Diseases in Code

Bryan Edds, 2014

An Explanation of Code Diseases, their Prevention, and their Costs by a Self-Described Code Pathologist

Let us take all opportunities to prevent diseases and to clear away existing ones when the opportunities arise so that we may live healthy lives in the service of our purpose.

Table of Contents

What is Code Disease?

A code disease is a pervasive property of a code base that harms or destroys basic ease of software development and maintenance. Often propagating problems up to the business execution and strategic levels, code diseases are costly and risk-inducing. As its strong name indicates, a code disease is very serious and potentially threatening, with its first victims the morale and sanity of the developers who live with it daily, and its final victims the customers who are punished for relying on your business's affected systems.

Code Diseases vs. Code Smells

The subject of code smells is well-covered, so a familiarization is available via the references and citations provided at this link - http://en.wikipedia.org/wiki/Code_smell.

The concepts of code disease and code smell are related, but the former is more concrete. While a code smell indicates the possibility of a problem with code, a code disease is often the indicated problem itself. A code disease is a fundamentally problematic property of a code base that must be at best prevented - or at worst managed and paid for indefinitely.

Like code smells, code diseases are easily sensed during the act of coding. With code disease, like with a biological disease, at minimum there is often a feeling of unease in several ways. A feeling of anxiety often stirs up amidst working with diseased code from the nagging idea that one cannot be sure about what could go wrong. More importantly, there is often a feeling of difficulty in accomplishing tasks that could be done with relative ease. When a disease factor in code can be rectified on the fly, one should proceed by first making the coding task easier through a bit of refactoring, and then thereafter engaging in the task.

But again, though code diseases and smells are related, they are distinct.

Diseases Found in Code

Like smells, there are many diseases found in code. Listed here are the several that are most pertinent to this discussion -

- Structural complexity – most commonly this diseased is expressed by object-oriented design models that inherently couple program behavior and representation - http://fsharpforfunandprofit.com/posts/cycles-and-modularity-in-the-wild/ - but is also by most other imperative design models.
- Inadequate encoding of desired behavior into independently verifiable artifacts - most commonly expressed by inadequate and / or under-utilized type systems, and under-restriction of possible program states brought about by excess mutability.
- Inability to simply reason about program behavior / cause and effect – commonly expressed by overuse / abuse of program state mutation. This is the disease functional programming seems to be most actively and effectively used to fight.
- Unclear, complex, and / or ambiguous program semantics – commonly expressed by code that is over-complicated and / or written without direct purpose to what it is intended to accomplish. Also commonly expressed by ad hoc and underspecified internal APIs.
- Unclear / awkward / overly-imperative programming idioms – The idioms used to design a program are often inconsistent within a code base, with each programmer, possibly migrant with no interest in existing idioms, polluting the stew with his own treasure trove of terrible idioms begotten of C circa 1987.
- Run-time inefficiency – commonly expressed by naïve algorithms in code and wide-spread abstraction inversions, but also found in program's whose execution model are excessively difficult to reason about.
- Inadequate program stress testing – commonly expressed by underutilized, inadequate, and / or late-bound type systems. Can be mitigated by unit tests, but with the additional associated maintenance costs.
- Opacity – commonly expressed when the conceptual structure within a program is not reflected by its code or any generate-able artifact. A simple example is a single UI screen whose design is implemented by straight-line textual code (or worse, code spread across multiple files).

As with opacity, an especially infuriating aspect of many code diseases is that they don't materialize simply by coding or designing overtly incorrectly. Often they come about by failing doing things in what would, to the untrained eye, seem like an overly narrow manner. Like an animal with a compromised immune system, code has an almost punished existence. Being merely left to its own devices, code is a sponge for disease.

