SALESPOINT
a look behind the Framework

by Andreas Bartho, Simon Fein and Benjamin Wetzel

February 6, 2004

Contents

Students of computer science at the Technical University Dresden and the UniBW Munich have to fulfill a Software-internship during their studies. Most of those students use the so called SalesPoint Framework for their internship. This framework, which is written in JAVA, offers basic services and methods to develop small business applications or simulations thereof.

Although there is an extensive documentation for this framework, many students have the desire to understand how the framework or single modules thereof work and are implemented, be it out of interest or to make special changes for their own applications which have originally not been meant to be supported by the Framework. This documentation is especially for those students.

Good JAVA-knowledge and experience in programming with SalesPoint are presumed. Basic concepts of SalesPoint are not explained. They can be found in the technical description.

This technical report is also written for people who are developing and enhancing the framework. Before beginning to implement new functions or to correct mistakes in the framework it is necessary to exactly understand what happens within the framework and what side effects certain methods can have.

The explanations of the framework functions are geared to the source code, which is freely available in the Javadoc. Under certain circumstances it can be helpful to refer to these sources for a better understanding.

This report covers only parts of the framework's functions, namely the Data Management with DataBaskets and Catalogs, the Process Management and the Display Management.

Chapter 1

Data Management

1.1 DataBaskets

1.1.1 Design of DataBaskets

DataBaskets are containers for information which describe transactions. This includes removing, adding and editing of CatalogItems or StockItems. The information itself is represented by DataBasketEntry objects. DataBasketEntries are not directly stored in DataBaskets. Instead the DataBasket consists of one or more SubDataBaskets which in turn contain DataBasketEntries.

Through the possibility of saving many SubDataBaskets, the user has the option of having many ”saving points” in a DataBasket. The problem of data corruption (changes in a SubBasket must also be changed in the other SubBaskets, which is often forgotten) must be solved by the programmer, the framework does not test the consistency.

In practice usually only one SubBasket is used for every DataBasket. This is also the standard configuration when creating a DataBasket.

1.1.2 Design of DataBasketEntries

DataBasketEntries describe transactions of StockItems or CatalogItems and contain the following attributes:

- `mainKey`: differentiation of the type of DataBasketEntry: for StockItems or CatalogItems
- `secondaryKey`: the key of the CatalogItems or StockItems, for which a transaction is described by the DataBasketEntry

- `value`: the CatalogItem or StockItem, for which a transaction is described by the DataBasketEntry. The key of this item corresponds to the `secondaryKey` or the DataBasketEntry

- `source`: the container (Catalog or Stock), from which the item is removed. Item refers to the CatalogItem or StockItem for which this DataBasketEntry was created

- `destination`: the container (Catalog or Stock) to which the item is added

- `handled`: describes, whether a commit or rollback of this DataBasketEntry has already been performed

1.1.3 Design of DataBasketConditions

DataBasketConditions are filters for DataBasketEntries. Hence they contain the largest number of attributes of DataBasketEntries:

- `mainKey`: differentiation of the type of DataBasketEntry: for StockItems or CatalogItems

- `secondaryKey`: the key of the CatalogItems or StockItems for which a transaction is described by the DataBasketEntry

- `value`: the CatalogItem or StockItem for which a transaction is described by the DataBasketEntry

- `source`: the container (Catalog or Stock) from which the item is removed

- `destination`: the container (Catalog or Stock) to which the item is added

Additionally there is a `match()` method, which returns `true` but needs to be overridden in special cases. The purpose of this method will be covered in chapter ”Iteration over SubDataBaskets”.

In case an attribute has the value `null`, the framework will interpret it as matching.

DataBasketConditions will come into use when iterating over the SubDataBaskets, for example when a `commit()` or `rollback()` is used. (Refer to the section ”Iteration over SubDataBaskets”)

1.1.4 SubDataBaskets

Design of SubDataBaskets

The main aspect of a SubDataBasket is a HashMap `dbeCatagories`. Within `dbeCategories` there are another two HashMaps. One is for StockItemDBEntries and one is for CatalogItemDataBasketEntries. Each of these maps contains lists. Stored within these lists are the actual DataBasketEntry objects. The key through which the lists are saved in their map is the same key as that of the item which is to be saved as a DataBasketEntry (therefore the `secondaryKey` of the DataBasketEntry). This enables the saving of items with the same name and of the same type, for example saving two CatalogItems, both called ”item1”, in the same SubBasket, hence allowing collisions.

