Better Stories, Better Languages

François-René Rideau

TUNES Project
`fare@tunes.org`

Abstract

Software tools imply a story. New stories can help invent new tools. Explicit stories are a great meta-tool... and provide a systematic way to improve the design of programming language design.

1998 ACM Subject Classification Design, Human Factors, Languages

Keywords and phrases Storytelling, Cybernetics, Programming Language Design

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Stories are the best way that humans communicate understanding. Maybe it's something hardwired in the structure of our brains; maybe it's related to communication happening through the bottleneck of a sequential signal stream, to be output as a coherent whole, then coherently processed back into understanding. Now not all stories are equivalent; for stories have consequences, and depending on which stories we internalize, we may behave quite differently, with very different outcomes. To freely translate Confucius [2], "when concepts are unfit, plans are unadapted and actions are unsuccessful". Therefore, whichever topic we're interested in, it's important to frame the topic in term of better stories rather than worse ones.

Programming languages are themselves frameworks for telling stories, except that, as Henry Baker once remarked [1]: "Computer language is inherently a pun – [it] needs to be interpreted by both men & machines" We contend that the stories commonly told *about* programming languages can be improved upon; and that this improvement, while it might not yield any new feature at some lower level, can yield completely different programming environments and programs at a higher level.

This essay, based on a 2009 talk given to the Bay Area Lispers, will present a set of stories, some familiar, some less familiar; they will be presented in pairs (or sometimes triples) where a common but sad story is followed by a happier story that improves upon it. Hopefully, one does not have to be convinced by each and every story improvement to get a gist of the overarching meta-story being told.

Let's start with easy story improvements that have a large consensus, then focus on programming languages as ways to specify semantics, then on issues of software robustness, then on non-functional properties of programming environments, before we conclude with a meta-story about the previous story improvements.

2 Easy Software Stories

2.1 Funding

If you ask "how to fund software?" then your concern is that Software costs money to produce. The underlying story is that *software* is owned and sold, which implies human

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

relationships of vendors and customers. The result is proprietary software distributed as closed binaries, unmaintainable by anyone but the vendor (who may or may not be interested in maintenance or competent at it).

If instead you ask “how to fund programming?” then your concern is that starved programmers don’t code. The underlying story is that *labor* is owned and sold, which implies human relationships of contributors and users. The result is Free Software distributed as Open Source, shaped into maintainability by anyone and everyone who is both interested in it and competent at it (quite possibly the competent hired by those more directly interested).

The shift between the two stories is quite familiar, and many people will agree that the first story is a comparatively sad story, and the second story is a happier one. Though not everyone agrees that the second story should wholly displace the first, at least most everyone agrees that its existence, and its being made explicit, was an improvement over its absence, and enabled the collaborative writing of great software that would have been very hard or impossible to achieve before.

2.2 Factorisation

The question “how to decompose programs?” addresses the concern that most software doesn’t fit in one (conscious) brainful. Its underlying story is that software must follow a hierarchical decomposition into components, fully informed by some central designer. The result is top-down management, the waterfall process, and UML diagrams. That’s a sad story.

The question “how to decompose programming?” addresses the concern that many programmers must cooperate. Its underlying story is that software development is done by many partially informed programmers with scarce means of coordination. The result is the growth of networks of programmers, organized around software packages grouped into software distributions. This happier story scales to larger, more elaborate software.

2.3 Quality

The question “how to achieve great software?” addresses the difficulty of producing quality software. Its underlying story is that each component should be designed by the best possible experts, from whom the information will flow to less expert developers; in each specific topic, the less expert people are excluded from making decisions. Development will be divided into teams segregated by expertise, each following the best standards defined for the field. That’s a sad story that leads to the worst demonstrations of Conway’s Law, and to people who can’t communicate with each other by lack of a common language to express mutual concerns.

Ask instead “how to foster better programming?” to address how to do our best and compete with others, learning from our and their successes and failures. The story is that you gain experience through experiments, with experience as an output, rather than expertise as an input. The happier result is to cultivate good incentives, where the scarce resource isn’t information (it’s free) but coordination (it’s expensive), and instead developers sell services to modify an open code base on an open market that values both their experience and their ability to learn new skills, in communities.

3 Programming Languages

3.1 Features

Ask “how to choose features in a programming language?” to address access to all features of the machine. The story assumes that programming languages are for machines. The sad result is a Turing tar pit that matches language features with hardware features.

