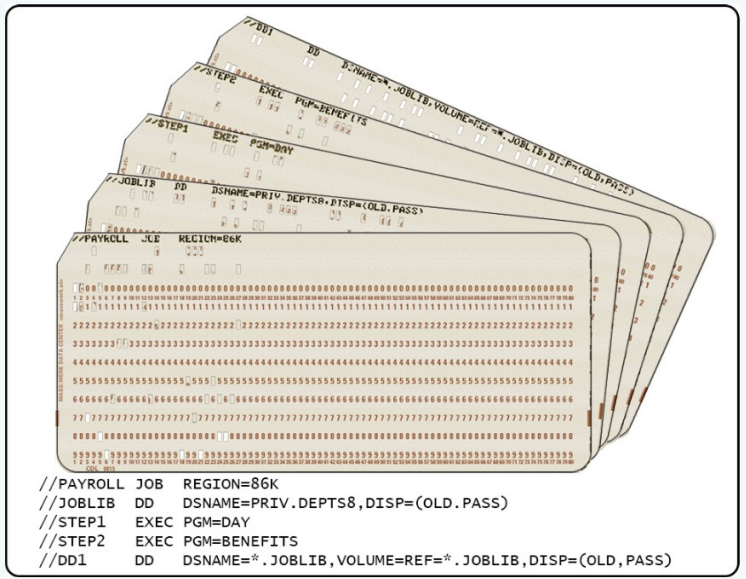


IBM 360 Job Control Language

“The worst computer language ever”

JCL was a scripting language for batch jobs, so hard to learn that people copied working scripts blindly. It forced all programmers to use a second language. Its operator set (verbs) was not well matched to the complexity of the task. It provided no iteration and almost no branching, and the litany of shortcomings goes on.

“The biggest flaw of all was that JCL is indeed a programming language, but it was not perceived as such by its designers” (Fred Brooks)

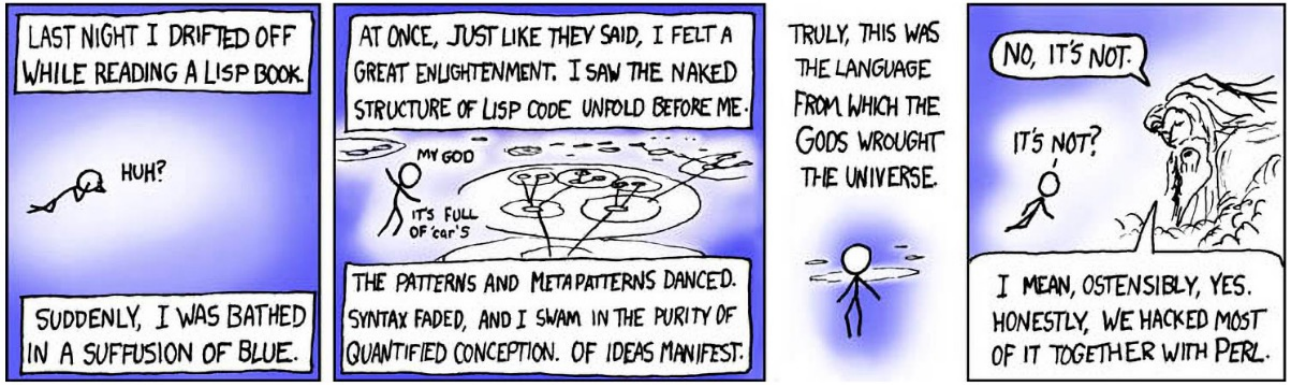


```
//PAYROLL JOB  REGION=86K
//JOBLIB  DD   DSNAME=PRIV.DEPTS8,DISP=(OLD.PASS)
//STEP1   EXEC PGM=DAY
//STEP2   EXEC PGM=BENEFITS
//DD1     DD   DSNAME=*.JOBLIB,VOLUME=REF=*.JOBLIB,DISP=(OLD,PASS)
```

[IBM 1971 p.114, Landsteiner 2021]

The generality-power trade-off

- In striving for generality, languages miss the opportunity to provide formal rigor for the domain-specific languages that support particular applications or programming methods.
- Rigor and formality are useful at many points on the generality-power trade-off curve.



The Correctness myth

se mythos

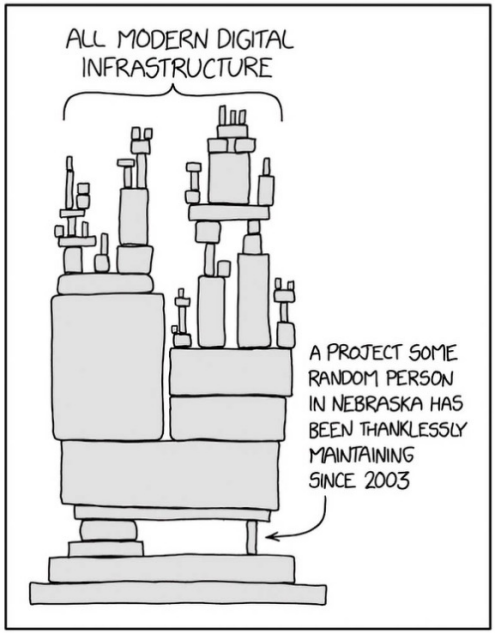
- Software can be provably correct.
- *Specifications* are sufficient, complete, homogeneous, static, and purely functional.

