CALM Developer Guide

I. Burak Ozyurt

October 15, 2008

Contents

List of Figures

Chapter 1

Form Designer

1.1 Form Components Model

CALM represents an online form by a composition of containers and display components. This representation is called the form components model. The class diagram for the form components model is shown in Figure 1.1. A display component is a general term for any construct that acts as a building block of a form. A display component has an identity, a bounding box, a location, a parent (in most cases). It can draw itself when requested. It can serialize itself in XML format or deserialize from XML. Certain types of display components can contain other display components. Those display components are called *containers*. Every display component needs to implement the interface *IDisplayComponent*. The common functionality is provided by abstract class *AbstractDisplayComponent*. Currently, there are seven concrete display components available corresponding to (multi-line, styled) static text, text input field, checkbox, radio button, dropdown, textarea HTML form fields and buttons with predefined actions.

1.1.1 CALM Document Representation

The generated online forms by CALM needs to survive between CALM user sessions, hence they should be persisted in a format, that can be used to recreate the layout/score association work done previously on it. Each CALM online clinical assessment is saved in a XML file as defined in CALM document specification document and in CALM document XML schema as defined in *calm_document.xsd* which is available under CVS with the rest of the CALM source distribution. The main class that represents a CALM document is *Document*, which maintains a list of form pages as *Page* objects. A document is persisted in XML by a call of *saveDocument* method of a *Document* class and loaded into memory by a call to *loadDocument* method.

1.2 Property Editors/Customizers

CALM follows JavaBeans specification and can be seen as a kind of bean container. The public properties of all form element beans and also other non-visual beans are made modifiable by the use of property editors or bean customizers for more complex beans. The *PropertyEditorPanel* class provides a generic UI component for modifying the public properties of a Java bean. Most of the form element beans implementing *IDisplayComponent* have at least a BeanInfo class to provide explicit information about their corresponding bean to guide introspection so that only certain fields from available public properties are user modifiable and/or correct getters and setters are called for the properties. Also with non-primitive properties with a range of possible values, custom property