Ask instead “how to express programming ideas?” to address how to convey all the meanings of the humans. The story assumes that programming languages are for humans. The happier result is to match language features with human cognition and social processes, yielding simpler programming languages.

3.2 Redundancy

Ask “how to deal with repetitive programs?” to address excessive repetition in programs. You then see the programmer as grunt worker to whom the programming language is a given delivery target. The sad result is to theorize repetitions as “Design Patterns”, whereby more programmers endure ever more drudge, manually enforcing consistency in patterns being used.

Ask instead “how to remove programming drudge?” and like Olin Shivers “object to doing things that computers can do”. You see the programmer as abstract thinker to whom language is an evolving platform for expression. The better result is metaprograms implementing domain-specific languages, without which one doesn’t actually take advantage of a language’s often alleged Turing-equivalence.

3.3 Syntax Extension

Ask “how to extend the syntax?”, to address hooking metaprograms into the existing language. Your story assumes a One True Syntax that (as in Common Lisp) will be side-effected to accept extensions. The sad result is ugly interference as all programmers compete to grab pieces of the same global syntax state.

Ask “how to explore useful syntaxes?”, to address the best expression of each program fragment. Your story assumes the scoped specialization of syntax. Your happier result will be more modular ways of locally specifying syntax, as with Racket languages or OMeta.

3.4 Users vs Programmers

Ask “how to address the difference between users and programmers”, and the sad result of focusing on this difference will be dumbed down user interfaces and all-powerful interfaces for developers, with lots of frustration on one side of the segregation and many painful accidents on the other side.

Notice instead that the difference between a programmer and a user is that the programmer knows there’s no difference between using and programming. Recognizing that all computer interaction is programming, trivial as programs may be, your happier result will be a single, simpler interface, in which a same language may be either spoken or written. This integrated interface can then accommodate a continuum of proficiencies in users with many language levels and dialects.

3.5 Programmers vs Language Implementers

Try to address the difference between programmer and language implementer focusing on how writing a compiler is hard and a correct one even harder, and so must be done by specialists while mere programmers use the product. Your sad result will be implementation of each programming language from scratch, reinvent the wheel (badly) each time.

Recognize instead how the previous story improvement works at the meta-level, and that since using is programming, programming is implementing the language to be spoken by the users. Your happier result will be the incremental development of domain-specific languages (DSL) by any regular programmer, using usual techniques for modularity in first-class implementations (see notably PCLSRing).

3.6 Language Implementers vs Language Designers

A similar improvement pattern can be found at yet another meta-level. A sad story would focus on how language design is hard and requires expertise, with the result of segregating language definition and implementation, synchronized around the slow standardization of decades-old designs, with blind spots and bit rot aplenty.

A happier story would unify design and implementation: by recognizing that they are the same activity at different levels of abstraction, one would focus on providing tools to explore the continuum of detail or abstraction vying for declarative, incremental, modular and orthogonal ways of specifying language semantics and implementation strategies.

3.7 Domain Specific Languages

Ask "how to get a specialized language?" to address heterogeneity in software development activities, and your sad result will be segregation of experts by domains each implementing their own badly designed external DSL or "scripting language".

Ask how to specialize conversations between programmers and machines, and your happier result will be a universal language with a algebraic structure of contexts by which it can adapt to any domain using an appropriate internal DSL.

4 Robustness

4.1 Debugging

Ask how to get your programs debugged, to address bugs that need be fixed. Your story is then that bugs are an exceptional situation, and that ad-hoc tools must be retrofitted to handle them. Your sad result is a fixed low-level debugger with ad-hoc debugging information.

Ask how to explore the space of interesting program semantics. Your story is then that programming is not obvious and requires exploration, and that imperfection is the default situation to accept. Your better result is an environment for exploration, with reversible transformations that compile and decompile code through lenses.

4.2 Security

Ask "how to secure existing software?" to address the difficulty of security, which requires specialists. Your sad result will be that developers who don't understand security will try to retrofit it as an afterthought, and/or that professional paranoids who don't understand the application will try to force developers to jump through expensive hoops that don't make sense. The software will forever have leaks that need to be patched, and protection that happens too low a level to be either effective or understandable.

Ask instead "how to build software securely?" to address how security is intrinsic to software design. Your happier result will be designs that account for security, from the start and across the whole system, using properly restrited capabilities rather than as an afterwards for each isolated component. Programmers are educated to pay attention to details that matter.