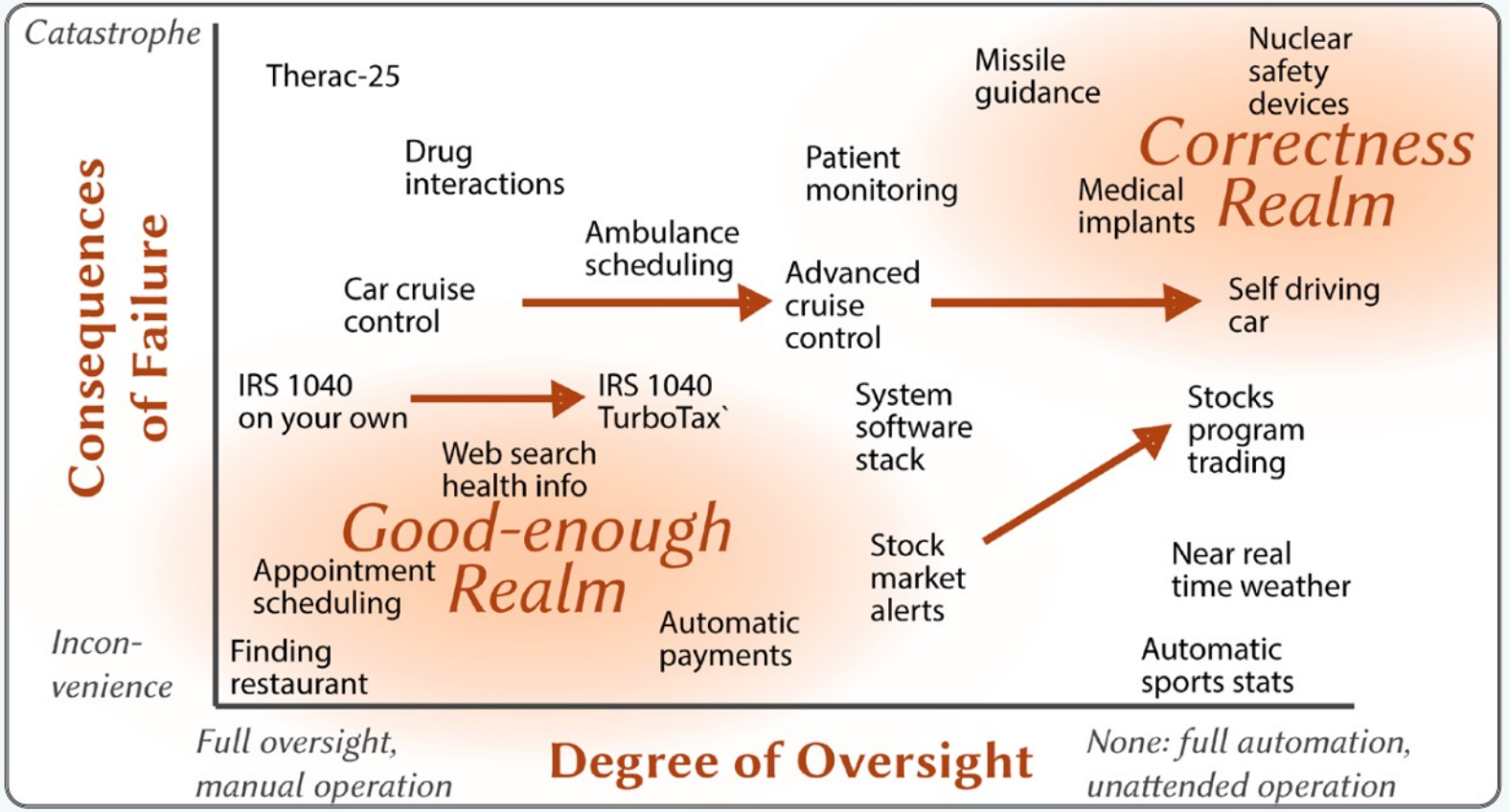
se pragmos

- Proving correctness is usually too expensive. So are full formal specifications.
- *Credentials* are incomplete, heterogeneous, evolving, extra-functional, and non-monotonic – what we know. The information has variable confidence and provenance.

“Correctness” *vs* “Fitness for intended purpose”

- Specifications require certainty, real software is uncertain.
  - Even verified modules execute in unverified environment.
  - Understanding of requirements coevolves with development.
  - Much software is “in the cloud” and updates without notice.
  - Software in physical systems inherits physical uncertainties.
  - Solutions to societal problems don’t have consensus definitions.
  - Acquiring specifications has costs.
- “Good enough” depends on consequences of failure and likelihood that a problem will be caught before failure.
- Assurance cases for compliance with safety standards rely on combinations of test cases, proofs, and human judgment.





[Shaw 2015, 51:27–52:57]

Vernacular developers…

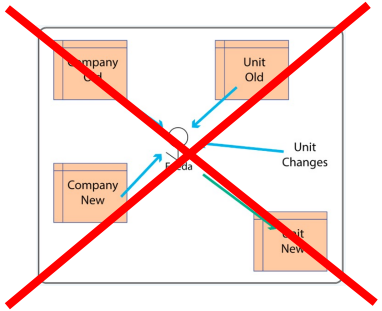
… often program to learn what they really need, working *toward* a specification rather than *from* one