An example would be: if three times five StockItems ”item1” are erased from a CountingStock, there are three DataBasketEntries which are written into the same list within the map for the StockItemDBEntry. The framework can access these lists through the key ”item1”. The three DataBasketEntries are identical:

- `mainKey`: `__MAINKEY:_STOCKITEM_IMPL`
- `secondaryKey`: `item1`
- `value`: 5
- `source`: The Stock object, from which the items have been removed
- `destination`: `null`

Iteration over SubDataBaskets

SubDataBaskets have their own iterator, which can be accessed through the method `iterator (DataBasketCondition dbc, boolean fAllowRemove, boolean fShowHandled)`.

The nested structure of SubDataBaskets makes it necessary to iterate in 3 steps.

- over the HashMap `dbeCategories`. The used iterator is called `iCategories`.
- over the HashMap that contains the lists with the StockItemDBEntries and CatalogItemDataBasketEntries. The used iterator is called `iSubCategories`.
- over the lists in the SubCategories, which themselves consist of the actual DataBasketEntries. The used iterator is called `iItems`.

All three iterators are defined globally in the class SubDataBasket.

hasNext() and next()

The methods `hasNext()` and `next()` both make use of the method `findNext(boolean fGet)`. While `hasNext()` executes `findNext()` with the parameter `fGet = false` and returns the result value it receives, `fGet` for `next()` is `true`. The returned value is the global variable `dbeCurrent`, in which the returned `DataBasketEntry` is saved. The actual saving of the DataBasketEntries in `dbeCurrent` is done in the method `checkItems(boolean fGet)`.

findNext(boolean fGet)

The method consists of a do-while loop. It starts with `checkSubCategories(boolean fGet)`. If `true` is returned, there was a SubCategory with a fitting DataBasketEntry found in the category (which is either the Map for StockItems or CatalgItems). In case `fGet` is `true`, an entry was saved in `dbeCurrent`. `True` is returned and the method finished.

If no DataBasketEntry was found, the next category must be scanned, if existing. This is achieved by letting the iterator `iCategories` continue running to the next category. For the new category an iterator is created and saved as `iSubCategories`, it will then be used by the method `checkSubCategories()`, during the next execution of the do-while loop, in order to iterate over the subcategories of the category.

In case a DataBasketCondition has been passed to the constructor of the iterator, there will be a check whether the `mainKey` of the condition is `null`, or if the `mainKey`s of the DataBasketEntry and DataBasketCondition are identical. If not, this category will be skipped and the following category will be examined the next time the do-while loop is executed. If there is no category left, `findNext()` finishes returning `false`.

checkSubCategories(boolean fGet)

The method works in analogy to `findNext()`, only one hierarchy level below it. Thus, at the beginning of the do-while loop, it will be tested whether fitting DataBasketEntries can be found in the items of the subcategory using `checkItems(boolean fGet)`. If yes, the method will be ended with `return true`, else it will go to the next subcategory, on which the iterator `iItems` is defined, to iterate over the items of that next subcategory. Here again the DataBasketCondition must be considered. A subcategory is taken into consideration, if the `secondaryKey` of the condition is identical to the key of the subcategory or `null`.

checkItems(boolean fGet)

First the method checks `dbeNext`. If it is not `null`, the presence of a fitting DataBasketEntry has already been detected by an earlier iteration step and `true` is returned. If additionally `fGet` is `true`, meaning `checkItem()` was executed by the `next()` method, `dbeNext` is saved in `dbeCurrent`. Yet if `dbeNext`is `null`, it is unknown whether another DataBasketEntry exists. Therefore `iItems` iterates over the current list of items. Every DataBasketEntry is tested for matching the search criteria. A DataBasketEntry matches if:

- it has been ”handled” (see Commit of DataBasketEntries) and `fShowHandled` is `true`
- the DataBasketCondition is `null` or matches the DataBasketEntry, thus corresponding in `source`, `destination` and `value`. If the `value` of the `DataBasketCondition` is `null`, the `match()` method decides whether the DataBasketCondition fits the search criteria.

If a suitable DataBasketEntry is found, it is saved in `dbeNext` or, if `fGet` is `true`, directly in `dbeCurrent`. Finally the method is finished returning `true`. In case no suitable entry is found `false` is returned.

1.1.5 Commit of DataBasketEntries

A `commit()` on DataBasketEntries is passed to the SubDataBaskets by the DataBasket. A DataBasketCondition can be specified. The `commit()` will then only be executed for DataBasketEntries which match the DataBasketCondition. Alternatively a `commitSubBasket()` can be used to address a specific SubDataBasket which should be committed. Accessing a specific SubBasket for a commit and at the same time giving it a DataBasketCondition as a filter is not possible. If needed, this function can be easily implemented in a subclass of DataBasketImpl.

