The common architecture that self-adaptive Systems of Systems share involves two or more systems, each one including a managing system / controller and a managed system / target. The controller monitors the behavior of the target, and the target adapts to the directives of the controller, in order to achieve specific domain functionalities.

There are three basic architectures to achieve interaction between systems, featuring feedback loops in the form of runtime data:

1. Local Adaptation

There are local feedback loops that do not coordinate directly, only between pairs of managed systems. Feedback loops share no information with each other, so there is a uncertainty about other systems and the environment: indirect interaction may trigger side-effects / emergent events between individually optimised systems of systems.

For example, in a traffic lights managing system, each installation in an intersection may have an independent control system that check if all three lights are working. If it detects a faulty light, it may trigger an alarm signal while waiting for the traffic light to be repaired, for instance by activating an emergency blinking yellow light.

1. Regional Monitoring - Local Adaptation

In addition to the local adaptation architecture, controllers receive feedbacks from neighboring managed systems to support decision making of adaptations, which also reduce potential side effects of indirect feedback architectures and increases dependency. For instance, a traffic monitoring system including cameras distributed along the road may allow information sharing between multiple cameras, in order to detect traffic jams and providing information to clients.

1. Collaborative adaptation

Both controller and managed systems send feedback loops between each other and local loops may cooperate with one another.

For instance, considering independent groups of GPS devices that interact with a server, with each group consisting of a master and multiple slave devices, there may be two feedback loops: the first local loop deals with the activation / deactivation of the GPS service; the second loop acts in a group context and allows the master device to collect data from the slaves and adapt the group in case one GPS service fails.

**Code debt**. “Problems found in the source code that can affect negatively the legibility of the code making it more difficult to be maintained”

* Lower internal quality (unjustified issues): low readability, misuse of programming constructs, unnecessary code complexity…
  + Read input, write splits, read splits again
  + Throw wrong exception
  + Multiple nested if-else blocks
* Workaround (justified low quality code): compromise between code quality and specific requirements
  + Necessary “ugly” code for a specific unavailable function, until something better is available
  + Hack for backward compatibility

**Design debt.** “Debt that can be discovered by analysing the source code by identifying the use of practices which violated the principles of good OO-design”

* Code smells: violation of OO design
  + Feature envy: a method to be moved to a different class
  + Code clones: duplicate code
  + Lexical bad smells: poor lexicon that can lead to poor comprehensibility / increase soft fault proneness
  + Long method: method containing too many lines of code
* Design patterns: need for introducing a design pattern

**Documentation debt:** “Problems found in software documentation that can identified by looking for missing, inadequate or incomplete documentation”

* Inconsistent comments: misleading information, already addressed technical debt, still pointed to an issue that has been already closed and classified as “won’t fix”
* Licensing

**Defect debt:** “Known defect that should be fixed, but due to competing priorities and limited resources have to be deferred to a later time”

* Defects: known issues to be solved. Temporary patches implemented while waiting official fix
* Low external quality: problems that might result in a defect
  + Missing thrown exception
  + Missing input parameter control

**Test debt.** “Issues which can affect the quality of testing activities”

* Impossibility to reproduce bug
* Failing assert statement to check
* Low quality code in test suites

**Requirement debt**. “Tradeoffs made with respect to what requirements the development team need to implement or how to implement them”

* Functional: improvement to features needed, new features to be implemented
  + Comments reporting implementations going against the requirement specs
  + Comments referring to missing features
  + Comments reporting doubts about the implementation of requirements
* Non functional: performance issues

Deriving appropriate threshold values is a challenging open problem that has attracted the attention and effort of several researchers [20], [21], [22]. As a matter of fact, the approaches based on source code analysis suffer from high false positive rates [23] (i.e., they flag a large number of source code elements as problematic, while they are not perceived as such by the developers), because they rely only on the structure of the source code to detect code smells without taking into account the developers’ feedback, the project domain, and the context in which the code smells are detected.

However, relying solely on the developers’ comments to recover technical debt is not adequate, because developers might be unaware of the presence of some code smells in their project, or might not be very familiar with good design and coding practices (i.e., inadvertent debt). As a result, the detection of technical debt through source code comments can be only used as a complementary approach to existing code smell detectors based on source code analysis.

Despite the advantages of recovering technical debt from source code comments, the research in self-admitted technical debt, thus far, heavily relies on the manual inspection of code comments. The current-state-of-the art approach [14] uses 62 comment patterns (i.e., words and phrases) derived after the manual examination of more than 100K comments.

We create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to avoid removing comments that contain one of the predefined task annotations (i.e., “TODO:”, “FIXME:”, or “XXX:”)