Program Disease Matrix

What follows is a matrix describing the occurrence and severity of disease in modern programs where the programs in orange have been built without disease prevention techniques and those in blue have been (here as enabled by F#) -

	Structural complexity	Inadequate encoding of desired behavior into verifiable artifacts	Inability to simply reason about program behavior / cause and effect	Unclear, complex, and / or ambiguous program semantics	Unclear / awkward / overly-imperative programming idioms	Run-time inefficiency	Inadequate program stress testing
Typical Framework / Game Engine	*****	*****	*****	*****	*****	****	****
Typical Enterprise App	*****	*****	*****	****	*****	*****	****
Typical Language Interpreter	*****	****	****	****	*****	*	***
Typical Indie Game	*****	*****	****	****	*****	**	****
Nu Game Engine		*				*	***
Typical Enterprise F# App	**	**	**	*	**	**	**
Aml			*			*	**
OmniBlade (Projected)	*	*	*				**
*where * count := severity of disease*							

From the matrix, it is clear to see that typical, non-trivial applications are riddled with most of the severest diseases. From experience, it is no wild guess to say that it will be highly unlikely that the unhealthier programs will ever see significant reduction in their current disease state. From the author's professional experience across many large commercial software projects, such significant disease profiles are never reversed.

Origins of Diseases found in Code

The most effective tact to address the issue of code disease is prevention, and the less effective tact is removal post hoc. But to execute with either tact, one should start by understanding the common origins of widespread diseases.

Most diseases found in code have origins beyond the tool from which it was authored –

- Deficiencies in programming languages
- Unsound thinking on behalf of the development team
- Unhealthy development processes

Programming Language Deficiencies

Perhaps the most interesting origin of a code base's disease profile is rooted in the deficiencies of the programming languages used to build it. Here is a non-comprehensive list of such programming language deficiencies –

- Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
- Lack of or overly-complex immutability support
- Unclear or overly-imperative programming idioms
- Run-time inefficiency or difficulty to reason about thus
- Portability difficulty or non-portability
- Syntactic inflexibility
- Unclear / complex / ambiguous syntax

Language Deficiency Matrix

What follows is a matrix describing the occurrence and severity of deficiencies in commonly used programming languages in the context of the large programs this paper is concerned with (enterprise software, large web applications, frameworks / game engines, and commercial games) –

	Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts	Lack of or overly-complex immutability support	Unclear or overly-imperative programming idioms	Run-time inefficiency or difficulty to reason about thus	Portability difficulty or non-portability	Syntactic inflexibility	Unclear / complex / ambiguous syntax
C	*****	*****	*****		*	*****	*
C++	****	****	*****		**	****	****
D	***	***	****	*	***	**	**
Rust	*	***	***	*	***	*	**
Lisp	*****	***	***	***	****		***
Scheme	****	**	*	***	***		**
Clojure	****	**	*	***	*		**
C#	***	***	***	**	*	****	**
Java	****	*****	****	***	*	*****	*
Scala	***	**	****	***	*	*	****
Haskell	***		**	***	***		***
ML	**			*	***	*	**
F#	**		**	**	*	**	*
Idris					***	***	