… may have little training or systematic support for reasoning about programs

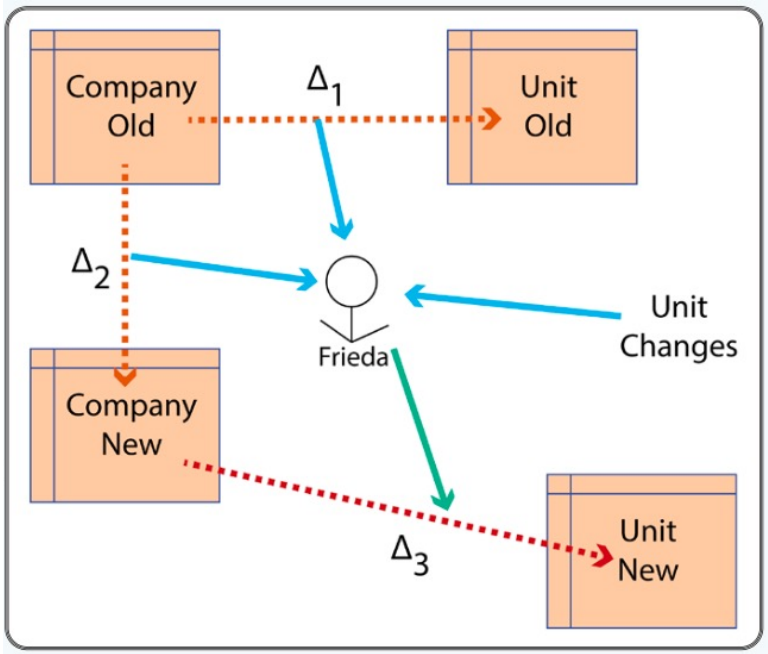
… without a specification, couldn't use many correctness techniques anyhow

… are left with little alternative to look for plausible behaviour

Frieda's department budget, revisited



- Frieda adapts the company budget to her department each year.
- She must reverse-engineer the changes and re-apply where she still can, otherwise make new changes.
- Cut-and-paste doesn't work because of the variable binding rules.
- She is really composing two transformations, but gets no help with seeing the transformations and defining the new one – not even the concept.



(Burnett & Myers, 2014)

The Specifications myth

se mythos

- Full, preferably formal, specifications are possible.
- Some kind of requirement or specification should precede writing software.

se pragmos

- People often write programs to *discover* the requirements, not (just) to *implement* them.

Scientific data analysis

- Scientists create software to help them understand phenomena.
- To figure out if they're doing the right thing, they do spot checks at every step, often with visualizations.
- The checks are simple but depend on the scientist's insight.
- Spot checks don't leave traces and are not re-runnable, especially if they're visualizations.
- Unit testing isn't designed for this.

Writing code to a specification … lets you check the outputs easily. However, if you're simulating a rapidly-rotating black hole, the entire reason you're writing the code is that there isn't a closed-form equational solution.

So how do scientists create software to help them understand a phenomenon and figure out if that software is doing the right thing? … spot checks … applying those heuristics always depends on the data scientist's evolving insight into the data in question.

Greg Wilson

Exploratory programming

- Studies of vernacular developers show a pattern of **incremental, ad hoc, opportunistic exploration** to understand their problems fully as they write their programs.
- Traditional programming languages serve the *problem-solving* aspect of design, but the ***problem-setting*** side is neglected.
- Schön views the problem-setting aspect as “a conversation with the materials”.

> “I shall consider designing as a conversation with the materials of the situation.... A designer makes things. Sometimes he makes the final product; more often he makes a representation—a plan, program, or image—of an artifact to be constructed by others. He works in particular situations, uses particular materials, and employs a distinctive medium and language.... He shapes the situation in accordance with his initial appreciation of it, the situation ‘talks back’, and he responds to the situation’s back-talk. In a good process of design, this conversation with the situation is reflective.” [Schön 1984]

The AI Revolution myth

se mythos

- AI (really, machine learning) systems are so different from conventional systems that we need a brand-new software discipline to cope with them.
- Data dominance, opacity, nondeterminism, inadequate specifications, unpredictable component interaction, …

se pragmos

- AI has a long history as an incubator of disruptive programming techniques.
- ML challenges resemble things SE has seen before.
- Data is more prominent.

AI as incubator for programming techniques

- List processing
  - In the late 1950s, giving up half of memory for links was stunning.
- Mathematics as a programming language
  - That was LISP, too – functional programming
- Heuristic search
  - Search was AI mainstay long before the Internet
- Exploratory programming
  - Expert systems written in productions (condition-action pairs)
- Design as problem-solving
  - From Simon, eclipsing the complementary aspect of problem-setting.

Machine learning

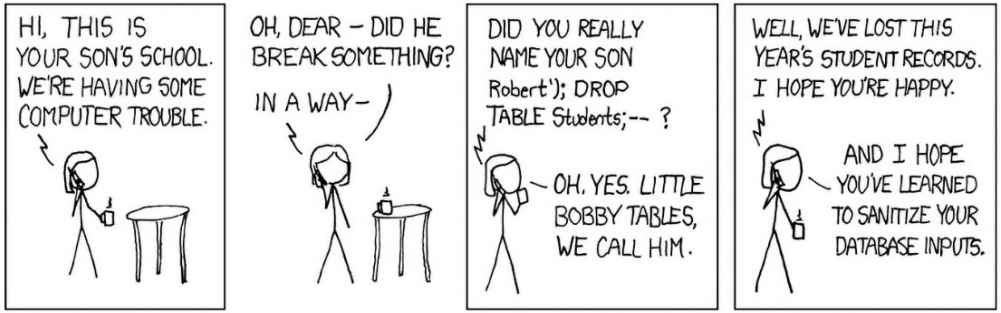
- Concerns about machine learning involve data dominance, opacity, nondeterminism, inadequate specs, unpredictable component interaction, dynamic change, correctness, and other extra-functional properties.
- The software field has addressed similar issues in the past. We can draw on those techniques to develop solutions.
- By embracing the inherent incompleteness and uncertainty of these systems, perhaps we can break free of our myths.