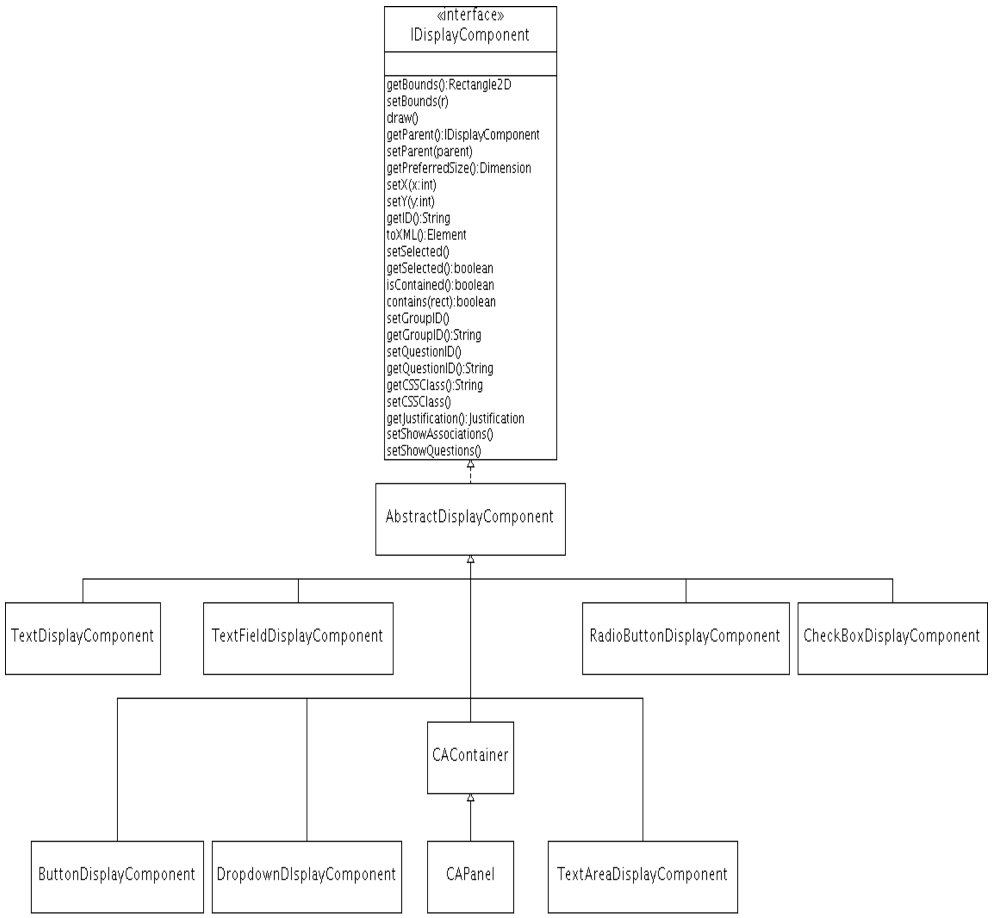

Figure 1.1: Simplified class diagram for Form Components Model.

editors are provided. For example, each form element bean has a justification property provided by *Justification* class. The property editor for displaying the possible justification field as a dropdown is *JustificationEditor*. The naming (Editor suffix) is mandated by JavaBeans specification to recognize the custom property editor. Certain beans require more than modification of their properties to customize. For those beans, a bean customizer is provided. For example the button form element bean has a customizer named *ButtonDisplayComponentCustomizer*, to provide question ranges and operation type selection for *Skip* button. The bean info class *ButtonDisplayComponentBeanInfo* registers the customizer for the bean.

1.3 Layout Management

Each physical page of an paper assessment form usually maps to a single HTML form. One of the main responsibilities of CALM is to provide a layout management mechanism for the online assessment form pages. Every display element layout manager class needs to implement the *ILayoutManager* interface as depicted in class diagram in Figure 1.2. CALM has currently a single layout manager class namely *CAGridLayout* which allows the display components and containers to be arranged in rectangular grids consisting of (ir)regular sized rows and columns. Each *CAContainer* or its subclass *CAPanel* is associated with a single layout manager. *CAPanel* class is used to represent the outermost container encapsulating a form page as shown in CALM layout pane. *CAGridLayout* layout manager class maintains grid cell information in a *GridCellInfo* class. The layout manager is responsible for satisfying the constraints associated with each grid cell by solving the constraint equations resulting in appropriate sizing of the grid cells containing display elements. There are currently two different type of constraints that can be imposed on the size of a grid cell. However, only percentage based grid cell constraints must be used all the time. Multiple cell span constraints are no longer supported.