** count := severity of deficiency*

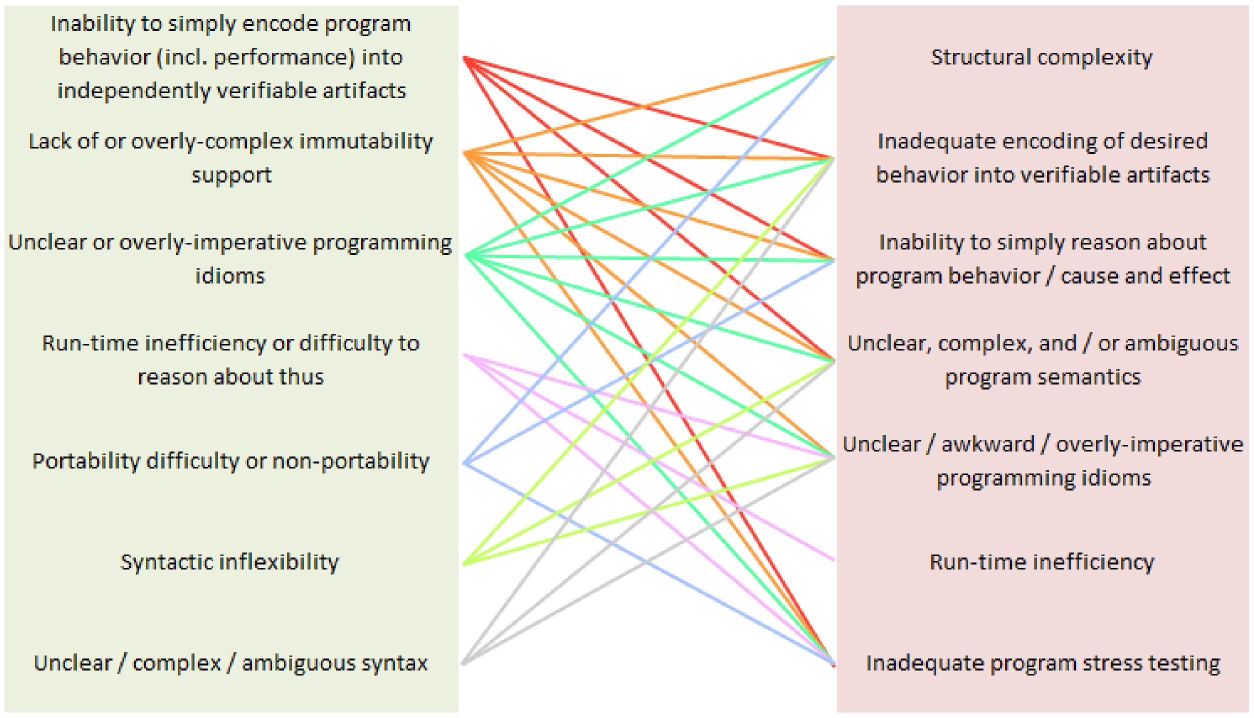
** Please accept my apologies for the subjective nature of many of these ratings. Please take them with a grain of salt for now as I do not know of any prior work that I can reference that captures a more objective view.*

Here we see large variances in deficiencies in commonly used languages, some of which can be show-stoppers in practice in many large commercial programs (to avoid the distraction of controversy, I will not elaborate my opinion here).

** Apologies to Lisp advocates of all stripes, but a lack of a static type system guts one's ease in simply encoding program behavior into independently verifiable artifacts. Sure, a billion unit tests and / or type annotations could be used instead, but in the author's view, most commercial software teams will find both approaches insufficiently effective and generally impractical.*

Mapping from Language Inadequacy to Code Disease

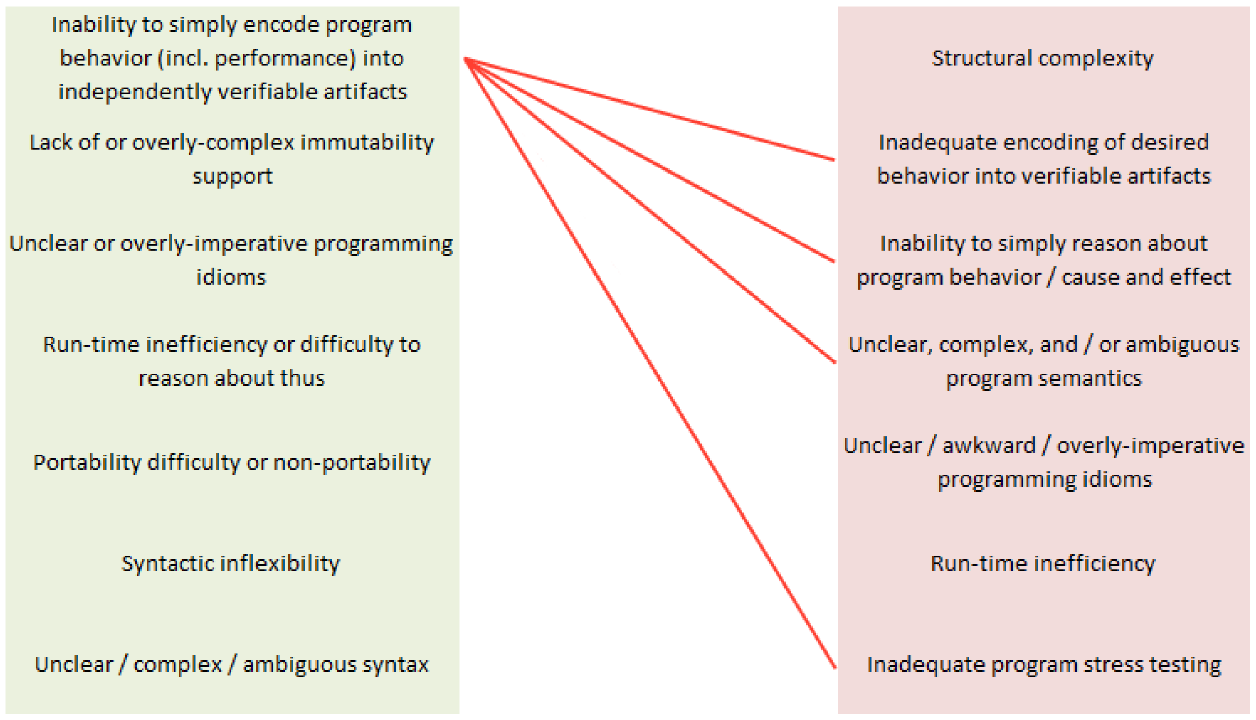
It is natural to see how a programming language's deficiencies map to the resulting program's code disease profile. In detail, these programming language deficiencies tend to lead to the following code diseases –


Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

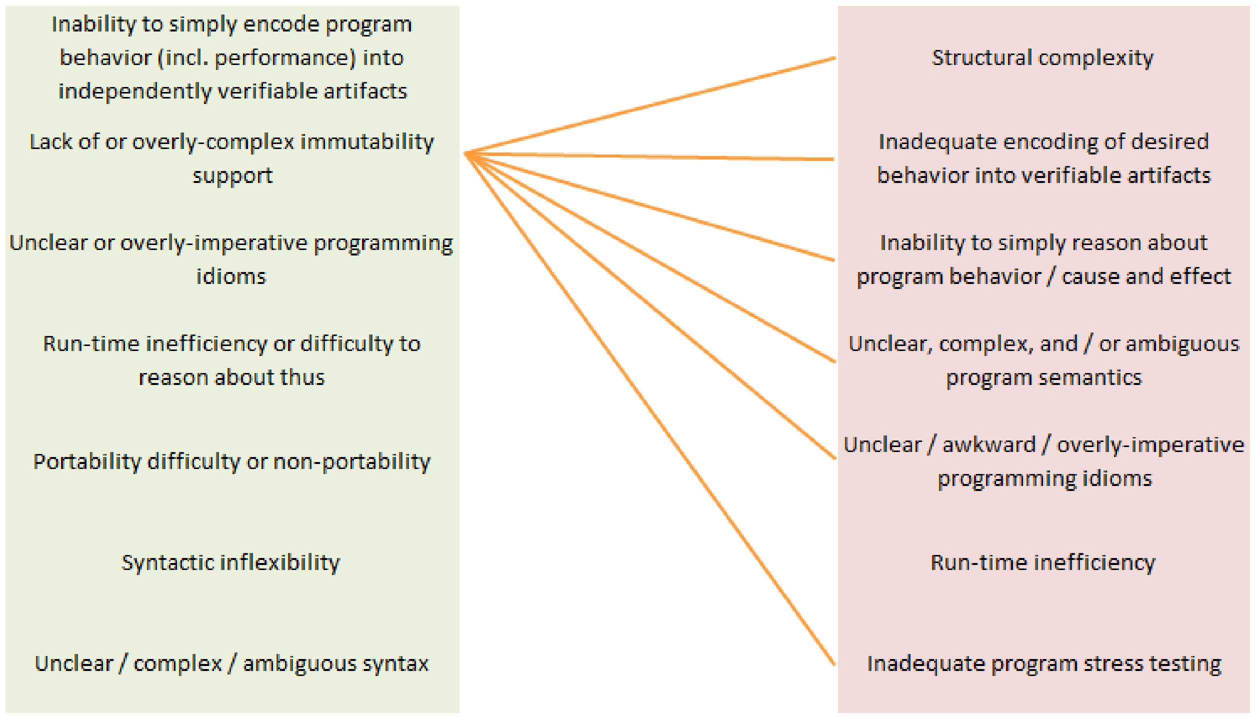
Of course, this is information overload, so what follows is a breakdown for each inadequacy category.