SubDataBaskets pass the commit to their entries. It is iterated over the SubDataBasket using a possibly passed DataBasketCondition as a filter. (see: Iteration over SubDataBaskets). For each DataBasketEntry, which is returned by the iterator a `commit()` is executed, provided the DataBasketEntry is not marked as `handled`. Finally the `remove()` method of the iterator is used to delete the DataBasketEntry from the SubDataBasket.

When committing a DataBasketEntry, `handled` is set to `true` to indicate that the entry has been processed and does not represent an active transaction. After this there is a `commitRemove()` executed for the source of the DataBasketEntry, which is saved as the attribute `source`. For the destination of the entry, which is described by the attribute `destination`, a `commitAdd` is performed. This causes temporarily added or removed items to be finally added to or removed from their source. More details about the methods can be found in the documentation on Catalogs and Stocks.

1.1.6 Rollback of DataBasketEntries

A `rollback()` functions in analogy to `commit()`. During a `rollback()` on a DataBasketEntry a `rollbackAdd()` or `rollbackRemove()` is executed, which leads to the removal of temporarily added items and addition of temporarily removed items.

Further details to these methods can be found in the documentation on Catalogs and Stocks.

1.1.7 Meaning of the Flag fHandled

On first view, the flag `fHandled` appears unnecessary since the fully processed DataBasketEntry is deleted from the SubDataBasket when a `commit()` or `rollback()` is executed. Yet it is possible to execute a `commit()` or `rollback()` ”by hand” without erasing it from the SubDataBasket. To ensure data integrity every processed DataBasketEntry is invalidated via `handled`.

1.2 Catalogs

1.2.1 Design of Catalogs

Catalogs are containers for CatalogItems. The CatalogItems within a Catalog can be in different states:

- They can belong to the catalog.
- They can temporarily belong to the catalog, this case is given when added with a DataBasket.
- They can be temporarily removed from the catalog with the help of a DataBasket.
- They can have the status ”editable”.

A catalog knows all its CatalogItems and every CatalogItem knows its catalog. The term ”Parent Catalog” is applied here.

The above mentioned four states are represented by four Maps:

- `mItems`: contains the CatalogItems of the catalog.
- `mTemporaryRemoved`: contains CatalogItems which have been removed through a DataBasket until a commit or rollback is executed.

- `mTemporaryAdded`: contains CatalogItems which have been added through a DataBasket until a commit or rollback is executed.
- `mEditingItems` contains CatalagItems presently editable.

1.2.2 Editability of Catalogs

As Catalogs are CatalogItems themselves, they can be ordered hierarchically. The catalog in which another one is contained, is the parent catalog. The editability of catalogs follows the same rules as CatalogItems. A catalog is editable when one of the following applies:

- it does not have a parent catalog
- it is contained in `mEditingItems` of the parent catalog

1.2.3 Addition of CatalogItems

CatalogItems are added to a catalog via the `add(CatalogItem ci, DataBasket db)` method. First it is verified whether the catalog is editable, if not, the method is terminated by a `NotEditableException`.

Then it is verified, whether the CatalogItem is contained in one of the maps, hence already in the catalog. In case the CatalogItem is contained in `mItems` or `mTemporaryAdded`, a `DuplicateKeyException` is returned and the operation terminated. A CatalogItem cannot appear twice in the same catalog.

In case the CatalogItem is in `mTemporaryRemoved`, adding it would nullify the previous removal. Precondition herefore is that both removing and adding use the same DataBasket. In this case, a `rollback()` is executed on the temporary removal and the modification counter increased, else a `DataBasketConflictException` is thrown. If during this time the temporarily removed CatalogItem was added to another catalog, there will also be a `DataBasketConflictException`.

After these tests are completed, the actual adding is performed. In case `null` was used as a DataBasket, the CatalogItem is directly saved in `mItems`, else in `mTemporaryAdded`. In the second case an entry must be inserted into the DataBasket, which describes the addition. A CatalogItemDataBasketEntry is created, which has the current catalog set as its destination. It is possible, that there is already an existing DataBasketEntry for the CatalogItem. This is given when the CatalogItem was temporarily removed from another catalog. In this case the entry is updated, so that the temporary addition is recorded.

Finally the modification counter is increased, the CatalogItem is given a reference to the catalog it belongs to and listeners are informed that the CatalogItem has been added.

1.2.4 Removal of CatalogItems

CatalogItems are removed either via `remove(CatalogItem ci, DataBasket db)` or `remove(String sKey, DataBasket db)`. Additionally the removed CatalogItem is returned.