Collection and curation of large datasets

- The Code IS the Software myth ignores data, but … databases ….
- Data quality requires intense curation against many threats
- incorrect data: input errors, sensor failures, malicious errors
- ill-formed data: jpegs when you expect natural language
- inconsistent data: failure to meet integrity constraints

These are familiar problems, with new wrinkles in ML

Bias: even accurate data may be the result of biased sources or algorithms



Electronic health and medical records

- High error rates in electronic medical records
  - ❖ In patient review of doctors' notes, >20% find mistakes (40% serious)
  - ❖ Patient record matching as low as 80%, as low as 50% between hospitals
- Systematic bias in diagnoses, influenced by social and insurance factors
- Electronic medical vs health records
  - ❖ Medical records' provenance is certified professionals or devices
  - ❖ Health records from patients and their devices can be useful, but less trusted
    - Self-reports are unreliable
    - Wrist-worn HRMs: 95% of readings within only 35-40 bpm of ECG
  - ❖ Tracking confidence and provenance of data is required

Unnerving aspects of machine learning

- Opacity: ML algorithms can't explain their outputs.
  - ❖ We can't actually understand conventional components, either.
- Nondeterminism and dynamic change from fresh data.
  - ❖ Ordinary embedded and third-party components are also dynamic.
- Lack of specifications: ML models phenomena that data is about.
  - ❖ Real-world systems have credentials rather than specifications.
- Unpredictable effects: hidden dependency, feature interaction.
  - ❖ Hidden dependencies from coupling have been known for years.
- Criteria for correctness and bias.
  - ❖ Is the standard accuracy, fitness for task, or societal norms?

Old models, new parameters

- In “Information Rules”, Shapiro and Varian faced similar questions about the “new economy” of software, which was said to invalidate traditional economics. They disagreed.
- We should take a cue from them: adapt established techniques: modularity, localization, firewalls, checkable assertions, architecture, feedback, fault tolerance, graceful degradation, …

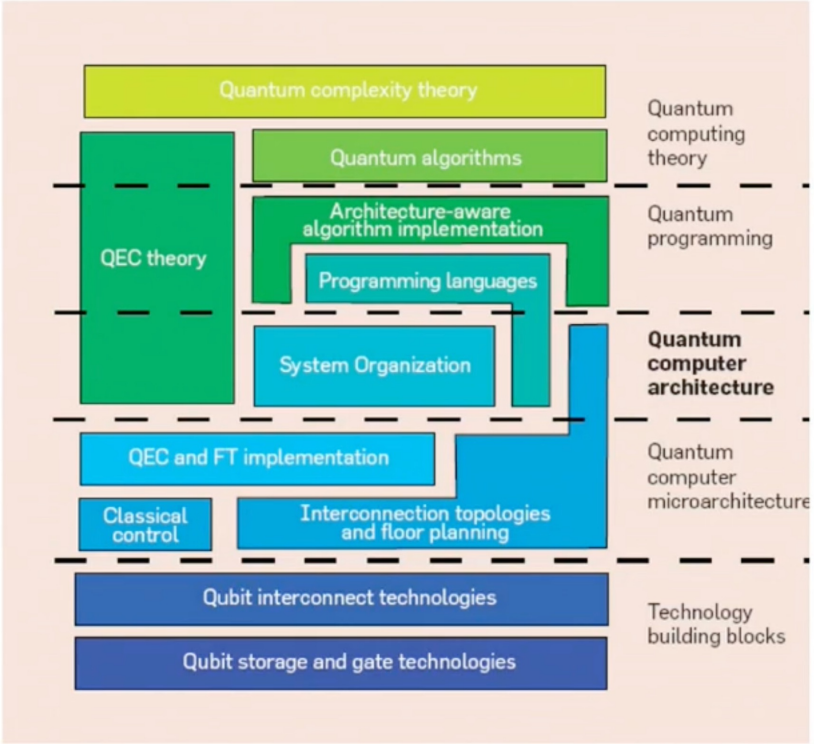
> … we kept hearing that we are living in a “New Economy.” The implication was that a “New Economics” was needed as well, a new set of principles to guide business strategy and public policy. But wait, we said, have you read the literature on differential pricing, bundling, signaling, licensing, lock-in, or network economics? … Our claim: You don’t need a brand new economics. You just need to see the really cool stuff, the material the didn’t get to when you studied economics. … Even though technology advances breathlessly, the economic principles we rely on are durable.
>
> Shapiro and Varian

Bringing ML into the SE fold

- ML brings new kinds of elements to software systems.
  - ❖ Much innovative engineering in ML is required to bring them from research prototypes to routinely-used components.
- SE has rich concepts and theories that apply broadly.
  - ❖ These offer a starting point for innovating in the integration of ML.
- Software that incorporates ML elements is still system software.
  - ❖ ML should continue engineering of the ML elements.
  - ❖ SE should extend our established techniques to integrate ML elements.

Quantum Computing

- The AI revolution will not be the end of new computing paradigms.
- Quantum computing is waiting in the wings with new challenges for programming languages and system design.
  - ❖ Levels of abstraction?
  - ❖ Semantics?
  - ❖ Probability clouds as results?
  - ❖ Correctness?
  - ❖ Quantum entanglement?