Diseases Originating from Inadequate Program Behavior Encoding

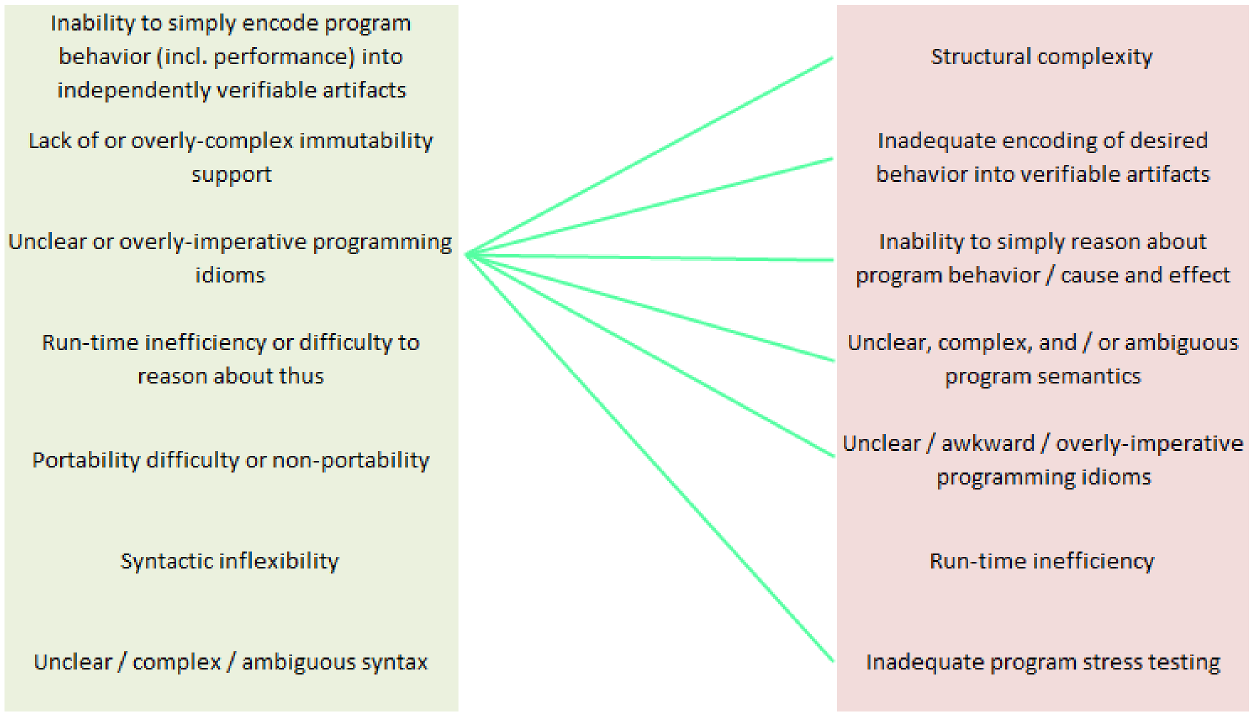
TODO: expand on each of these diagrams…


Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

Diseases Originating from Inadequate Immutability Support

