Universidad de los Andes

Hoja de Trabajo – Facilidad de Pruebas ISIS - 2503

Objetivo:	Utilizar este caso de estudio para entender las principales decisiones de diseño que afectan la facilidad de pruebas

Tomado de: http://baruzzo.wordpress.com/category/design-for-testability/ How testable is a software architecture?

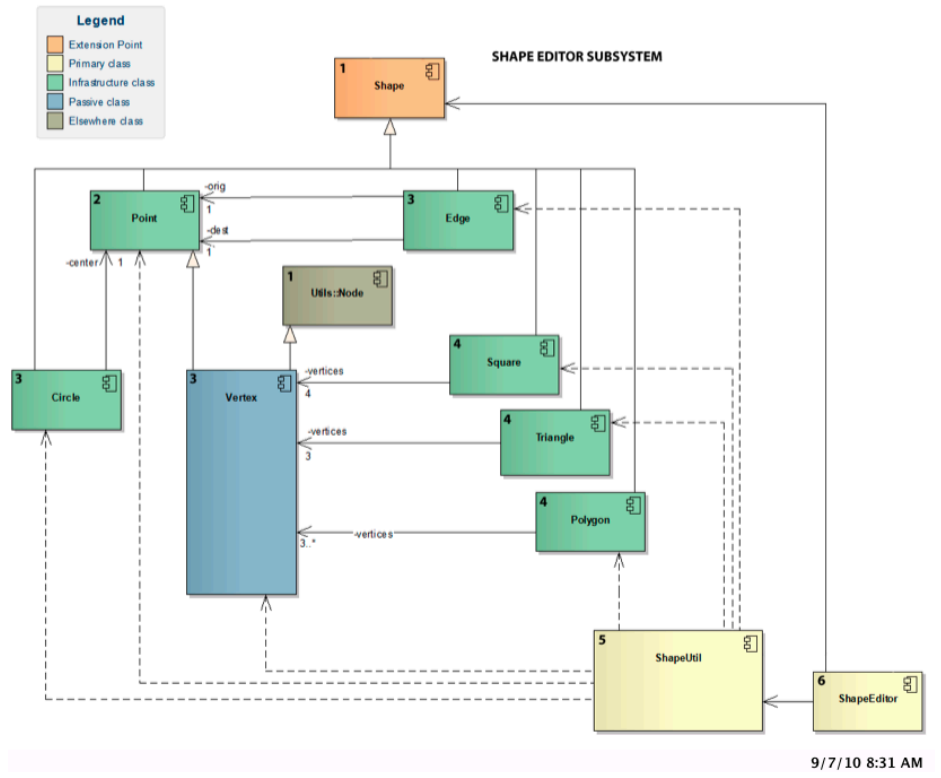
Testability is not testing. A (software) system is testable when there is an effective test strategy that can be used to verify the conformance of a particular implementation with respect to its specifications. Vice-versa, a system is tested when a specific test suite has been executed, verifying that such implementation really conforms to its specifications. In other words, testability is a property of the design, whereas "tested" is a state the system must attain before we release it to the customers.

The initial question (how testable is a software architecture?) is thus a question concerning the quality of the design. Objectively, it is a complex question, which can be tackled from several different perspectives. There is no single "good answer", no magical tool. I try to address it considering the quality aspect from the physical design perspective, which I have discussed in previous posts. This is also a good occasion to talk about two types of testing strategies:

- Isolation testing;
- Hierarchical testing.

One technique useful to analyze the testability of a system is to check either if it is layered or how far it is from a layered structure. In the case it is layered, we want to analyze also both the complexity of each layer (how many components, how many dependencies, etc.) and the cumulative complexity of the entire system. Layering is a technique to produce levelizable systems. The idea is to identify a method for partitioning all the components constituting the system under testing into equivalence classes called levels. This partition is based on the physical dependencies existing between the components. Each level is then referred by a non-negative integer number, called level number. An example will clarify the idea.