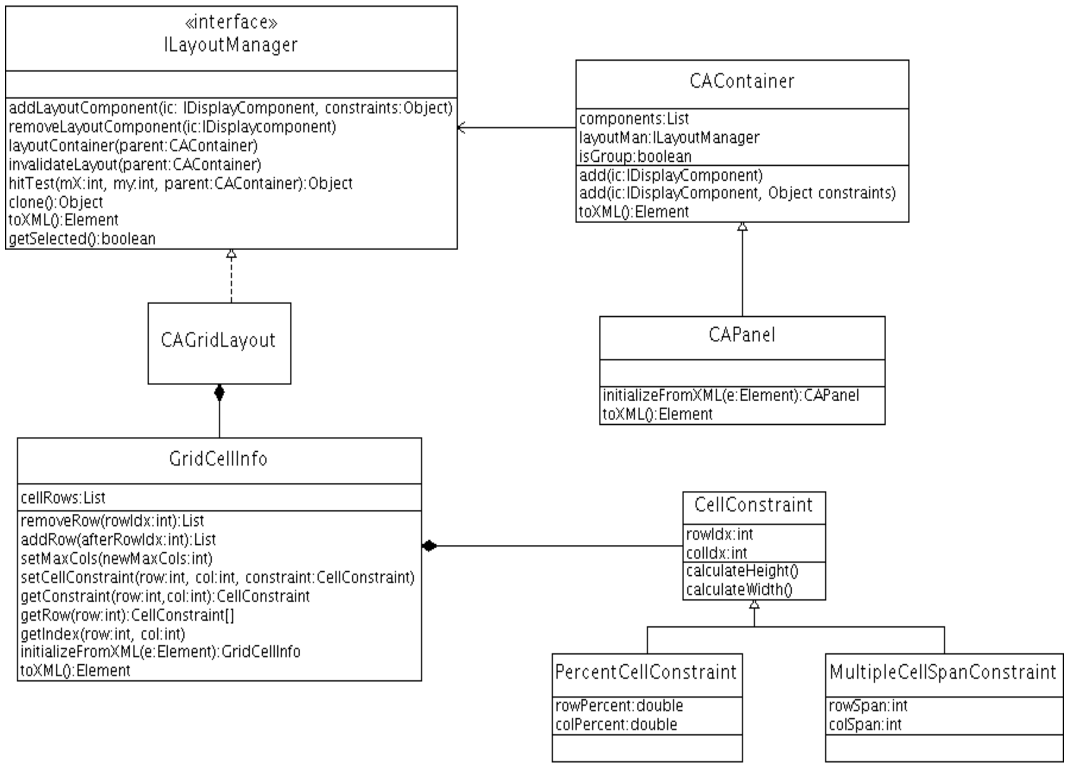
«interface»
ILayoutManager
addLayoutComponent(ic: IDisplayComponent, constraints:Object)
removeLayoutComponent(ic:IDisplaycomponent)
layoutContainer(parent:CAContainer)
invalidateLayout(parent:CAContainer)
hitTest(mX:int, my:int, parent:CAContainer):Object
clone():Object
toXML():Element
getSelected():boolean
CAContainer
components:List
layoutMan:ILayoutManager
isGroup:boolean
add(ic:IDisplayComponent)
add(ic:IDisplayComponent, Object constraints)
toXML():Element
CAGridLayout
CAPanel
initializeFromXML(e:Element):CAPanel
toXML():Element
GridCellInfo
cellRows:List
removeRow(rowIdx:int):List
addRow(afterRowIdx:int):List
setMaxCols(newMaxCols:int)
setCellConstraint(row:int, col:int, constraint:CellConstraint)
getConstraint(row:int,col:int):CellConstraint
getRow(row:int):CellConstraint[]
getIndex(row:int, col:int)
initializeFromXML(e:Element):GridCellInfo
toXML():Element
CellConstraint
rowIdx:int
colIdx:int
calculateHeight()
calculateWidth()
PercentCellConstraint
rowPercent:double
colPercent:double
MultipleCellSpanConstraint
rowSpan:int
colSpan:int

Figure 1.2: Simplified class diagram for Layout Manager.

1.3.1 CAGridLayoutCustomizer - Laying out a container

Each container has a layout manager class associated to handle how the contained display elements and/or containers are laid out visually on CALM layout pane. The *GridLayoutCustomizer* is the customizer for the *CAContainer* bean (See Figure 1.3). This customizer provides a preview panel where you can also manipulate the grid layout by mouse and add/delete rows from anywhere of the grid using the *PreviewPanel* inner class. The constraints associated with a selected cell is modified using the inner class *PercentConstraintEditorPane*. The preview panel class *PreviewPanel* uses an instance of *PreviewPanelMouseHandler* for mouse event handling, while delegating popup menu handling to the inner class *PopupMenuHelper*.