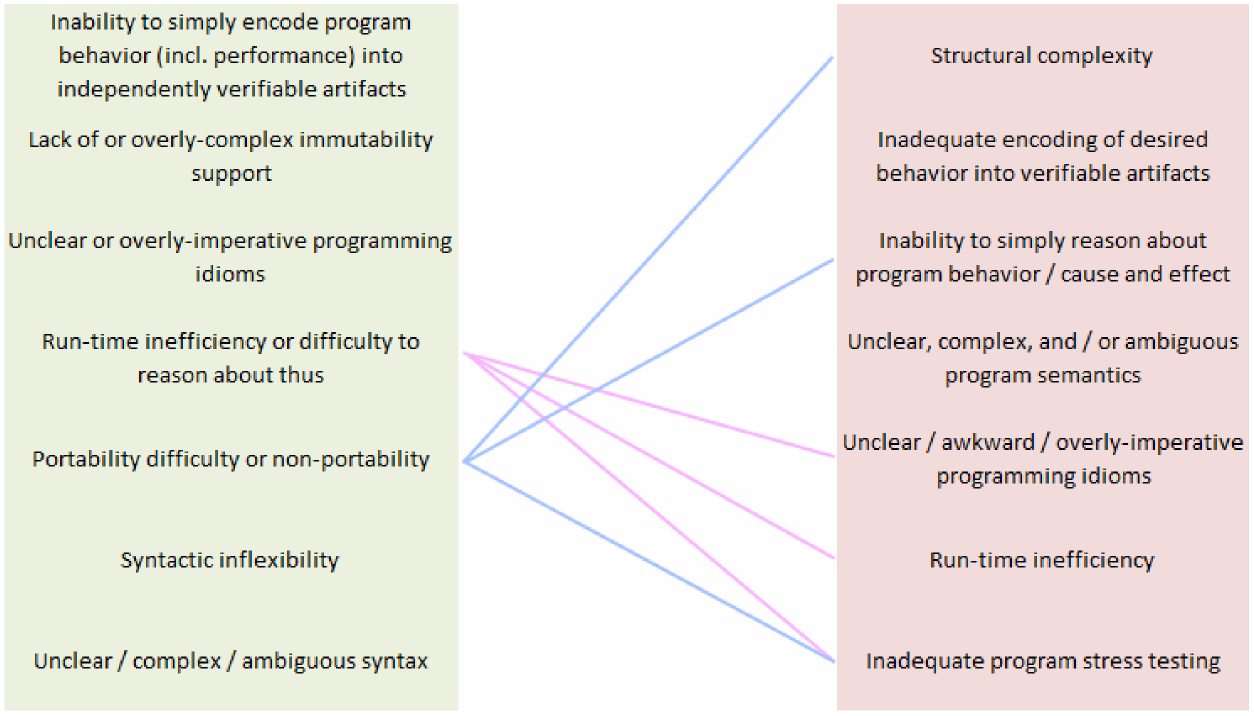

Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

Diseases Originating from Inadequate or Overly-Imperative Idiom Support



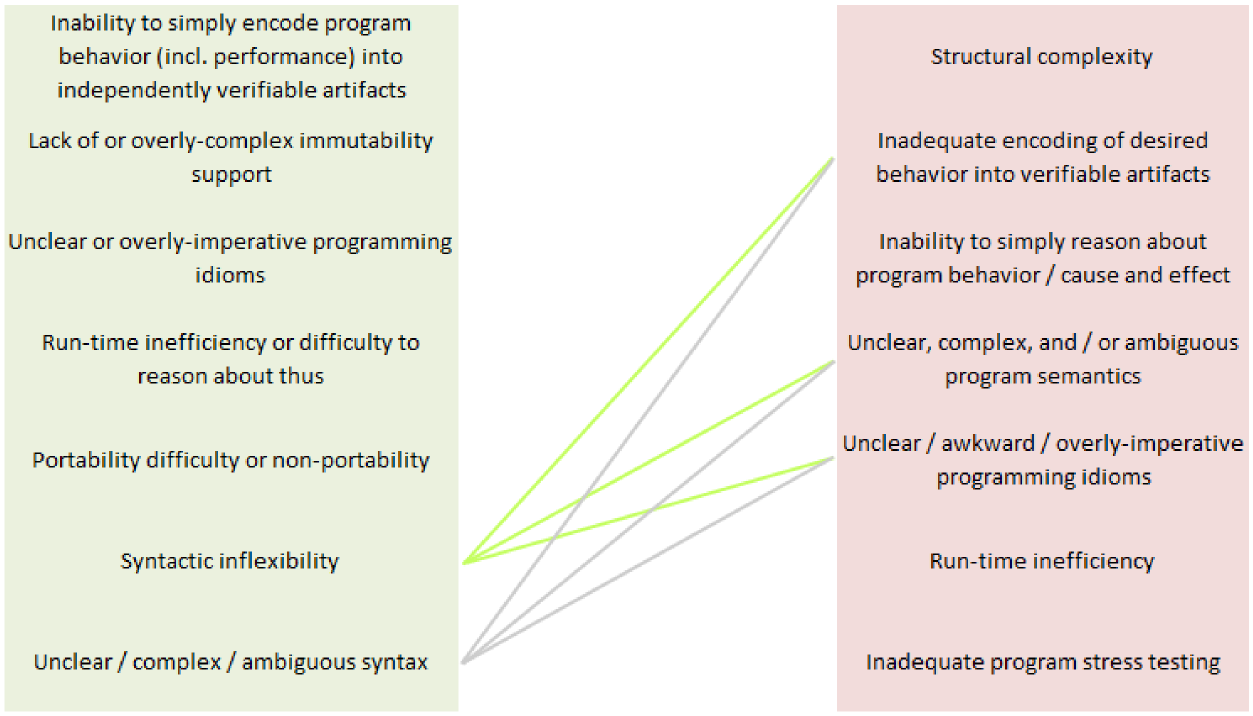
Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

