Design Document: Business Planner

Wesley Murray, Lee Kendall

Purpose

The purpose of this program is to allow users to construct, store, and evaluate business plans. Business plans are frequently used by organizations to outline their goals, including Centre College. A program which enables for easy business plan writing could prove highly convenient for the organizations which utilize them. Since there are a wide variety of business plan templates, our design allows developers to easily add new types of templates for users to expand upon.

Key Terms

User Template - A clone of a developer template which the user manipulates to represent their business plan.
Developer Template - A template object created by developers that represents the basic structure of a given business plan.
Developer - A programmer that creates types of business plan templates.
User - Any person that uses this application to create a specific user template from a provided developer template.

Specifications

The program must be flexible enough to allow developers to add new business plan templates while having to modify existing code as little as possible. For now, the design should be able to store a VMOSA business plan, a Centre assessment business plan, and one other type of business plan. It must also be possible to serialize the state of the program into an XML file in such a manner that it can be read back from memory. To make this process simple, the design should be non-cyclical. At the same time, the design should allow for additional tools to be implemented in the future to improve user-business plan interactions. Moreover, the data structure should allow developers of template to set restriction on the form of the template. For example, for VMOSA template, users are allowed to have only 1 mission statement.

Design Overview

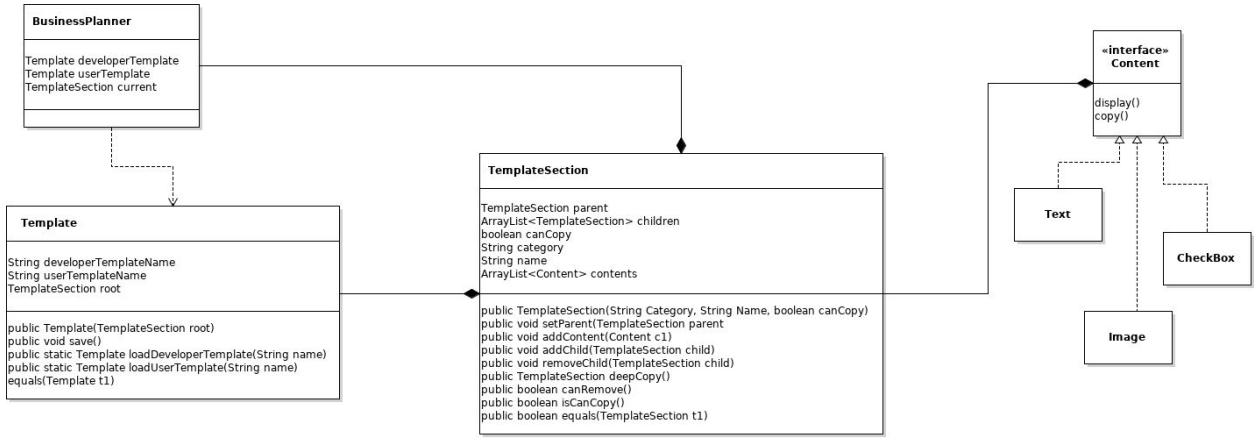

Figure 1 - Program UML

The BusinessPlanner class is the main object which users and developers interact with. It stores a developer Template object and a user Template object. Both objects are instantiations of the template class, but they serve different roles. The developer template is the developer-created business plan format and the user template is a clone of the developer template. The developer template specifies a business plan structure, and the user template is the template object the user can edit to represent their specific business plan. Each Template object has a string attribute signifying the developer template used and a name attribute representing the name of the user's specific business plan. The Template object also contains the root of a tree of TemplateSection objects. Template objects are instantiated by passing a TemplateSection object to its constructor. The class includes methods for copying, saving, and loading a template. TemplateSection objects correspond to a business plan section from any of the business plan types. Each TemplateSection object in the tree has pointers keeping track of its parent and any children. TemplateSections have a string indicating their title (such as "Objective 1" or "Vision"), the category that section falls into (such as "Objectives" or "Strategies"), and an array of Content objects. The Content object implementations represent any information TemplateSection objects might need to store. This could include text, checkboxes, images, and so on. The developer can add more Content implementations as needed, lending great flexibility to our design.