The method `remove(String sKey, DataBasket db)` mainly performs a few tests and then calls `remove(CatalogItem ci, DataBasket db)` itself. Therefore the latter shall be further examined.

Before the removal it is tested whether the catalog is editable, if not, a `NotEditableException` is thrown and the removal cancelled. As a next step it is examined in which form the CatalogItem is contained in the catalog.

If it is in `mTemporaryAdded`, a removal may have the same effect as undoing a temporary addition. This is only possible if the same DataBasket was used for addition and removal. If so, listeners are asked whether the CatalogItem can be removed. If no listener vetoes, a `rollback` of the temporary removal is executed. In case the CatalogItem has not only been added to this catalog but also temporarily removed from another, a `rollback()` cannot be performed. The reason herefore is that the `rollback()` will also undo the temporary removal from the other catalog. Instead the `rollback()` of the temporary addition is simulated: the `destination` field of the DataBasketEntry is set to `null`, the CatalogItem is removed from `mTemporaryAdded`, the modification counter is increased and listeners are informed of the changes. Finally the reference of the CatalogItem to its catalog is erased and a CatalogItem returned.

In case the CatalogItem is in `mTemporaryRemoved`, it cannot not be deleted again. Therefore a `DataBasketConflictException` is thrown.

Further it is tested whether the CatalogItem is in `mItems`. If not, it is not part of the catalog and can therefore not be deleted. The `remove()` method is exited returning `null`. Otherwise the removal starts.

As a first step listeners are queried whether the CatalogItem may be removed. If no veto is returned, the CatalogItem is deleted from `mItems`. If the used DataBasket is not `null`, the item is only temporarily removed. Hence it is saved in `mTemporaryRemoved`. If the DataBasket contains an entry stating that the CatalogItem has already been temporarily added to another catalog, the `source` field of this entry is set to the current catalog's name. This describes a shifting operation, as can be achieved with a TwoTableFormSheet. Otherwise a new DataBasketEntry is saved in the DataBasket recording the removal from the catalog.

Finally the modification counter is increased, the reference from the item to the catalog is removed, listeners are informed of the removal and the CatalogItem is returned.

1.2.5 Retrieval of CatalogItems

CatalogItems can be fetched from the catalog via `get(String sKey, DataBasket db, boolean fForEdit)`. The parameter `fForEdit` indicates whether or not to make the returned CatalogItem editable. A CatalogItem retrieved as none-editable can also be edited, but only for editable ones the DataBasket provides the possibility to rollback changes.

Before the desired CatalogItem is returned some tests are performed. If the CatalogItem is to be fetched as editable, the catalog must also be editable. If this is not the case, a `NotEditable-Exception` is thrown.

If the CatalogItem is in `mTemporaryAdded` and there is an associated entry in the DataBasket, the item is returned in case it should not be editable. However, if it should be editable, a test for a shifting operation is performed. If this is the case, the CatalogItem has been removed temporarily from a different catalog. Changes to the retrieved CatalogItem would have to be applied to the CatalogItem which is stored in `mTemporaryRemoved` of the other catalog and vice versa. As this is a rare case and would result in a complex implementation, editable retrieval of shifted CatalogItems is not permitted and a `DataBasketConflictException` is thrown instead. If the CatalogItem is not involved in a shifting operation, listeners are queried if the CatalogItem should allowed to be editable. If no listener vetoes, the CatalogItem is stored in `mEditingItems`, listeners are informed that the CatalogItem is now editable and the item is returned. The `get()` method is thus finished.

If the CatalogItem is not temporarily added, it might be located at `mItems`. If it is, it is returned in case it should not be editable. If an editable CatalogItem is required, precautions have to be taken that enable a `rollback()`. It should be noted that the DataBasket must not be `null` when retrieving an editable CatalogItem, because with a `null` DataBasket there is no possibility to store information that allow a `rollback()`.

First listeners are queried if the CatalogItem should be allowed to be editable. If no listener vetoes, a shallow clone of the CatalogItem is created using the item's `getShallowClone()` method which has to be implemented by the application programmer. The original CatalogItem is removed from `mItems` to `mTemporaryRemoved`, meaning it is temporarily removed. The shallow clone is added to both `mTemporaryAdded` and `mEditingItems`, meaning it is marked as temporarily added and editable. That followed two DataBasketEntries which describe the temporary addition and removal are created and stored in the DataBasket. Then the reference to the catalog is removed from the original CatalogItem and the shallow clone receives a reference to the catalog. Listeners are informed about the removal and additon/editability of the CatalogItem and the shallow clone, respectively. The modification counter is increased and the shallow clone is returned.