Diseases Originating from Inadequate Run-time Efficiency or Reasoning about Thus



Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

Diseases Originating from Inadequate Syntactic Flexibility, Simplicity, and Clarity


Inability to simply encode program behavior (incl. performance) into independently verifiable artifacts
Lack of or overly-complex immutability support
Unclear or overly-imperative programming idioms
Run-time inefficiency or difficulty to reason about thus
Portability difficulty or non-portability
Syntactic inflexibility
Unclear / complex / ambiguous syntax
Structural complexity
Inadequate encoding of desired behavior into verifiable artifacts
Inability to simply reason about program behavior / cause and effect
Unclear, complex, and / or ambiguous program semantics
Unclear / awkward / overly-imperative programming idioms
Run-time inefficiency
Inadequate program stress testing

Unsound Thinking

There are several common forms of unsound thinking in software development that are at the root of code diseases -

Disease Denialism - To dismiss the notion or consequences of code disease (such as described in this document) is unsound thinking, and all too common of programmers and managers. Delusion too often becomes acceptable replacement for necessary intellectual fortitude. The typical consequence of this denialism is a consistently severe disease profile affecting all projects across an organization.

Disease 'Waitism' - To think that one should defer taking action against code disease in a post hoc manner is, in the author's experience, dangerously unsound. At best, attempts to correct even partially advanced code diseases seem to cost roughly an order of magnitude more than preventing it. In the author's experience with commercial code bases (not to mention his own), a stitch in time really does prove to save nine. In the context of the modern commercial development environments that the author has worked in, the most severe and common outcome of 'waitism' has been practical incurability; there's never seems to be the time to sufficiently treat most diseases once they become pervasive.

Disease Complacency – Failing to leverage your developers effectively is a common aspect of what the author would call 'disease complacency'. Any developer to be put into a position of making the most critical technical decisions must at understand the problems outlined in this document. To that, his political talent with clients is secondary. If a client has certain therapeutic needs (as they almost all do), complement that technical developer with another more politically-focused developer.

Unhealthy Development Processes

Very, very much has been said about unhealthy software development processes, and many, many solutions have been proposed. On the subjects of Cowboy Programming, Waterfall Programming, Scrum, XP, and Agile, more than enough has been said in the respective literature.

The only thing worth adding here is that even an impeccably healthy development process has only so much utility. While helpful, development process will not prevent or correct most of the diseases found in code. Too many organizations, perhaps led by unintentionally myopic software process consultants, think that this approach will do as much. It cannot.

A healthy development process is a necessary but *not* sufficient solution to a majority of code diseases and software development ills generally.

The Bottom Line

A healthy and useful code base helps your company for the length of its lifespan. It is an asset to your company, and will directly add substantial value to your company and its services for its lifetime.

A diseased code base will actively harm your business starting immediately by ramping up development costs, and throughout its lifespan by lowering the efficacy by which your company delivers value to clients both internal and external. It will be a constant liability, source of risk, and general resource drain on your company. The best case scenario for a sufficiently diseased code base is a lifetime of constant and expensive clean up procedures that, at best, distract from company goals and initiatives. The worst case scenario is the code base itself turning into much like a cancer that requires organizational ‘chemotherapy’ to completely remove before it begins compromising ever more vital business foundations.