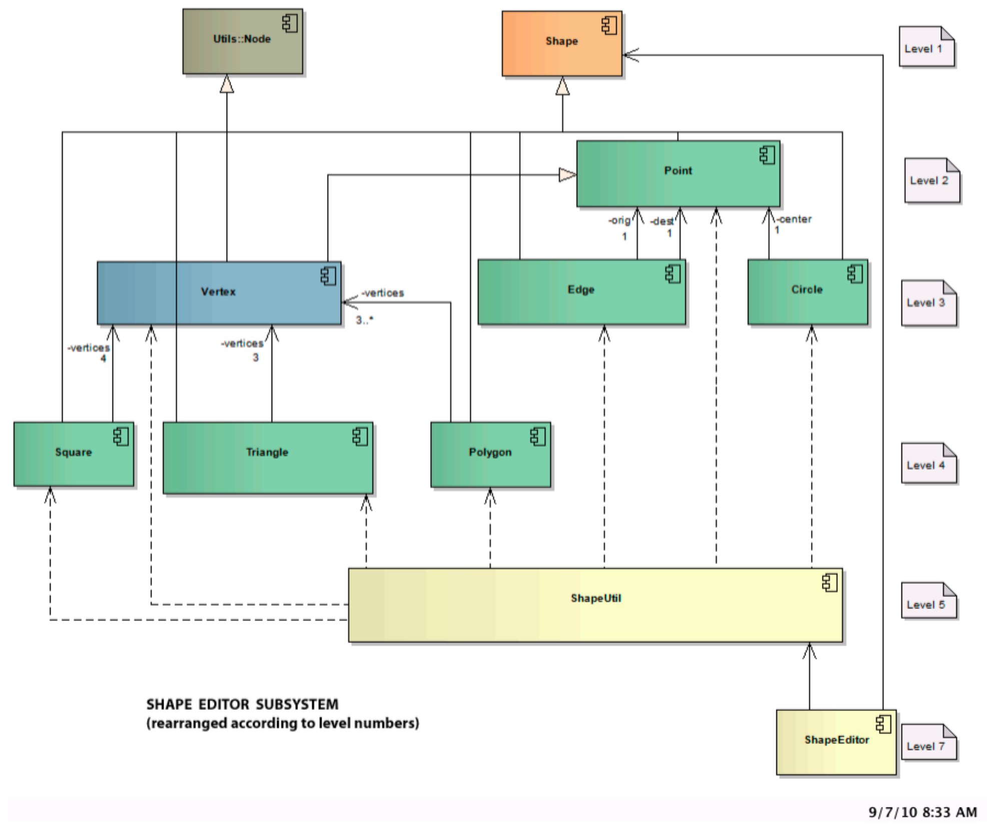
Several years ago I developed a geometrical engine to solve some research problems in the field of computational geometry. This engine was intended to provide not only the geometrical algorithms, but also a predefined library of shapes upon which the programmers could build their applications. I was asked also to develop a simple shape editor implementing the most common operations concerning the shapes manipulation. Inspired by the work of Lakos, I designed the shape editor subsystem as illustrated in Figure 1 (actually, this is a simplified version of that system, but it suffices to illustrate the point). It was composed by several components, which do not form any cycle in the physical dependency graph. I strove to not introducing cyclic dependencies because only a configuration of components that has a directed acyclic physical dependency graph



(DAG) forms a levelizable system. In a DAG, we can always assign a unique level number to each component using the following rule. At level 0 we place all the components external to the system under test (e.g. a third-party library component, assuming that such components have already been tested). At level 1 we place components internal to the system under test with no local physical dependencies (e.g. Shape and Utils::Node components in the figure). Finally, at level N we place those components internal to the system under test which depend physically on components at level N-1, but not higher. In this way, a component has a level number one more the maximum level of the components upon which it depends.

The rationale in this level number assignment is that a level number counts the length of the longest path from that component through the (local) component dependency graph to the (possibly empty) set of external or compiler-supplied library components. In general, we will say that any software system is levelizable if its physical dependency graph can be assigned unique level numbers. A cycle is not levelizable, so we cannot assign to its components unique level numbers. In that case, the cycle must be considered as a unique component in order to assign a unique level number to it.

It is very simple to rearrange the components in the diagram in order to emphasize each level. The diagram in Figure 2 reflects this new disposition, appearing now as a typical layered architecture.



