design of DAG framework (Figure 8) aimed at capturing the concepts of a directed acyclic graph, a commonly used visual presentation structure for decision models. Likewise, Tree (Figure 9) was designed to represent a rooted directed acyclic graph, modeled as a specialization of DAG. For clarity's sake, it is not depicted in Figure 9 the specialization relationships between DAG and Tree classes (i.e. Tree Node is a Node, Tree Edge is an Edge, Tree is a DAG).

![Class Diagram of the DAG Framework](image)

**Figure 8 - Class Diagram of the DAG Framework**

![Specialization of the DAG framework to the Tree framework](image)

**Figure 9 - Specialization of the DAG framework to the Tree framework**

Actually, Tree differs from DAG essentially on the maintenance of the constraints that characterize each type of graph structure (e.g. it has a root, a root cannot be the sink node of any edge). In this sense, Tree acts as a normal DAG, and includes special constraints like connected and root element that Tree instances must conform to. Additionally, if we consider different types of graphs, they basically differ in this type of structural constraints (e.g. connected, directed, rooted, etc). So, we can say that one of the variable aspects in a GRAPH framework is the semantics constraints involved a specific type of graph structure.

Using the Strategy design pattern (Figure 1), we have treated each of these constraints as distinct constraint verification algorithms that iterate over a graph object structure to validate it. The redesigned framework, named Graph, is shown in Figure 10. Though not shown in the picture, it is composed of Nodes and Edges, just as DAG in Figure 8. The class GraphConstraint represents the common interface that any graph constraint must conform to. Specific algorithms are represented as subclasses of GraphConstraint. Another interesting point is that now one may add and remove constraints even at runtime using AssignConstraint, thus changing dynamically the type of visual graph structure admitted by the application.

Another point is that some constraints can be further decomposed into smaller constraints, many of them shared by different types of graph. For instance, a DAG is a directed, connected and acyclic graph. A Tree is also acyclic and connected. Besides, it is important to the treat composed and individual constraints in the same way. These are some of the problems that the Composite
design pattern deals with [GAM94]. In Figure 10, Algorithm is an abstract class, and its subclasses (e.g. IsDirected, IsConnected) provide the specific code for the operation IsValid. CompositeConstraint represents algorithms defined in terms of other algorithms (e.g. DAG, Tree), which can be in turn elementary or composed. A CompositeConstraint has a template implementation for IsValid: it returns true if all constraints included as component are in turn valid. Specific composite constraints can be created either at runtime (instantiation of CompositeConstraint) or compile-time, by the creation of a subclass (e.g. DAG).

In this new design, a Graph object delegates to its component object (a GraphConstraint subclass instance) the structural constraint verification responsibility. The polymorphic operation IsValid is used to verify if a graph is valid or not, i.e. whether it obeys the constraints imposed by the constraint component. The design is black-box in the sense that by composing Graph and/or CompositeConstraint objects differently, entirely distinct applications can be developed, and this configuration can be even changed at runtime. The important point is the definition of standard interface (IsValid) to which any GraphConstraint subclass must conform to.

The design of this new framework presents many advantages over the original ones. First, the variable component (constraints over a graph structure) was isolated from the more stable ones (graph composed of nodes and edges), allowing them to evolve independently. The process of customization in order to represent a special kind of graph becomes clearly easier. This is due to the black-box approach, that requires from developers knowledge only about the existence of constraint algorithms and of the standard interface for connection.

This example shows how an analysis and white-box framework evolved into a design and black-box one. We are currently using other patterns to redesign other frameworks, as well as the definition of a standard communication protocol to enable easy connection between objects of