Design Defense

Why choose a tree structure over a graph or linked list?

- The tree structure was chosen because many business plan sections have a one to many relationship with other sections. A linked list cannot support this kind of relation. A graph would work but is not necessary. Moverover, a graph would overcomplicate traversing the relationships between sections and makes serializing a template difficult.

Why have a Content interface?

- At this point, we can't say exactly what information might be associated with each TemplateSection. We can expect there to be a text object signifying the actual subject matter of the section, but we don't know how exactly each section will be displayed and how we may want to manipulate it in the future. The Content interface provides a flexible design pattern, allowing developers to add any information they may need to manipulate the TemplateSection objects.

Why is Template a concrete class and not an abstract class or interface?

- No future change is expected in the template class. What we do expect to change is the content in each template section provided by developers.

Why not have a separate object for developer templates and user templates?

- The developer template is a blank set outlining the structure of a kind of business plan, which the user fills in as the user template. Having both types of templates represented in the same object reduces the code needed without sacrificing functionality.

Why is TemplateSection a concrete class not an abstract class or interface?

- This design uses a bridge pattern, encapsulating any Content that will compose a given user template section into a Content object. All potential future changes to a given TemplateSection are handled by the bridge pattern.

Template Development

Each business plan is divided into categories which can be further divided into sections. For example, "Objectives" is a business plan category. The Objectives category is composed of many individual objectives that are each considered a section. Business plan sections are represented by the TemplateSection class. Each TemplateSection object has a category, name, and a list of Content objects. Each section has the same structure. What changes from section to section is the list of Content objects. Some sections may require text, images, check boxes and so on. These items are represented by a Content interface with the understanding that a TemplateSection is given a Content object depending on what that section requires. The developer will either use existing Content objects or create their own to build a template section. For instance, in the future developers may want to keep track of multiple versions of business plans as they change from year to year. They may wish to

add a “date written” implementation for content, so business plan writers can know which version of a certain business plan section they’re looking at.

Most business plans have a hierarchical structure, with some sections depending on others. For example, under VMOSA objectives have multiple strategy sections associated with it. To handle this relationship, the TemplateSection function as tree nodes. Developers will link together the TemplateSection objects into a tree structure reflective of the type of business plan they wish to implement.

One constraint for the structure of any given business plan is to keeping in mind the possible number of TemplateSections one can have for each category. VMOSA is a design that allows for only one Vission and one Mission, therefore it should be impossible for the user to create more TemplateSections with these categories. To solve this, each TemplateSection object also has a boolean attribute indicating if the currently accessed section can be copied or not. Given that there can only be one Mission in VMOSA, the Mission TemplateSection object would have its “canCopy” boolean set to false. In contrast, there can be any number of Objectives, so its canCopy boolean is set to true.

To limit which sections users can remove, we rely on a canRemove method in the TemplateSection class. This method only returns true if the parent of the currently accessed section has at least one other child. For instance, canRemove would return false if the user tried to remove the only objective, as the mission section would only have a single child left. However, canRemove would return true if the user tried to remove an objective when there were three objectives present.

Developer Template Creation Example

Create Vision TemplateSection
Add Content objects to Vision
(These two steps are represented by new *TemplateSection* below)
Create VMOSA Template(new Vision)
Vision.addChild(new Mission)
Mission.addChild(new Objective)
Objective.addChild(new Strategy)
Strategy.addChild(new Action)

Figure 2 - Developer Template Creation Pseudocode

The above pseudocode expresses how a developer would start creating a VMOSA template. It has a Vision segment, Mission segment, Objectives segment,