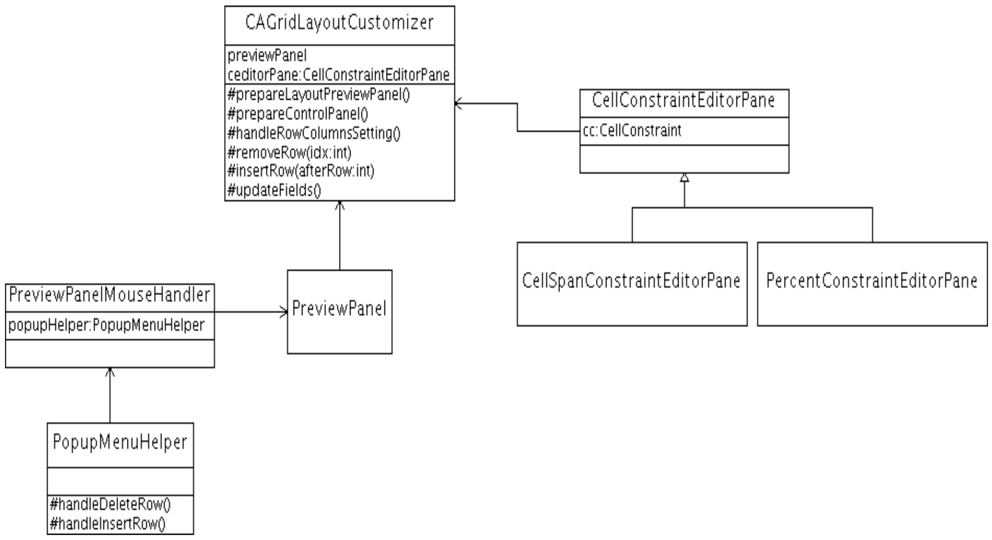

Figure 1.3: GridLayoutCustomizer class diagram.

Chapter 2

Form View - Backing Model Representation

2.1 Building/Modifying Assessment Visually

The clinical assessment form pages are simply a view of the backing data model namely the assessment domain object. An assessment domain object contains a hierarchy of score domain objects. Each score object can have zero or more scorecode domain objects. Also each score domain object can have an assessment item domain object representing the question text. In CALM, the assessment domain object concept is visualized by a tree view in the left side of the CALM screen. The tree panel is an instance of *AssessmentMaintenancePanel* class, which maintains the tree view and provides popup menu for creating/updating the assessment domain model. Each tree node is a view of the backing domain object. For mandatory fields, the backing domain object is a simple string; for the assessment node, it is an instance of *AssessmentInfo* class; for a score node, it is an instance of *ScoreInfo* class.

2.1.1 Actions on a score node

- Add Subscore - adds a subscore to the selected score node using the *ScoreInfoDialog* dialog box class.
- Add Score Codes - adds a score code (enumerated value) to the selected score node using the dialog box defined in *ScoreCodeDialog* class.
- Properties - shows properties on the backing model object (ScoreInfo in this case) using *PropertyEditorPanel* generic property editor panel in a dialog box.
- Update - allows updating of the properties of the selected score node via *ScoreCodeDialog* class.
- Update Score Codes - allows updating of the score codes for the selected score node via *ScoreCodeDialog* class.
- Delete Score - removes the selected score from the assessment tree.
- Associate - starts association (binding process) for the selected score node with a form input element or a logical group of mutually exclusive form input elements. This process is explained in more detail in the following section.

2.2 Associating scores with form input fields

The singleton *AssociationHelper* acts as the repository of form input field to assessment score and mandatory field associations. It is also an event dispatcher notifying registered interested parties on the change in association information to propagate change info in a decoupled manner synching up different parts of CALM. The different types of associations are represented by subclasses of *Association* class. A mandatory field to form input/logical group mapping is represented by *MandatoryFieldAssociation* class.

The *AssessmentMaintenancePanel* and the form layout panel class *ScoreLayoutPanel* beans communicate with each other using bounded property *SelectedModelDataInfo* to coordinate binding a score with a form input/logical group mapping. During initialization, *ClinicalAssessmentLayoutManager* creates both beans and registers property change listeners to each other.

1. User selects a score in *AssessmentMaintenancePanel* and chooses *Associate* from the popup menu.
2. The *handleAssociate* method on *ModelTreeMouseAdapter* creates a *SelectedModelDataInfo* object encapsulating the score/mandatory field selected and sets the bound property on *AssessmentMaintenancePanel*.
3. Since *ScoreLayoutPanel* is registered as a listener for the properties of *AssessmentMaintenancePanel*, it receives a property change event, checks if it is the kind it has interested in, finds it is an association request, changes itself into association mode.
4. The user clicks on the form input/ logical group in the form layout panel.
5. The display component selected for association is signaled back to property change listeners of the *SelectedComponentInfo* property which is a wrapper around the selected display component.
6. Since *AssessmentMaintenancePanel* is a listener on this property change event, it receives the *SelectedComponentInfo* in the property change event. It checks if the display component can be associated with the score, if so via *AssociationDialog* dialog box, collects score code association user input (if any) and finishes the association process.