Once a CatalogItem has been made editable, future calls of `get()` will always return the shallow clone. It is irrelevant if `fForEdit` is `true` or `false`. The only possibility to make the CatalogItem non-editable again is to do a `commit()` or `rollback()`.

Saving or Discarding of Changes

Any changes to the CatalogItem are now applied to the clone. On `commit()` the old CatalogItem is removed and the shallow clone with all its changes becomes the new original. On `rollback()` the shallow clone is deleted and the original CatalogItem is copied back to `mItems`.

If the CatalogItem was added temporarily as mentioned previously, no shallow clone is created. Is a `commit()` or `rollback()` still possible? The answer is yes. If changes are made and approved by a `commit()`, the temporarily added CatalogItem is copied to `mItems` with all changes made, on `rollback()` it is removed. A shallow clone is not necessary because there is no original CatalogItem which might have to be restored.

Veto on Editing of CatalogItems

Before CatalogItems are made editable, CatalogChangeListeners of Stocks or Catalogs can veto. Stocks veto if they contain StockItems which are currently temporarily added or removed and refer to the CatalogItem to be made editable. Catalogs veto if they have a shallow clone. In that case the clone must be used to retrieve editable CatalogItems.

1.2.6 Name Change of CatalogItems

The SalesPoint framework provides interfaces and implementations for flexible enforcement of naming conventions. These are explained first because the renaming of CatalogItems is based upon these.

To force an object to obey special naming rules it must implement the Nameable interface. The actual rules are implemented in an object implementing the NameContext interface. The renaming is done by calling the `setName(String sName, DataBasket db)` method of the nameable object. This method should perform the following steps:

- Check for NameContext to be `null`
  The NameContext is saved as an attribute in the nameable object.

  1. If yes, the new name is assigned
     There are no rules to be obeyed.
  2. Otherwise
     - Method `checkNameChange(DataBasket db, String sOldName, String sNewName)` of the NameContext is called.
       It checks whether the desired name change is valid, that is, this method defines the naming rules and enforces their compliance. If the new name is not allowed, a `NameContextException` is thrown.

  - The new name is assigned.
  - Method `nameHasChanged(DataBasket db, String sOldName, String sNewName)` of the NameContext is called.
    It contains internal cleanup code, if necessary.

The actual implementation of the Nameable interface is done in the class AbstractNameable. CatalogItemImpl and StockItemImpl are subclasses of AbstractNameable. Consequently the implementation of the NameContext interface is done in CatalogImpl and StockImpl which need to permit or prohibit name changes of their items. The implemented rule in CatalogImpl is that no two items can have the same name. StockImpl does not allow any renaming. StockItems can only be renamed by renaming the belonging CatalogItem. However, on name change of CatalogItems the StockItems of StoringStocks must also be renamed. This done by the method `internalNameChange()` which temporarily detaches the NameContext of a StockItem and attaches it again after the name has been changed. This method should never be called directly by the application programmer.

CatalogItems and StockItems receive a reference to their NameContext when they are added to their Catalogs or Stocks, respectively.

Method checkNameChange()

The method `checkNameChange(DataBasket db, String sOldName, String sNewName)` of CatalogImpl checks whether the CatalogItem to be renamed is editable, whether there is no CatalogItem using that name and if the right DataBasket is used.

What happens in detail is the following:

- Check if the CatalogItem is in `mEditingItems`, if not, a `NameContextException` is thrown, stating that the CatalogItem must be made editable first.

- Check in `mItems`, `mTemporaryAdded` and `mTemporaryRemoved` if there is already a CatalogItem with the same name as the one to be set. If so, a `NameContextException`, stating that a CatalogItem with that name already exists.

- Check if the passed DataBasket is the same as the one the CatalogItem has been made editable with. If not, a `NameContextException` is thrown, stating that the DataBasket does not contain an entry for the CatalogItem to be renamed.

Method nameHasChanged()

After the actual name change `nameHasChanged(DataBasket db, String sOldName, String sNewName)` is executed. This method updates the key with which the renamed items are found by their HashMap. This is done by removing the affected CatalogItems from their maps and adding them again with their new name as a key. Then the DataBasket is updated.

What happens in detail is the following:

- The renamed CatalogItem which is still saved in `mEditingItems` under its old key `sOldName` is removed from `mEditingItems` and added again with its new name as a key.
- The same procedure is performed for `mTemporaryAdded`.
  Both of these maps contain the shallow clone created when the CatalogItem has been made editable. The original CatalogItem is not renamed and thus a name change can be rolled back.
- Update of the DataBasket
  When the CatalogItem had been made editable, a DataBasketEntry has been added describing the temporary addition of the shallow clone to the catalog. This entry is replaced by an entry that is almost identical, the only difference is the `secondaryKey` which now matches the new name of the CatalogItem.