Strategy segment, and Action Plan segment. The developer must first create a vision TemplateSection. After the appropriate Content is added to the vision object, it is passed to the constructor of Template to create a new Template objects. Then, the developer creates a mission TemplateSection as a child of vision, followed by an objective TemplateSection as a child of mission. This process will continue until an action object is created as a child of strategy. At this point, a full VMOSA template will have been created. The structure would look like what is shown below :

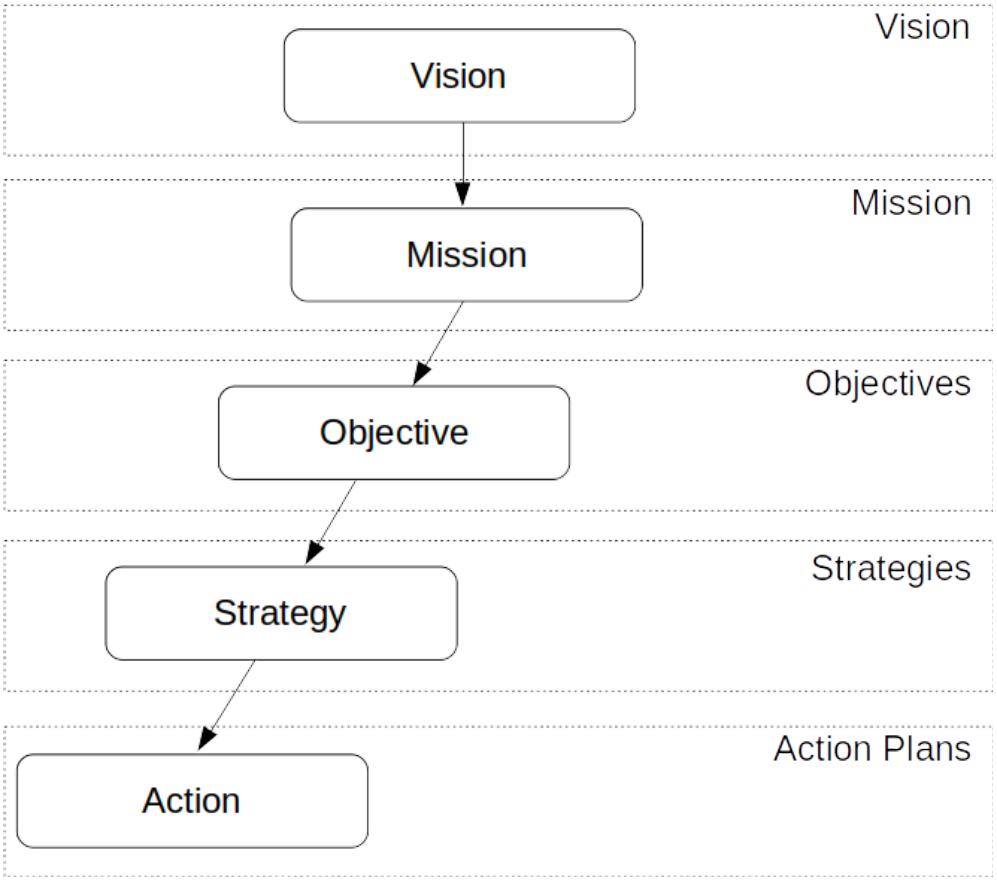

Figure 3 - Example Developer Template Tree Structure

User Template Creation

When a user chooses to use a certain template, the main program first loads the corresponding develepor Template from memory. BusinessPlanner then creates a copy of the template using depCopy(). This object is stored as the user template. At this point the user can interact with the user template. The user will be able to fill in the Content of each template section in the tree. They can add additional template sections as nodes in the tree as needed. For instance, if the user wants to add an objective, they'll use a method which clones the developer template tree from the objective node down to the action node. The new objective node will be added as a child of the mission node.