2.3 Data Source Interaction

The data access layer classes for CALM are summarized in Figure 2.1.

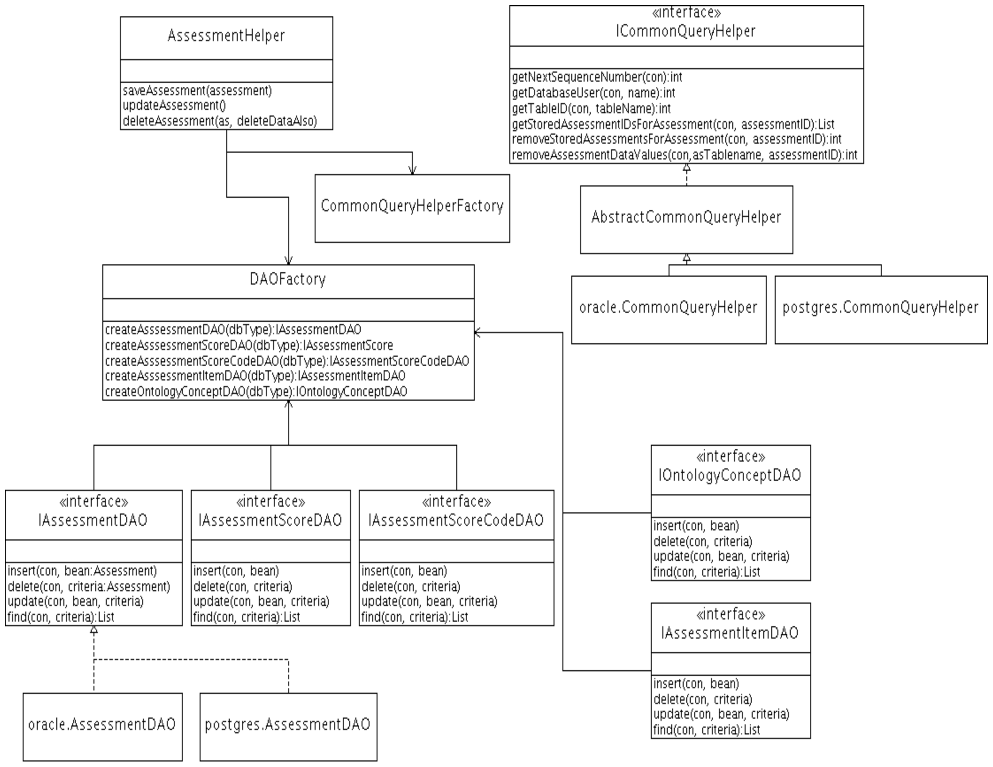
AssessmentHelper
saveAssessment(assessment)
updateAssessment()
deleteAssessment(as, deleteDataAlso)
«interface»
ICommonQueryHelper
getNextSequenceNumber(con):int
getDatabaseUser(con, name):int
getTableID(con, tableName):int
getStoredAssessmentIDsForAssessment(con, assessmentID):List
removeStoredAssessmentsForAssessment(con, assessmentID):int
removeAssessmentDataValues(con,asTablename, assessmentID):int
CommonQueryHelperFactory
AbstractCommonQueryHelper
oracle.CommonQueryHelper
postgres.CommonQueryHelper
DAOFactory
createAsssessmentDAO(dbType):IAssessmentDAO
createAsssessmentScoreDAO(dbType):IAssessmentScore
createAsssessmentScoreCodeDAO(dbType):IAssessmentScoreCodeDAO
createAssessmentItemDAO(dbType):IAssessmentItemDAO
createOntologyConceptDAO(dbType):IOntologyConceptDAO
«interface»
IOntologyConceptDAO
insert(con, bean)
delete(con, criteria)
update(con, bean, criteria)
find(con, criteria):List
«interface»
IAssessmentDAO
insert(con, bean:Assessment)
delete(con, criteria:Assessment)
update(con, bean, criteria)
find(con, criteria):List
«interface»
IAssessmentScoreDAO
insert(con, bean)
delete(con, criteria)
update(con, bean, criteria)
find(con, criteria):List
«interface»
IAssessmentScoreCodeDAO
insert(con, bean)
delete(con, criteria)
update(con, bean, criteria)
find(con, criteria):List
«interface»
IAssessmentItemDAO
insert(con, bean)
delete(con, criteria)
update(con, bean, criteria)
find(con, criteria):List
oracle.AssessmentDAO
postgres.AssessmentDAO