Finally the listeners are informed about the name change.

Chapter 2

Process Management

2.1 Introduction to Processes

Interaction with SalesPoint applications is achieved by means of processes. Processes require an environment to run in, the ProcessContext. The Framework provides two of those contexts, SalesPoint and Shop$ProcessHandle. The latter is an inner class of the Shop meant for background tasks.

A process works like a finite automaton, it consists of states that are called Gates, and Transitions. Processes can be suspended. As long as a process is suspended it does not leave its current Gate, which means that no transitions are executed. Process suspension is necessary if the Shop's state is to be saved. Saving the Shop requires all processes to be halted. Processes can only be suspended on Gates.

Gates are used for the execution of potentially long-lasting actions. The most important implementation of the interface Gate is the class UIGate which is capable of displaying FormSheets and therefore enables user interaction.

Transitions are meant to allow gate changes. In addition application specific code can be executed during a transition, for example processing user input entered in FormSheets.

2.2 Start and Operation of Processes

2.2.1 Preparation of Process Start

A process is run on a SalesPoint by calling the SalesPoint's method `runProcess (SaleProcess p)`. The SalesPoint receives a reference to its process. In turn the process receives a reference to the SalesPoint and the SalesPoint's DataBasket. Following, the actual process is started with a call of `p.start()`.

If the process is used as a background process, the Shop's method `runProcess()` is executed. It creates a new ProcessHandle, establishes references similar to the SalesPoint's `runProcess()` method and inserts the process into the Shop's list of all active ProcessHandles. If the Shop is not suspended, a new background process will be started with `start()`, otherwise suspended with `suspend()`.

2.2.2 Process Start

If a process has been newly created it is initialized first. The flag `fSuspended`, which indicates a process's suspension, is set to `false`. The initial gate of the process is obtained by the method `getInitialGate()` which has to be implemented by the application developer. It is then assigned to the variable `gCurGate`. If the `start()` method is called to resume a previously suspended process the assignment of `getInitialGate()` to `gCurGate` is omitted.

Then a Thread `trdMain` which calls the method `main()` is created and executed. If the process is not resumed but started, the ProcessContext's method `processStarted()` is called. This allows a ProcessContext to react appropriately on a process start. If the ProcessContext is a SalesPoint, the currently displayed FormSheets and MenuSheets are removed. Then the SalesPoint unregisters itself from the list of display listeners. If the ProcessContext is a Shop$ProcessHandle, no action is taken on invocation of `processStarted()`. The next instruction is `onResumeOrStart()` which by default does nothing. It is a hook method that can be overridden by the application developer to execute special initialization code for his process. If an error occurs during `onResumeOrStart()` a PrintErrorGate is assigned to `gCurGate`.

2.2.3 Process Loop

Then the actual process loop is entered. It runs as long as `cCurGate` is not `null`, meaning the process has neither been finished nor suspended. Within the loop a SaleProcess$SubProcess `trdGate` is created and executed. In this document it will be referred to as gate thread.

SaleProcess$SubProcess is a subclass of Thread. It extends Thread by the ability to catch the Throwable `ProcessErrorError`. The gate thread's task is to return a transition and assign it to the variable `tCurTransition`. This happens by calling the `gCurGate`'s method `getNextTransition()`. If the process is suspended before it can return a transition, meaning `fSuspended` is `true`, `tCurTransition` is set to `null` and the process loop is left.

As soon as the gate thread has returned, a SaleProcess$SubProcess `trdTransition` is created and executed. It will be referenced as transition thread in this document. The transition thread returns the next gate to jump to by calling `tCurTransition.perform()`. The returned gate is saved in the variable `gCurGate`. If there is an attempt to suspend the process while the transition thread is still active, the suspension command will be ignored. But as soon as the transition thread has returned and a gate has been found, `onSuspended()` is called. This is a hook method for the application developer to override. By default this method is empty but it can be used to execute a certain action if a suspend command occurs. If the process has not been finished or suspended, the process loop is executed again.

2.2.4 Process Finish

After leaving the process loop the hook method `onFinished` is called. If the process has been finished and not only suspended, the process context's `processFinished()` method is also executed. If the context is a SalesPoint, `processFinished()` will remove the SalesPoint's reference to the process. Additionally the changes made by `processStarted()` on process start are cancelled: the SalesPoint registers as a FormSheetListener at the display again and FormSheet and MenuSheet are set.