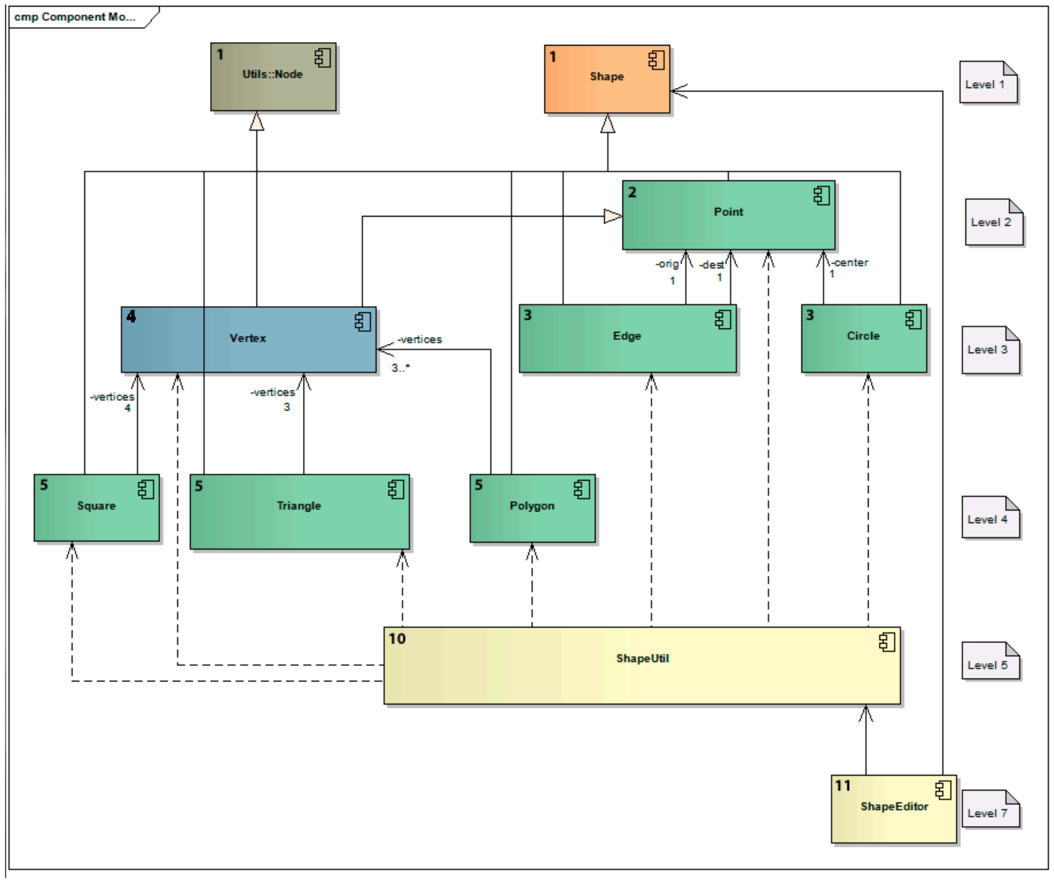
Why layering is a desirable property of a design? Why I was so carefully with cycles in the dependency graph? If we have a levelizable system and we want to test it, we can immediately identify which components are testable in isolation: it is sufficient to look at those components belonging to level 1. In a cycle there is no level 1! In the case of my shape editor illustrated in Figure 1, there are only two components that can be tested without linking any other component: Utils::Node and Shape. Moreover, by starting to test at the lowest level (level 1) and testing all the components belonging to that level before moving to the next level, we guarantee that all the components on which the current component depends have already been tested. This criterion is referred by Lakos as hierarchical testing, because level numbers define a partition that can be considered as a hierarchy of component dependencies in the physical graph.

