

area. A `CreateMethod()` function generates a series of byte code statements that create and populate the scene with graphical objects when executed. These statements are stored in an object called `VcCGMethod 410`.

[0134] `VcScene 400` contains a list `m_nodes` of objects derived from class `VcDrawingNode 402`. `VcDrawingNode 402` is an abstract base class for all shape and logic nodes which represent the contents of a scene. `VcDrawingNode 402` defines an interface, `GetExecStmts()`, for generating the list of `VcStmt` execution statements in derived classes. `VcDrawingNode 402` is subclassed by `VcShapeNode 404`. `VcShapeNode 404` is an abstract base class for nodes which generate shapes. Derived classes of `VcShapeNode 404` generate the byte code statements necessary to construct and set the properties of each type of shape.

[0135] `VcShapeNode 404` in turn is subclassed by the `VcDataNode` class `406`, which is a class of shape node which binds a data source to a graphical template representing the layout of objects for each row in a query. The byte code generated by the `VcDataNode` iterates over the result set supplied by the data source, creates a set of objects, and adds them to the active scene.

[0136] `VcDrawingNode 402` is also inherited by a class of drawing node `VcLogicNode 408`, which in turn is inherited by `VcDataSource 410`. `VcLogicNode 408` is a class of drawing node which generates byte code that defines relationships between shape nodes and the user. `VcDataSource 410` is a class of logic node representing an abstract source of data. Derived classes are responsible for generating byte code which results in a table of data used by the data node to generate the graphics in a layout.

[0137] `VcDataNode` class `406` contains an instance of `VcDataSource 410`. The `VcDataSource 410` is inherited by a class `VcQueryDataSource 412`. `VcQueryDataSource 412` is a class of data source responsible for interacting with a query object to generate the byte code which constructs a query instance, substitutes runtime parameters, and executes the query.

[0138] The `VcQueryDataSource 412` in turn points to a query object `m_query`, a member of `VcQuery 414` which stores information about an SQL query, including the names and types of columns and parameters. The `VcQuery` object `414` in turn has a set of columns stored in a `VcColumn` class `416` and a set of parameters stored in `VcParameter` class `417`. `VcColumn 416` stores information about a column of data that may be retrieved from the database so that it may be referenced by the context object used by the byte code execution statements. `VcParameter 417` represents a named placeholder object acting as an argument to a SQL SELECT statement. Before the SQL statement can be executed, the parameter is replaced with the runtime value of the parameter.

[0139] The `VcScene` object `400` generates an instance of the class `VcCGMethod 410` when its `CreateMethod()` member function is called. `VcCGMethod 410` in turn has a set of objects that belong to an abstract base class `VcStmt 420`. `VcStmt 420` is an abstract base class for all byte code statements. Its pure virtual method, `Perform()`, defines an execution interface that must be implemented by all derived classes. The `VcStmt` abstract base class `420` is inherited by a number of byte code statements including `422`, `424`, `426`,

`428` and `430`. `VcStmtCreateShape 422` is a byte code statement which creates an instance of a particular shape object when executed. The statement `422` records a tokenized name for the object for hash table lookup. `VcStmtCreateQuery 424` is a byte code statement which creates an instance of a particular query object when executed. `VcStmtBeginProperties 426` is a byte code statement which notifies a shape instance that its properties are about to be set. `VcStmtSetProperty 428` is abstract class for byte code statements which set the value of a shape's property when executed. Derived classes represent each data type stores the tokenized name of the shape and the set of property ID's necessary to uniquely address the property. Finally, `VcStmtEndProperties 430` is a byte code statement which notifies a shape instance that changes to its properties are complete and that it can perform any calculations which depend on more than one property.

[0140] The object model of the code execution process is similar to the object model shown in FIG. 4. Briefly, the `VcstmtCreateShape` byte code records a tokenized name for the object in a hash lookup table and creates an instance of a particular shaped object when executed. The `VcStmtCreatesShape` byte code statement contains a factory object derived from a `VcShapeFactory` abstract base class which is responsible for creating a shape within a scene. Additionally, the `VcStmtSetProperty` abstract class stores the tokenized name of the shape and the set of property ID's necessary to uniquely address the property. The `VcStmtSetStringProperty` byte code statement sets the value of a shape's property to a string value and contains a pointer to an abstract string function for evaluating the property value before setting the shape's property.

[0141] Turning now to FIG. 7, a property sheet entry process `440` is shown. In using the property sheet, a user enters an expression (step `442`). The expression is then parsed (step `444`) and checked for validity (step `446`). In the event the expression is invalid, the process clears the runtime value using the design value as needed (step `448`) and displays an error message (step `450`). Otherwise, in the event that the expression is valid, the process creates a function and stores the function as the run-time value (step `452`). Next, the process determines whether the function is a constant (step `454`), and if so, clones the function and stores the design-time value in place thereof (step `456`).

[0142] Next, the process invalidates a byte code execution image, which contains a run-time executable code for the byte code (step `458`). It then checks whether the run-time display needs to be automatically updated (step `460`). If so, byte code is generated (step `470`) and executed (step `472`). From step `460` or step `472`, the process exits (step `474`).

[0143] Turning now to FIG. 8, a process `820` for visually manipulating an object is detailed. First, the process determines which object property maps to which changed attribute (step `822`). Next, a constant function is created to represent the new attribute value which is then stored as run-time property value and design-time property value (step `824`). The byte code execution image is then invalidated (step `826`). The process then checks whether the run-time display needs to be automatically updated (step `828`). If so, the byte code is generated (step `830`). Step `830` is illustrated in more detail in FIG. 9. From step `830`, the process then executes the byte code (step `832`). Step `832` is illustrated in more detail in FIG. 11. From step `828` or step `832`, the process exits (step `834`).