Figure 2.1: Simplified class diagram for CALM data access layer.

Chapter 3

Code Generation

Ultimately, the main purpose of CALM is to generate dynamic online forms for HID web application, to be processed in a generic fashion by GAME. Once the pages making up a clinical assessment is laid out, the corresponding assessment metadata for the database is created, the scores are bound (associated) with their corresponding display elements or logical groups of display elements acting as a single form input mechanism and the question metadata is provided, CALM is ready to generate the necessary code for the HID web application. The generated code is comprised of a Struts form bean, two JSPs for each page of the online form, one as the Struts Tiles template binder, the other for the actual body of the form and the modification to *struts-config.xml.template* file.

A simplified class diagram for CALM code generator is shown in Figure 3.1. The contract for each code generator is provided by the interface *IGenerator*. The common functionality for all code generators are provided by the abstract class *AbstractGenerator*. The form beans are generated by *FormBeanGenerator* class, while the JSP corresponding to the laid out form page is generated by *StrutsJSPGenerator* class. The builder class *StrutsCodeGenerator* is responsible for coordinating code generation for the whole clinical assessment including struts configuration file updating. The necessary configuration input is provided via the *StrusCodegenConfig* class.

3.1 Struts JSP Generation

CALM maps the container hierarchy to a hierarchy of HTML tables. CSS based layout management is still at its infancy and it is not as fine granular as what you can do with tables for layouts with relative positions and not supported consistently across browsers. The *StrutsCodeGenerator* class generates the HTML table hierarchy recursively starting from the provided root *CAContainer* class for a form page. The *StrutsCodeGenerator* class uses *CodegenUtils* utility class for lower level functionality needed including

- determining their outermost container for multi-answer questions
- converting a grid layout row into equivalent HTML table row.

The generated JSP uses Struts custom JSP tags and Javascript for client side dynamic behavior. Most of the nearly static portions of the generated JSP including Javascript snippets are loaded from the *jsp_snippets.txt* file as located by *code.snippet.dir* application property as defined in *caslayout.properties* file.