If `processFinished()` of a ProcessHandle is executed, the reference of the finished process to the handle is removed and the handle itself is removed from the Shop's list of active ProcessHandles.

2.3 Function of UIGates

As mentioned above, a process is basically an alternation of gates and transitions. The most often used gates are UIGates. These provide the possibility to change a SalesPoint's FormSheet and MenuSheet. Thus UIGates enable the application programmer to use processes for user interaction.

Every gate has the method `getNextTransition()` with which the process that is in control of the gate can query the next transition to be executed. However, in a UIGate the return of a transition must often be deferred until the user has explicitly initiated an action which causes that transition to be returned. This is the reason why `getNextTransition()` consists mainly of a loop that is not left before the User performs an action which explicitly starts a transition.

The UIGate uses an integer flag `nChanged` which shows, whether a FormSheet, a MenuSheet or a transition has been set. On start of `getNextTransition()` the transition to be executed next is initialized with `null`. In the flag `nChanged` is saved that FormSheet and MenuSheet have changed. The reason is that there have been a FormSheet and a MenuSheet set on construction of the UIGate object. However, they have not yet been displayed on the SalesPoint. Marking both sheets as changed causes the following loop to execute code that actually displays the sheets.

Then a loop is entered and is executed as long as `nChanged` denotes that no transition has been set. If `nChanged` indicates that a FormSheet or MenuSheet has changed, the process context's methods `setFormSheet()` or `setMenuSheet()` are called and the sheets are set. Subsequently `nChanged` is set to `NOTHING`. The loop's body has been processed and could be executed again. This makes no sense, because it is unlikely that a user has caused a change of a transition, a FormSheet or a MenuSheet during one iteration of the loop. Most of the time the loop would run and find no changes. That is why the current thread is put to sleep.

The class UIGate contains the methods `setFormSheet()`, `setMenuSheet()` and `setNextTransition()`. They all work identically in principle. Fist the attribute to be changed - the FormSheet, the MenuSheet or the transition - is saved in an appropriate local variable, then `nChanged` is updated to indicate which attribute has changed. Then the thread which has been put to sleep is awakened.

As a result the loop in `getNextTransition()` is executed again. If a transition has been set, the loop is left immediately and the transition is returned, giving control to the process which can now execute its new transition thread. Otherwise the FormSheet or MenuSheet is updated as mentioned above and the loop thread is put to sleep again.

2.4 Function of Transitions

A transition's method `perform()` is the equivalent of `getNextTransition()` for Gates. It returns the next gate to the controlling process. Within `perform()` arbitrary code can be executed, but the developer should take care that no user interaction is implemented in a transition, because that would prevent the process from being suspended and the whole application could not be closed.

Chapter 3

Display Management

3.1 Introduction to Displays

Displays make it possible to display FormSheets and MenuSheets in different ways without knowing or explicitly supporting the different displays. A Display contains a FormSheet and a MenuSheet. How those are displayed is encapsulated in the individual Display. There might be need of a controlling instance.

The Framework contains three different types of Displays:

- JDisplayFrame: Each Display is displayed as an individual window.
- JTabDisplay: Each Display is shown as a tab. All tabs are are displayed in a shop window.
- InternalDisplay: Each Display is a window, comparable to JDisplayFrame. But other than with JDisplayFrame, these windows can not be moved over the whole screen, but only within the boundaries of the shop window (refer to JDesktopPane).

3.1.1 Connection of SalesPoints and Displays

Even though Displays are used to display FormSheets and MenuSheets of a SalesPoint, JDisplayFrame, JTabDisplay and JInternalDisplay are not dependent on SalesPoint, the Shop and the shop window. This makes it possible to use Displays, including FormSheets and MenuSheets, without SalesPoints.

Since communication between SalesPoint and the shop window is necessary, the classes DisplayFrame, TabbedFrame and DesktopFrame are defined within the class MultiFrame (the shop window) to extend the original Displays. These have a reference to their SalesPoints as well as their shop window and are able to manipulate each of them on demand. For further details refer to MultiWindow.

3.1.2 Correlation of FormSheetContainers and Displays

FormSheetContainers are objects that contain FormSheets and are able to display those. Status changes in the FormSheet, that are supposed to change the swing surface on the screen, are sent to the FormSheetContainer. That implies that the FormSheetContainer is responsible that those changes - if necessary - are displayed on the screen. FormSheetContainers decouple FormSheets and their Displays.

