LIVATEK

# How did we get stuck here: the origins of technical debt

*Technical debt is the cost of unfinished work in software. In practice, we see technical debt as a rise in the "cost of change". That we need to spend more and more time, money, brains to maintain and improve our code. It comes from not having enough time and resources to do the "thing right" while trying busily to do the "right thing" – getting products and features to the market.*

Technical debt can be incurred deliberately or unknowingly. The deliberate technical debt is a conscious way of dealing with the gap between what we need to achieve now and the way we should be doing it. It can and should be documented – so to allow improvement later when more resources are available. And, it should be done at the micro level in the task management tool employed. Documenting your short cuts allows you to account for the technical debt and in the future make deliberate decisions about cleaning up.

Alas, most technical debt is not incurred deliberately. To avoid it, you should be aware of its roots.

Two main problems increase directly the effort needed and cost of change:

1. Difficult-to-read code

It is hard to understand and obviously difficult to extend. Imprecise naming of variables and unclear formatting may easily occur when you are trying to do something under pressure. This makes it difficult to return to improve and expand the code later. Code conventions, pair programming and code reviews are tools to help avoid this type of technical debt.

2. Non-extensible code

Code that is difficult to read, is difficult to extend. But it may also be that the underlying logic is too simple or even faulty. Algorithms employed can be difficult to understand, be only partly implemented or the developer has solved a specific case rather than prepared for a general solution to future problems. Developers should be encouraged to use design patterns (commonly accepted solutions to solving standard problems). Also, the anti-dote is similar to that of the

difficult-to-read code - to have more than one set of eyes on the code with pair programming and code reviews.

Whereas the previous issues concern the direct cost of writing new code, two other types of technical debt have broader, in-direct implications:

### 1. Architectural short-comings

Not even the most diligent developer can compensate for an overall architecture that is incomplete or not thought through. Modular design and clear interfaces are important for stability and productivity of development – especially maintenance and extension. In the beginning, it is much easier to hold in your mind the idiosyncrasies of the platform. Later, this will be forgotten and new people will be onboard. Then, lack of modularity and clear data flows can lead to unpredictable results when code is changed – causing structural software risk.

### 2. Lack of testing

A structured approach to quality is important for stability of the code. Also, an advantage of good modular design is that functionality can be tested internally at low level where the code can be tested to live up to also unusual situations. When systems are tested as a whole, it is difficult to test individual modules to their limits. Without good unit test coverage, it becomes much more difficult to extend with new functionality and debug when new code influences the old. Test driven development is a process approach to ensure code quality.