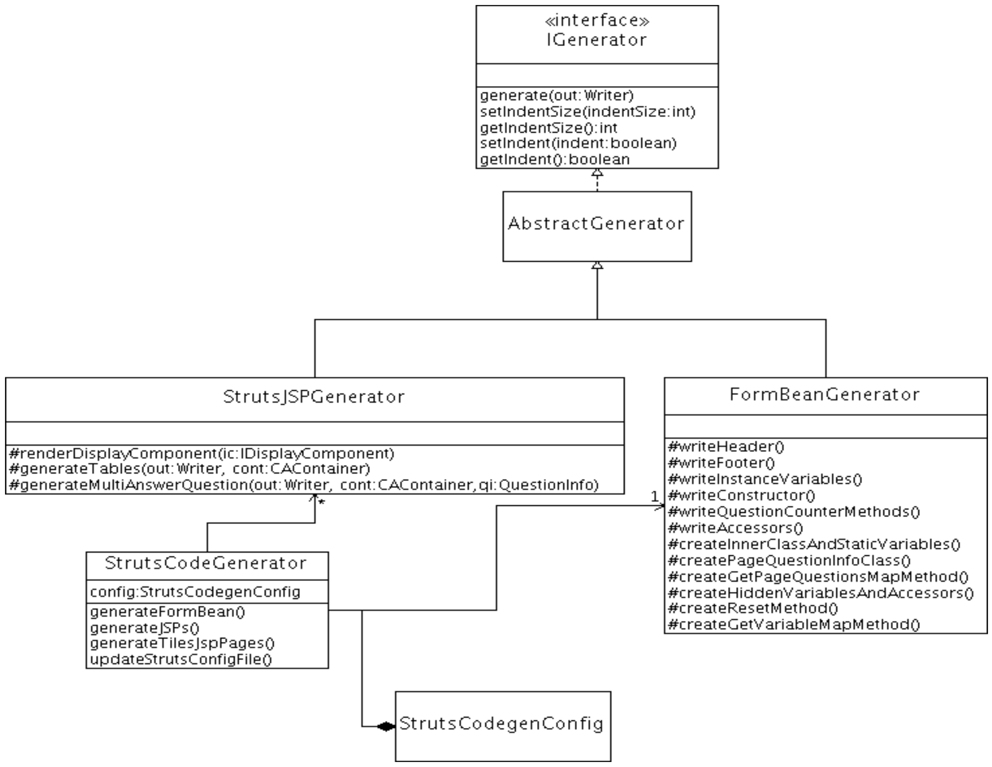
«interface»
IGenerator
generate(out:Writer)
setIndentSize(indentSize:int)
getIndentSize():int
setIndent(indent:boolean)
getIndent():boolean
AbstractGenerator
StrutsJSPGenerator
#renderDisplayComponent(ic:IDisplayComponent)
#generateTables(out:Writer, cont:CAContainer)
#generateMultiAnswerQuestion(out:Writer, cont:CAContainer,qi:QuestionInfo)
FormBeanGenerator
#writeHeader()
#writeFooter()
#writeInstanceVariables()
#writeConstructor()
#writeQuestionCounterMethods()
#writeAccessors()
#createInnerClassAndStaticVariables()
#createPageQuestionInfoClass()
#createGetPageQuestionsMapMethod()
#createHiddenVariablesAndAccessors()
#createResetMethod()
#createGetVariableMapMethod()
StrutsCodeGenerator
config:StrutsCodegenConfig
generateFormBean()
generateJSPs()
generateTilesJspPages()
updateStrutsConfigFile()
StrutsCodegenConfig
1
*

Figure 3.1: Simplified class diagram for CALM code generator.

3.2 Struts Form Bean Generation

The *FormBeanGenerator* class is responsible for generating a java bean responsible as the conduit between the HTML form pages and the underlying generic controller (GAME) in the HID web app. The form bean is used to populate the viewed forms and retrieve user input from the submitted HTML forms. Each form bean is associated with a single clinical assessment and responsible for data mapping of each HTML form page the clinical assessment is comprised of. Besides the assessment scores as properties, it includes metadata related to the page information of the scores, mandatory field related metadata, question related metadata and mapping for hidden variables necessary for missing value and multiple-answer question handling. The nearly static portions of the generated form bean are loaded from the *java_snippets.txt* file as located by *code.snippet.dir* application property as defined in *caslayout.properties* file. The generated form beans must be saved under the */clinical/web/game/forms* relative to HID web app home directory as specified in *Mandatory Information for Code Generation* dialog box. CALM remembers most of the almost invariant user input by serializing its operation state when you exit CALM and deserialize when you start it again including the HID web app home directory.

3.3 Updating Struts configuration file

In Struts web framework, page navigation form bean, action class definitions are defined in one or more configuration files. CALM updates *struts-config.xml.template* file once it generates the form bean and JSP pages. The *updateStrutsConfigFile* method in *StrutsCodeGenerator* class is the entry point for this operation. The generated code is put inside the sections of the configuration file delimited by the following markers

```
<!-- CALM generated start -->

<!-- CALM generated end -->
```

First, the existence of the configuration is checked to support incremental updates to the configuration file. This is acomplished by a call to *hasTheConfiguration* method which parses the configuration file and checks for the existence of the form bean and JSPs to be configured and returns an *ExistingActionFormInfo* object representing already existing JSPs and/or form bean configuration. If there are new JSPs added in this CALM session to the assessment, only those actions will be added to the Struts config file.