Layering let us quickly identify the most reusable components in the system and, if we arrange the UML diagrams as in figure 2, we will find such components looking at the top of the diagrams. However layering alone does not reveal a direct measure of the internal complexity of each level, nor it provides a measure of the cumulative complexity of the overall architecture. To do this, we need a metric based on the physical properties of a system: the Cumulative Component Dependency (CCD). CCD is not new: it has been discussed extensively by John Lakos in his book "Large-Scale C++ Software Design", published in 1993. Despite this metric is available long before the rise of UML, no modeling CASE tool supports it nowadays (at least to the best of my knowledge). CCD is useful because represents a good indication of the long-term maintenance cost associated to that component/system [1]. CCD is defined as follows:

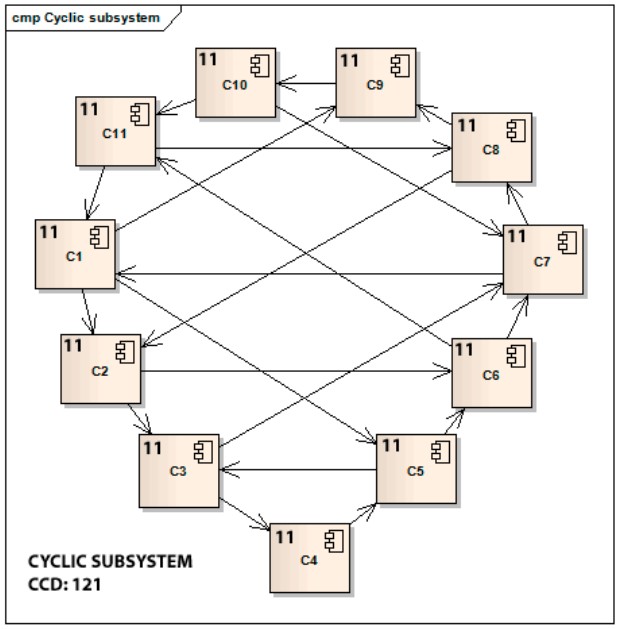
CCD = sum over all components C in a subsystem of the number of components needed in order to test each C incrementally

Calculate the CCD is straightforward: for each component C in the subsystem under test we count how many other components are reachable inspecting the dependency graph rooted at C. To test C incrementally we have to write a test driver for it and a single independent test driver for each component reachable from C. If there are n components reachable from C, we have to link k+1 components to test C incrementally. This k+1 is the link cost associated to C. Now we add the link cost of all the components in the subsystem under test. The total sum is the CCD value for the specific subsystem. In our example, the CCD of the *EditorShape* subsystem is 50, as shown in Figure 3. For each component in the upper-right corner I have shown now the link cost. To test incrementally the Edge component, we have to link to its test driver the test drivers of Point and Shape, so the link cost is 3. If we want to test Triangle, we have to link also Vertex, *Utils::Node*, Point, and Shape, so the link costs is 5. The CCD is the sum of all link costs.

CCD is useful because provides a numerical measure of the overall link cost associated with the incremental testing of a given system. Obviously a low CCD value identifies a design, which is simpler to test, to maintain, and to understand. A low CCD combined with a levelizable design allows to study pieces of the subsystem in isolation, to test, tune, and even replace them without having to involve the entire subsystem either mentally or physically [1].



If our subsystem is very tightly coupled, e.g. forming a cyclical graph of the same size (11 components) as illustrated in Figure 4, the CCD will be 121. Comparing the 49 CCD value of the system in Figure 3 with the CCD value of this second configuration, we have numerical evidence that the former design is better (from the testing perspective). Indeed, in Figure 4 none of the components can be tested in isolation or reused independently of the others. There is no partition that let us to test the subsystem hierarchically. Each independent test driver will be forced to link the entire subsystem in order to test a single component, causing a quadratic link time.



The conclusion of this long post is summarized by the following principle: Large software architectures must be levelizable if they are to be tested effectively.

As Lakos remarks:

CCD provides a raw measure to grasp the structural complexity of a software architecture in order to test it. Level numbers characterize the relative complexity of a component, providing also an objective strategy for testing (hierarchical testing). Cyclic physical dependencies among components inhibit understanding, reuse, and testing, hence they should be avoided or at least confined in a single package. If we design our software systems with an eye toward minimizing CCD, most cyclic dependencies would be eliminated, producing more testable architectures. Moreover, reducing the interdependencies between components as quantified by the CCD helps to keep the overall system simpler, improving both maintenability and understandability.

Bibliography

[1] Lakos, J. – “Large-Scale C++ Software Design”, Addison Wesley, 1996