Black-box frameworks are more dynamic, because one can change the connected components at runtime. In white-box frameworks, the adaptation is made at compile time. In contrast, black-box frameworks are more difficult to develop, since one has to design the interfaces to pluggable components very carefully, foreseeing the future uses of the framework. Interfaces not well designed can limit future reuse opportunities for a framework. At the present moment, because of these advantages of black-box frameworks over white-box, several frameworks experts favor the use of black-box over white-box (references).

After these considerations, it is clear that framework development is a very hard task. To create flexible and truly reusable frameworks, developers have to apply good design techniques [RIE96][JOH88], considering very carefully the stable and variable aspects of possible applications, and, as pointed in [GAM94], "design for change". Experienced framework designers have found good solutions for common problems in framework development. Transferring this knowledge to beginners designer will help them to avoid these same problems in their tasks.

The concept of design patterns [GAM94][PRE95][COP95][VL196][MAR98] were introduced in OO to record and disseminate successful solutions to common problems in framework design. This concept was imported from Architecture field, where [ALE77] has presented the concept of patterns, intended to record experience in building and construction. A pattern is identified by a unique name, and describes a solution to a recurring problem inside a context.

Design patterns offer a structured way to encapsulate knowledge and experience of expert software designers. [GAM94] presents a catalog of 23 design patterns addressing problems of framework design, almost independent of application domain. Most design patterns show how to solve reusability problems, particularly with the use of black-box techniques.

The problems considered by design patterns always concern how to isolate the aspects that can vary in a framework, i.e. how to implement the frozen and hot-spots, smoothing framework evolution. For instance, the Strategy design pattern [GAM94] shows how to deal with the problem of supporting various algorithms or strategies to execute some functionality, considering for instance the possibility of changing the algorithm at runtime, and also the trade-off between performance and quality. The solution described by Strategy suggests to isolate the different algorithms into a separated classes hierarchy, represented by an abstract class that defines the standard interface. The client of the algorithm interacts with it through this standard interface. Figure 1 shows the generic participants, responsibilities and collaborations of the solution suggested by Strategy pattern.

The generic concept of pattern has been used in the last years in the software engineering field to record and transfer software engineers experience, covering all stages of software construction. Particularly, pattern languages [ALE77][MES98] have been used to software in general and framework documentation. A pattern language is a set of patterns that work together to solve a complex problem, providing a default order in applying the patterns. It is an excellent way to write "how-to-like" guides.
3. OO Architecture to DSS Development

As it was mentioned, the OO Architecture proposed by [BEC93] contributes to the DSS development problem by addressing both developers and users problems. The benefits for developers are rapidity and flexibility in DSS development, through the use of frameworks of different natures. Resulting applications offer modeling facilities to decision makers, encouraging them to use the DSS to formulate and solve their decision problems.

From a developer's point of view, the architecture can be defined as a guide to the development of DSS, showing how to organize, adapt and bind different frameworks, and describing also how to incorporate these modeling facilities to decision makers.

The target DSS to be developed using this guide present some specific characteristics that help decision makers interact with the DSS:

- **modeling paradigm**: decision makers are able to express their specific problems in a conceptual level, directly in terms of decision elements and independently of resolution and implementation concerns. The decision makers use concepts of their cognitive world, and model formulation becomes the simple activity of instantiating, customizing and combining these concepts [BEC95]. For example, the Capital Budgeting class of problems involves decisions about investments whose returns are expected to extend beyond a year. The lifetime of a project thus spans over several years, and it implies forecasting all incomes and expenditures incurred in this period. The profitability of an investment can be evaluated by a number of criteria, such as the Net Present Value (NPV), Internal Rate of Return, Payback Period, etc.. Models for this class of decision problems can be formulated in terms of concepts such as lifetime of the investment project, time-schedules of incomes and expenses, cash flow, profitability criteria, cost of capital, etc.

- **visual representation**: domain concepts are given a graphical representation, using familiar presentations structures, such as table, graph, report, chart, etc.. It is through this visual representation that users create and manipulate instances of model concepts. For example, a tabular form is the most frequent graphical representation for Capital Budgeting problems, with columns representing the concept of time, and rows representing various concepts like cash flow, profitability criteria, etc. (Figure 2). Additionally, the hierarchical relationship between the incomes and expenditures, may be represented by a tree structure (Figure 3).