4.3 Catastrophe Recovery

Ask "how to deal with catastrophes?" to address bad manipulations that cause data loss. Your story assumes error is exceptional and catastrophic. Your sad result is confirm menus, removed file bin, undo button with a few short undo buffer, all of them programmer-intensive add-ons that can only handle simple cases and introduce failure modes of their own.

Ask "how to whooly eliminate catastrophes?" to make bad manipulations unexpressible. Your story assumes that error is an expected, casual occurrence. Your happier result is that you built the entire system on top of monotonic storage that never loses data by construction, so that all programs get in(de)finite undo by default. Extra performance, not correctness, is what requires additional work.

4.4 Interface Documentation

Ask "how to document software interfaces?" to address the fact that function and module signatures in your programming language don't fully formalize software intention. Your story assumes that your programming language is a given, so that what it can't express formally you have to express informally. Your sad result is informal contracts between heterogenous teams, leading to a lot of undetected breakage and manual bug fixes.

Ask "how to agree on responsibilities?" to address cooperation at a distance between multiple teams. Your story is that your teams are a given, and they get to negotiate not just on the interface, but at the meta-level on the formalism to express the interface. Your happier result is that the teams can formalize contracts for the things that they actually care about without having to hardwire useless details that they don't care about; they may have to extend the programming language to get better contracts and types, if after a cost benefit analysis they find the extension of their formalism worth it.[1]

4.5 Resource Management

Ask "how to Arbitrate Resource?" to address the need to maintain invariants when managing shared resources. Your story is that a central dictator is needed who will schedule resource possession between the various processes in your system. Your sad result is a mostly monolithic Operating System or Application Kernel that will manage a static set of resources according to fixed strategies.

Ask instead "how to resolve conflicts?" to address the need to identify or assign owners to each bundle of contested resources. Your story is that of self-enforcing contracts between resource owners, faithfully expressed using linear logic or something equivalent. Your happier result is a system where invariants are not enforced by a runtime kernel managing a fixed

[1] For instance, if you can afford the amount of testing that went into SQLite, you can probably afford correctness proofs instead.

set of resources, but by a type-aware dynamic linker allowing for a rich algebra of resource bundles.

5 Non-Functional Requirements

5.1 Computer Networks

Ask "how to connect computers?" to address the technical and social limits on what a single machine can do. Your story assumes that many different systems have to be put together. Your sad result is complex protocols for remote method invocation, shipping objects around, and manually ensuring that objects encoded here are decoded there into other objects that are reasonably equivalent.

Ask "how to distribute computation?" to address the fact that your computing system is made of many machines. Your story assumes that all these machines can be conceived as part of a single system in which many agents interact (be they hardware, software or wetware processes). The happier result is declarative deployment and content-based addressing of pure data arbitrarily many (or few) copies of which are equivalent by construction.

5.2 Trust

Ask "how to handle mistrust?" to address the need for protection barriers between trust domains. Your story tells of domains managed by a kernel according to a rigid model. Your sad result is a hodge-podge of ad-hoc coarse-grained entities (processes, containers, virtual computers) that are as expensive to deploy as they are inexpressive in modelling your precise requirements.

Ask instead "how to express limited trust?" to address the dynamic creation of bundles of capabilities. Your story is one where every entity is "root" in its own virtual world, and its sub-entities recursively so, by default. Your happier result is one where the programming language embeds support for virtual worlds in which sub-users can be created cheaply, virtualizing just the required functionality with manageable semantics at the proper language level, instead of having to virtualize and replicate all the mostly irrelevant details of a low-level machine at the assembly level.

5.3 Persistence

Ask "how to persist important data?" to address the need to persist important data in the face of hardware or software failure. Your story assumes programmers must manually persist data that is transient by default. Your sad result is ad-hoc filesystems and databases with which programs explicitly communicate through I/O operations.

Ask instead "how to write persistent software?" to address the fact that all data is either important or else not worth a programmer's time. Your story is one where everything is persistent by default, and programmers sometimes explicitly use transients (or linear type annotations) when they want extra performance. Your happier result is orthogonal persistence, wherein every entity in your programming language is implicitly persistent by default, and programmers don't have to care when object contents are spilled from main memory to disk any the more than they usually have to care when they are spilled from L2 cache to main memory, from L1 cache to L2 cache, or from register to L1 cache.