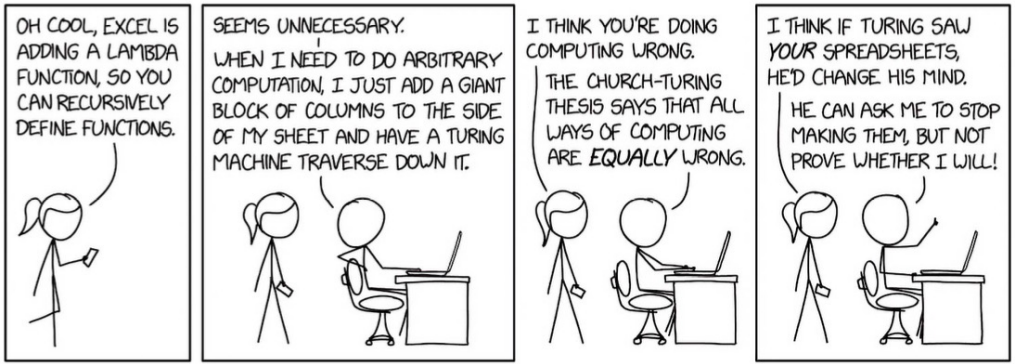
“Our myths should inspire us, but they should not hold us captive” (Shaw, 2021)

Our myths should inspire us, but they should not hold us captive

- Myths should give insights to better understand our world.
- Myths arise from a deep need for certainty.
  - ❖ Certainty can be found within formal systems.
  - ❖ Formal systems were, are, and will remain important.
- The world, however, is uncertain.
  - ❖ It’s messy, nondeterministic, ambiguous, exploratory, human.
  - ❖ Certainty from formal systems does not translate with confidence.
- We should support the inherent uncertainty in practical software.
  - ❖ We should also continue to aspire to precision and completeness.

Opportunities for programming language design

- Support the **background and mindset of the intended users**.
  - ❖ Hide complexity and higher math (domain-specific abstractions?)
- Support **exploratory programming** that seeks **understanding** as well as programming to satisfy specifications
  - ❖ Develop techniques for deciding how good is “good enough”
- Support scripting, mashups, constraints, graphical tools, domain-specific languages.
- Enrich type systems to handle untidy data and track provenance.



Blackwell’s Attention Investment Model

(Blackwell, 2002)

Cognitive features of programming

- Loss of **direct manipulation**
  - The cognitive benefits of direct manipulation arise partly from the fact that **image-based representations mitigate the "frame problem"** in cognitive science.
  - If a planning agent maintains a mental representation of the situation in which it acts, the process of planning relies on the agent being able to **simulate updates to the situation model**, and thereby **anticipate the effects of potential courses of action**.
  - Planning is only possible if the **scope of effects** of a given action can be **constrained**.
  - There must be a **defined boundary** beyond which the action will not have further effects. If there is no basis for setting such a boundary, **any action may potentially have infinite consequences**, and it will not be possible to place bounds on the planning algorithm.

Cognitive features of programming

- Use of **notation**
  - The program is represented using some **notation**.
  - **Concrete** action is that in which there is a **causal relation** between the action and a perceivable state of the world.
  - **Abstraction** results from forming some **representation** of the state of the world – either a mental representation, a linguistic representation or some other representational system.
  - Notational systems are **designed rather than being prescribed** by any necessary constraints, and that the design choices made are subject to **tradeoffs between factors that will facilitate some kinds of cognitive task while inhibiting others**.
  - There is thus **no ideal notation for any programming situation**, only designs that are more or less well suited to the activities of the people doing the programming.

Cognitive features of programming

- **Abstraction** as a tool for **complexity**
  - In a simple programming activity, the user is defining some abstract behavior which is **not directly observable because it will take place in the future**.
  - In contrast to this very simple programming situation, more complex situations can be approached by defining **changes to the notation itself** (e.g., higher-level abstraction modes) so that the user can **extend the vocabulary** with which he or she will then express required behavior.

A cognitive model for abstraction use

- The **Attention Investment Model** accounts for the cognitive challenges arising from these essential features of programming activity: *loss of direct manipulation*, *notation use*, and *development of abstractions*.
- The model is a decision-theoretic account of programming behavior.
- It offers a **cost/benefit analysis of abstraction use** that allows us to **predict the circumstances in which users will choose to engage in programming activities**, as well as helping tool designers to **facilitate users' investment decisions and reduce the risks associated with those decisions**.

Cost in attentional units

- The effort invested in programming can be described as a **nominal amount of "concentration"**, involving an integral of attentional effort over time.
- **Creating a program** requires some amount of **concentration** - an **investment** that has a **cost** in attention units.
- The **payoff** if the program works correctly is that it will **automate some task in the future**, thereby **saving attentional cost** (the user does not need to concentrate on a task that has been successfully automated).
- There is, however, a **risk** that the **investment will not pay off** (perhaps because there are bugs in the program).
- The **decision to write a program** can therefore be framed as an **investment equation**, in which the **expected payoff is compared to the investment and risk**.

Cost in attentional units

- The variables involved in this **cost-benefit analysis** are:
- **Cost**: attention units to get the work done.
- **Investment**: attention units expended toward a potential reward, where the reward can either be external to the model (such as payment for services) or an attention investment pay-off.
- **Pay-off**: reduced future cost, also measured in attention units, that will result from the way the user has chosen to spend attention.
- **Risk**: Probability that no pay-off will result, or even that additional future costs will be incurred from the way the user has chosen to spend attention.