However, the user will not be able to add new template sections if the canCopy boolean of the currently accessed section is set to false. For instance, in VMOSA, vision's canCopy would be false. Users would be unable to add more visions. An example of how the structure of the template will change with the addition of a single objective is shown below:

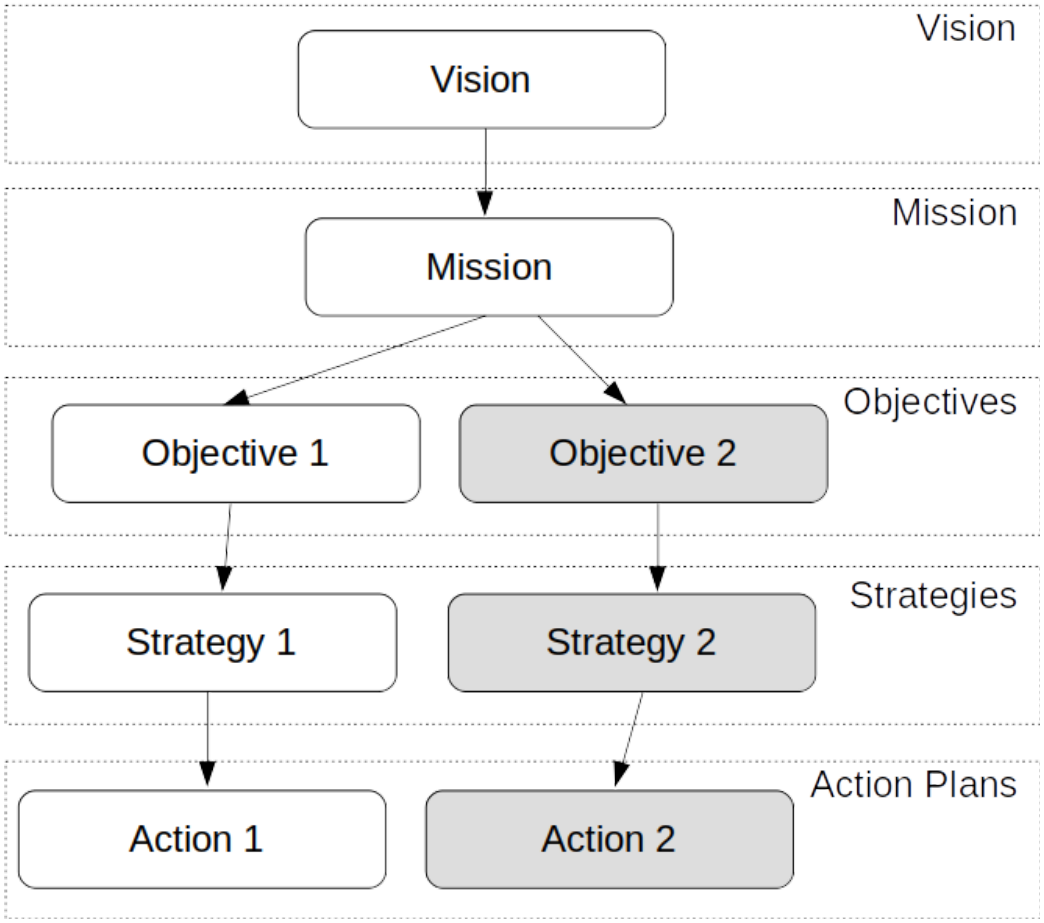

Figure 4 - User Template Example

Conclusion

Our design relies on manipulating two instantiations of the template class. The developer template is a blank layout created by a developer to specify the structure of a type of business plan, and the user template is a copy of the developer template which the user fills in. The template object comprises a tree of templateSection objects, which each contain content pertaining to what the developer wants each section to contain. Further implementations of the content interface can be implemented by the developer to add additional tools.

The next probable step in the development of this program would likely be to construct a graphical interface to make the process of using business plans practical to the average user.

Limitation-wise, It’s apparent that this design depends largely on a more strict hierarchical structure of the sections comprising each sort of business plan. So, in the future it may be of interest to devise a way to represent less conventional sorts of plans as well, such as those which are not strictly hierarchical or which may even contain cycles.