Still, there are infinitely many ways to implement persistence, and no one size fits all, though it is certainly possible to offer sensible defaults and options that cover easy common cases. And so if to address this you ask "how to specify persistence orthogonally to computation?" then your story decouples the semantics of a computation from "non-functional" properties of the underlying programming language implementation. Your even happier result is that your programming language can express first-class virtualization of computations, and possibly dynamic modification or migration of a program's underlying virtual monitor at runtime, allowing for management of computation as part of a well-defined semantic infrastructure rather than with ad hoc informally specified bug-ridden slow administration tools.

5.4 Change

Ask "how to Model a changing world?" to address the issue that mutations happen. The story assumes an object-oriented world where every object is always mutable at any time by default... including by other processes acting in your back. The sad result is imperative programming where everyone lives in fear, no one can ever trust anyone else, except for in expensive short temporary critical sections surrounded by lock operations. Access invariants are global properties of the entire code base that must be manually enforced by coordinating distributed transactions over many source files.

Ask instead "how to model changes to the world?" to address how to compose the transformations undergone by the world. The story assumes a value-oriented world where the values that describe the successive states of any part of the world are always pure immutable data (at least by default), and you can easily reason about programs in terms of how these values are transformed across time. The happier result is functional programming (OCaml, Haskell, etc.), where everything is pure data by default, at least at the base-level. Now if you truly believed in purity everywhere, you'd demand it at the meta-level, too, and never use identifiers and binding forms, only combinators, with explicit effects or monads.

Thus for an even improved point of view, ask "how do I discuss relevant change?", to address how important change happens. The story assumes a change-oriented world where change is first-class, where state is understood as meta-level modularity, where the same code can be viewed as either mutable or immutable depending on the kind of use or reasoning done with it. The yet happier result is a system with code differentiation and integration, and dynamic switch of point of view from direct style to monadic style and vice-versa.

6 Conclusion

6.1 The Grand Challenge

None of these Stories is revolutionary. Many of them are at least implicit in the "Mother of All Demos" [3] and each of them has been foretold in past systems, at least implemented once, though not always optimized, productized and integrated within a working system.

But indeed no *system* embodies all of these happier stories at once. That's an opportunity to do better! Yet, the previous remark shows that what is missing is not technical ability, but a shared vision.

Indeed, many of the properties sought for a computing system are a matter of complete reflection of the system's global semantics. They can't be usefully layered as an add-on extension to an existing system that doesn't already possess these properties. At least not while preserving the transparent availability of the system's existing code base.

6.2 The Meta-Story

If we analyse the sad and happier stories that we proposed, we can identify common patterns. These patterns constitute a meta-story about improving existing stories in general.

One first pattern is as follows: sad stories are about the things created, about the software as a finished product; instead happier stories about the people creating, about software development as a process.

A second pattern: sad stories try to bind good decisions early, entrusting them to a supposed experts possessing superior information, who will impose constraints to be manually enforced by or upon normal programmers; instead happier stories try to ban bad decisions early by making them unexpressible in the interaction design of the system, abstracting away the drudge of enforcing coherence and safety, and enabling normal programmers with a rich and free expressive algebra in this safe space.

A third pattern: sad stories tend to segregate people into castes by supposed prior training or expertise, each with its own exclusive interface and prerogatives denied to people from other castes; instead better stories offers a common interface to all, and selects user contributions a posteriori based on their value to the development process, otherwise without regard to the identity of the contributor, who is supposed to gather experience along the process.

By focusing on processes rather than products, overall algebraic structures rather particular objects and constraints, dynamic interactions rather than static organizations, we can see patterns that are otherwise invisible. That's the essence of Cybernetics[2], and computer scientists could greatly benefit from an increased awareness of it, for it has implications no just in the design not just of programming languages, but also of user interfaces, development management, software and hardware architecture, distributed systems, etc.

References

1 Henry Baker. The Legacy of Lisp, 2005.
2 Confucius. *Analects.* 5th century BC.
3 Douglas Engelbart. The mother of all demos, 1968.

[2] Cybernetics is a look at the big picture of interactions within a system (in this case, the software development ecosystem, that involves both men and machines), that tries to identify on the one hand what are the invariants (conserved quantities such as energy, information, amount of programmer effort, etc.), the monotonically increasing variants (distance to goal, entropy, etc.), and the feedback forces that when negative imply stable convergence towards an equilibrium and when positive imply runaway evolution towards ever more elaborate patterns.