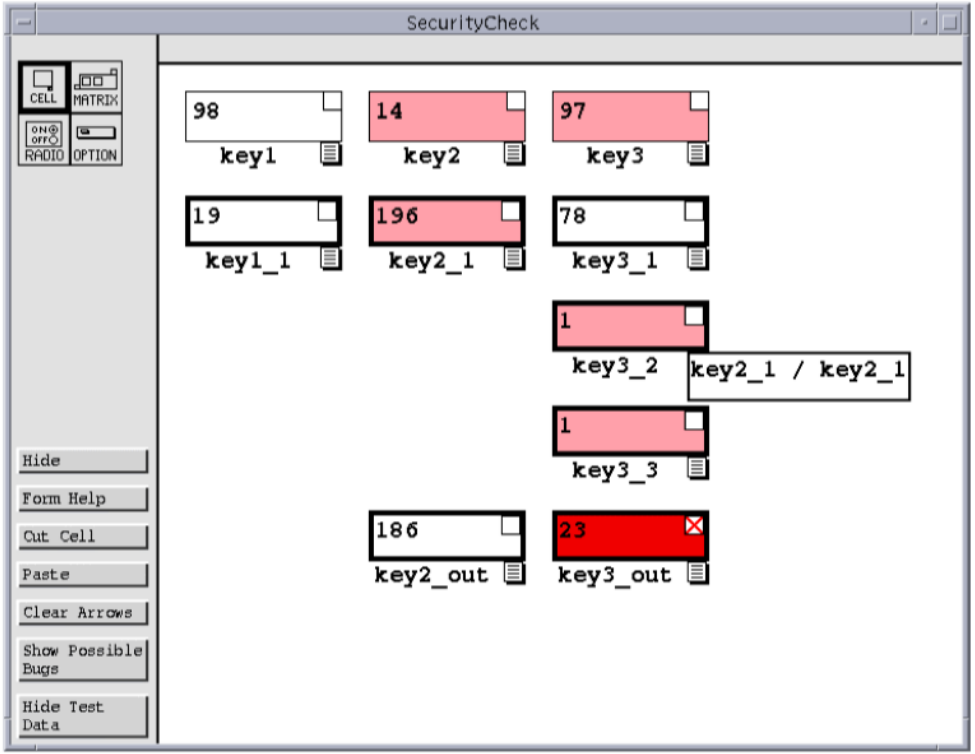
Costs involved in making the decision

- The cognitive model simulates these phenomena using an agent architecture, in which **all possible courses of action are represented by agents competing to be scheduled on a single processing agenda**.
- **Only one agent can be activated at a time**, thereby simulating a **unitary locus of human attention** – we can only attend to a single location (usually by visual fixation) at any time.
- The agent that gets activated is selected according to a **decision criterion that estimates the best cost-benefit return**, subject to the quality of the information (observed or from previous experience) on which that estimate is based.

A few concrete examples

- Products where users **set preferences in advance**.
  - **Choosing options has an attention cost**, and the **payoff is not apparent until the user has experience using the system**. Many users avoid such features because there is learning needed in order to understand the payoff; learning that may prove irrelevant or unhelpful to their primary task.
- If there is a button a user does not know how to use; it will take attention to figure out how to use it. **The risk in expending this attention is that the button will not do anything useful for the user's task, or even that it will cause new problems and thus future costs**.

A few concrete examples



Ko’s EUP Learning Barriers

(Ko et al., 2004)

EUP Learning Barriers

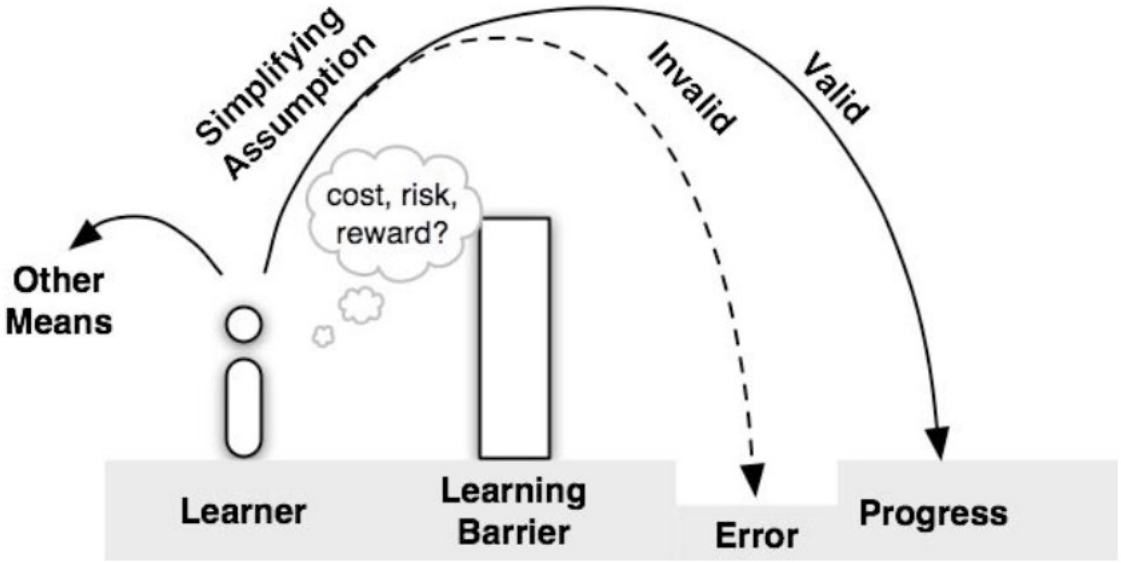
- End users are in constant need of **learning** how to operate and use the EUP system features.
  - ❖ This poses a potential **learning barrier**.
- From an *attention-investment perspective*[1], end users will **weigh the cost, risk, and reward** of overcoming the barrier.
- If the risk of failure outweighs the reward, end users **may abandon the system** for other tools.