In the framework FormSheetContainers exist in combination with the Displays. In this combination the Displays have the responsibility for the ”user view”, that means they are displaying the swing surface on the screen. The FormSheetContainer is responsible for the ”framework view”, the accepting, processing and/or passing of events that relate to changes in the FormSheet. A Display and a FormSheetContainer represent a unit that performs jobs as described in the following paragraph.

This is the reason that the method `getDisplay()` of a FormSheet does not return a Display but a FormSheetContainer. The FormSheet itself uses the ”framework view”.

JDisplayFrame, JTabDisplay and JInternalDisplay implement the interface Display but not the FormSheetContainer. The FormSheetContainer is implemented as an inner class and saved as an attribute. This way was chosen since it causes less effort in serialization.

3.2 Title of Displays

Each of the displays JDisplayFrame, JTabDisplay and JInternalDisplay possess two titles, a primary title and a secondary title. The primary title is usually the name of the Salespoint, the secondary title is the name of the displayed FormSheet. For both `null` is allowed and as a result no title is displayed. Primary and secondary title are not required by the interface Display.

3.3 Setting of FormSheets

A FormSheet is assigned to a Display with `setFormSheet (FormSheet fs)`. In the case, that there is already a FormSheet assigned, the `cancel()` method of the already assigned FormSheet will be executed. The result is the execution of the display’s method `closeFormSheet (FormSheet fs)`. (refer to 3.4 Removal of FormSheets)

Afterwards the complete content of the ContentPane belonging to the Display is removed. In case the returned FormSheet is not `null`, the complete content is made visible on the Display:

- Setting the Secondary title according to the name of the FormSheet
- Attaching the FormSheetContainer to the new FormSheet
- Adding the FormSheet's component to the ContentPane of the Display
- Adding the FormSheet's button bar to the ContentPane of the Display
- In the case that another FormSheet was displayed before the new FormSheet was set, the Display is repainted. This ensures that everything is displayed properly.

Following this step the Listeners are informed that a new FormSheet is set.

In case `fs.getWaitResponse()` is `true`, the Java thread setting the FormSheet is stopped until a previously set FormSheet is closed with `closeFormSheet(FormSheet fs)`.

3.4 Removal of FormSheets

A FormSheet is removed from a Display using the method `closeFormSheet(FormSheet fs)`.

During the removal the following things happen. First, the current FormSheetContainer and the FormSheet are separated and the method `formSheetClosed` is called up. Within the method is defined what the FormSheet has to take care about while closing down. This method is a so called hook method, which can be overridden to change the standardprocedure of the Display while closing down. Usually `exitForm` is called, that makes the Display invincible and shuts it down afterwards.

Following the removal of the FormSheet all threads, that were blockaded are continued. In the end all Listeners are informed that the FormSheet was removed.

3.5 Display Size

Calling `setBounds (Rectangle r)` sets the position and size for JDisplayFrames and JInternalDisplays. The first two coordinates of the rectangle `r` represent the position and the last two the hight and the width of the window. Additionally the current position of the borderlines are saved.

In JTabDisplay the handed over rectangle is also saved. But in this case it is not possible to change the length and the width, since the size of a tab depends on the content.

3.6 Usable Displays

Displays have a `isUsableDisplay()` method which always returns `true`. The only exception is a class called NullDisplay, that is a dummy-display used for a background process. These background processes are not allowed to display MenuSheets or FormSheets.

Summary

This report explains in detail the functions of Data Management, Process Management and Display Management.

Catalogs keep their data by means of CatalogItems. Those CatalogItems can be added to, removed from or read out of Catalogs, additionally they can be editable, meaning that any changes applied to them can be rolled back. With DataBaskets, a transaction management is made possible which is comparable to transactions used within databases with respect to committing and rolling back data changes.

Interaction with SalesPoint applications is achieved by means of processes. Processes are an alternation of gates and transitions. Gates and transitions are implemented internally by the process through threads which work in a synchronized manner. User interaction should be allowed only at gates.

Displays are the interface between the software and the user. A Display contains a FormSheet and a MenuSheet. The Framework contains three different types of Displays: JDisplayFrame, JTabDisplay and JInternalDisplay. Since communication to SalesPoint is necessary, the classes DisplayFrame, TabbedFrame and DesktopFrame are defined to extend the original Displays.

The future of SalesPoint:

Every year questionnaires are given out where the students are encouraged to bring up criticism and ideas concerning the framework. Frequently mentioned issues are improved if possible. Through this it was discovered that changes in Storage Management (not covered by this report) and visual improvements are desired.

The standards and understanding of ”good” Software change through time. But since every year another group of students uses the framework, the standard and the functions of the framework are up to date and provide features which a majority of the programmers considers to be useful.