[1] A. Blackwell, "First Steps in Programming: A Rationale for Attention Investment Models," IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, Sept. 3-6, 2002, 2-10.

EUP Learning Barriers

- End users may also decide that **progress is worth the risk of failure**.
- In this case, they will make **several simplifying assumptions** about the EUP language, environment, and libraries in trying to acquire the necessary knowledge.
  - If their assumptions are **valid** with respect to the programming system, they will **make progress**.
  - If their assumptions are **invalid** – what's called ***knowledge breakdowns*** - they are likely to make an error.
- Learning barriers are defined as *aspects of a programming system or problem that are* ***prone to such invalid assumptions***.

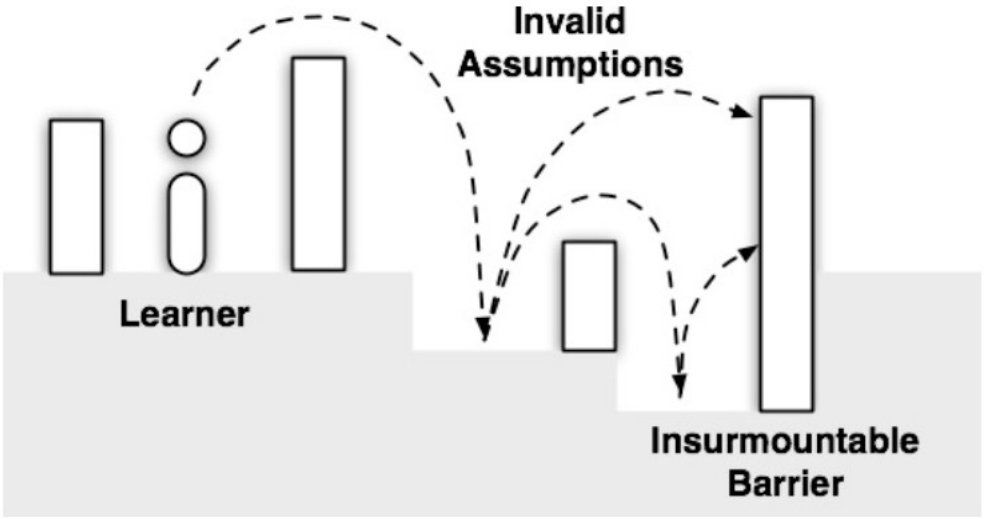
EUP Learning Barriers



(Ko et al. 2004)

EUP Learning Barriers

- End users can also reach ***insurmountable barriers*** which are properties of the EUP system or a programming problem that the the users could not understand **despite considerable effort**.



(Ko et al. 2004)

EUP Learning Barriers

- The **six learning barriers** proposed by Ko et al. (2004) are:
  - ***Design*** barriers
  - ***Selection*** barriers
  - ***Coordination*** barriers
  - ***Use*** barriers
  - ***Understanding*** barriers
  - ***Information*** barriers

Design Barriers

I don’t know what I want the computer to do...

Design Barriers

- ***Design barriers*** are **inherent cognitive difficulties** of a programming problem, **separate from the notation** used to represent a solution (i.e., words, diagrams, code).
- End users are **unable to conceive of a systematic way** to solve a problem.
  - ❖ E.g., when end users are unable to conceive of a systematic way to sort names and their best solution is *“just keep moving the names until it looks right!”*.
- Even when they are able to conceive a solution, they make **invalid assumptions** about their solution.
  - ❖ E.g., end users successfully test one cycle of their algorithm on a single data set on paper, and believed it to be correct for all.

Selection Barriers

I think I know what I want the computer to do, but I don’t know what to use...

Selection Barriers

- ***Selection barriers*** are properties of an environment's facilities for **finding what programming interfaces are available** and **which can be used** to achieve a particular behavior.
- These emerge when learners **could not determine which programming interfaces were capable** of a particular behavior.

Coordination Barriers

I think I know what things to use, but I don't know how to make them work together...

Coordination Barriers

- Coordination barriers are a programming system's limits on **how programming interfaces** in its language and libraries **can be combined to achieve complex behaviors** – what may be called "the invisible rules".
- End users encounter these **when they know what set of interfaces could achieve a behavior**, but **did not know how to coordinating them**.

Use Barriers

I think I know what to use, but I don't know how to use it...

Use Barriers

- ***Use barriers*** are properties of a programming interface that obscure:
  - **in what ways** it can be used,
  - **how to use** it, and
  - **what effect** such uses will have.
- These arose when end users **know what interface to use**, but were misled by these obscurities.

Understanding Barriers

I thought I knew how to use this, but it didn’t do what I expected...

Understanding Barriers

- ***Understanding barriers*** are properties of a **program's external behavior** (including compile- and run-time errors) that **obscure what a program did or did not do at compile or runtime**.
- These emerged when end users **cannot evaluate their program's behavior relative to their expectations**.

Information Barriers

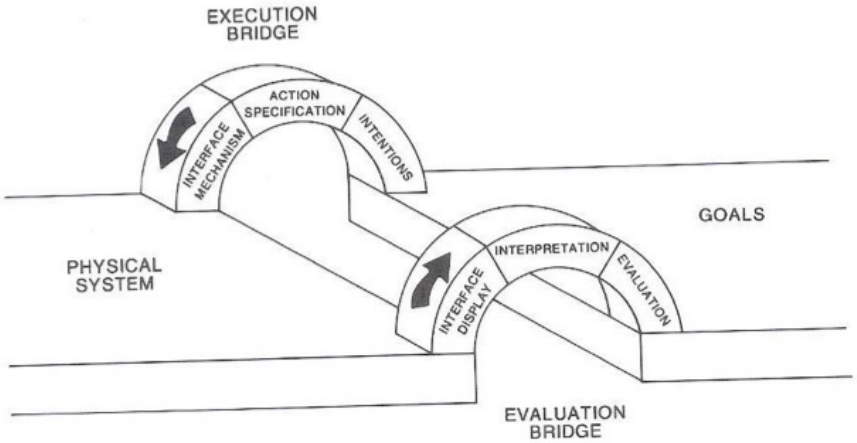
I think I know why it didn’t do what I expected, but I don’t know how to check...

Information Barriers

- ***Information barriers*** are properties of an environment that **make it difficult to acquire information about a program's internal behavior** (i.e., a variable's value, what calls what).
- These arise when **end users have a hypothesis** about their program's internal behavior, **but are unable to find or use the environment's facilities to test their hypothesis**.
  - ❖ E.g., when end users could not find the code that caused a particular error, they delete all of their recently modified code, confident that part of it must be guilty.

Relationship with Norman's Gulfs

- The barriers share characteristics of Norman's **gulf of execution** (the difference between users' intentions and the available actions) and **gulf of evaluation** (the effort of deciding if expectations have been met).



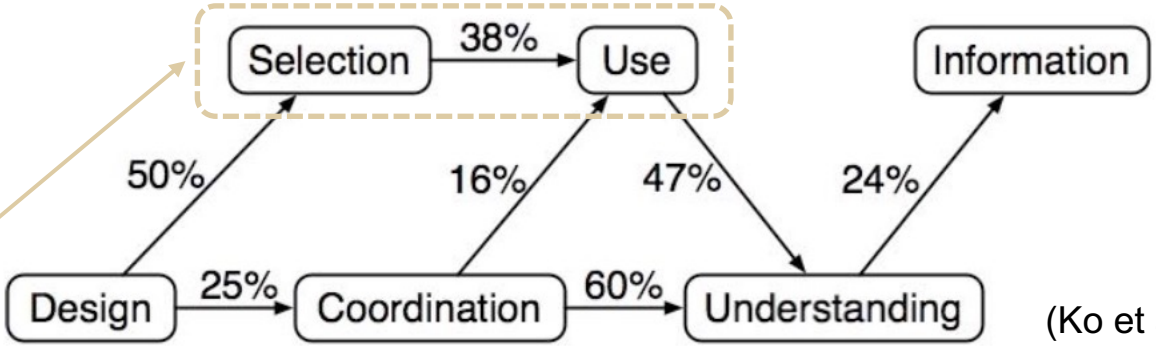
Relationship with Norman's Gulfs

- Three barriers pose **gulfs of execution** exclusively:
  - **Design**: mapping a desired program behavior to an abstract description of a solution.
  - **Coordination**: mapping a desired behavior to a computational pattern that obeys "invisible rules".
  - **Use**: mapping a desired behavior to a programming interface's available parameters.

Relationship with Norman's Gulfs

- Two pose **gulfs of execution and evaluation**:
  - **Selection**: mapping a behavior to appropriate search terms for use in help or web search engines, and interpreting the relevance of the results.
  - **Information**: mapping a hypothesis about a program to the environment's available tools, and interpreting the tool's feedback.
- **Understanding** barriers pose **gulfs of evaluation exclusively**, in interpreting the external behavior of a program to determine what it accomplished at runtime.
- Norman's recommendations on bridging gulfs of execution and evaluation **can be adapted** to programming system design.

How are the Learning Barriers Related?



(Ko et al. 2004)

Preventing invalid assumptions about programming interfaces' ***capabilities*** might avoid assumptions about their ***use***.

Challenges for more learnable EUP systems

- ***Design is inherently difficult***. To overcome **design barriers**, end users need **creativity**. **Programming systems should help scaffold creativity** with salient examples and other forms of inspiration.
- ***Finding behaviors is difficult***. To overcome **selection barriers**, **end users need help searching for behaviors**. Also, as more behaviors are offered, current tools will be increasingly ineffective.
- ***Invisible rules are difficult to show***. To overcome **coordination barriers**, **end users must know a programming system's invisible rules**. Today's systems lack explicit support for revealing such rules, merely implying them in error messages.

Challenges for more learnable EUP systems

- ***Textual programming interfaces are limited***. To avoid **use barriers**, **the feedback and interactive constraints of every programming interface must be carefully designed to match its semantics**. The textual, syntactic representations of today's systems make this goal difficult to achieve.
- ***Behavior is difficult to explain***. Overcoming **understanding and information barriers** requires some **explanation of what a program did or did not do**.

Further Reading

- Mary Shaw (2021), ***"Myths and Mythconceptions: What does it mean to be a programming language, anyhow?"***, Proc. ACM Program. Lang. 4, HOPL, Article 234 (June 2021), 44 pages.
- Alan F. Blackwell (2002), ***"First Steps in Programming: A Rationale for Attention Investment Models"***, In: Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10, IEEE.
- Amy J. Ko, Brad A. Myers, Htet Htet Aung (2004), ***"Six Learning Barriers in End-User Programming Systems"***, In: 2004 IEEE Symposium on Visual Languages - Human Centric Computing, pp. 199-206, IEEE.

Psychology of Programming

Thiago Rocha Silva (trsi@mmmi.sdu.dk)
Associate Professor