Property Tests User Guide

Contents

[Introduction 3](#_Toc393894175)

[1. Project Structure 4](#_Toc393894176)

[1.1 Packaging 4](#_Toc393894177)

[1.1.1 Business Layer 4](#_Toc393894178)

[1.1.2 Test Layer 5](#_Toc393894179)

[1.2 Maps – UI elements id, xpath 6](#_Toc393894180)

[1.3 Metadata 7](#_Toc393894181)

[1.4 Panel 7](#_Toc393894182)

[1.4.1 Edit panels 8](#_Toc393894183)

[1.4.2 Summary panels 8](#_Toc393894184)

[2. Element Types 9](#_Toc393894185)

[2.1 Base elements 9](#_Toc393894186)

[2.2 AssetList 10](#_Toc393894187)

[2.3 Composite elements 10](#_Toc393894188)

[2.4 Table 10](#_Toc393894189)

[3. Element Declaration 11](#_Toc393894190)

[3.1 Single Element declaration. 11](#_Toc393894191)

[3.2 AssetList declaration 11](#_Toc393894192)

[3.2.1 AppMetaData.xls 11](#_Toc393894193)

[3.2.2 Mapping between AppMetaData.xls and AssetList 13](#_Toc393894194)

[3.3 Composite Element declaration. 14](#_Toc393894195)

[3.4 Table declaration 14](#_Toc393894196)

[4. Dataset Structure. 15](#_Toc393894197)

[5. How to execute test 16](#_Toc393894198)

[5.1 Configuration 16](#_Toc393894199)

[5.2 Xml suits 17](#_Toc393894200)

[5.3 Test execution using suit file 19](#_Toc393894201)

[5.4 Test execution using testing.xml 19](#_Toc393894202)

[5.5 E2E scenarios execution 21](#_Toc393894203)

[6. How to create new test 22](#_Toc393894204)

[6.1 Class creation 22](#_Toc393894205)

[6.2 Dataset creation 23](#_Toc393894206)

[6.3 HSS test creation 28](#_Toc393894207)

[6.4 PUP test creation 30](#_Toc393894208)

[6.5 Results verification 32](#_Toc393894209)

[6.6 Database connector 34](#_Toc393894210)

[6.7 JobUtils class 34](#_Toc393894211)

[6.8 Policy data 36](#_Toc393894212)

[6.9 Dollar 37](#_Toc393894213)

[6.10 TimeSetterUtils and TimePointsHelper 38](#_Toc393894214)

[Appendix 1 42](#_Toc393894215)

[How to create more complex test script 42](#_Toc393894216)

[1. How to create policy using Policy data with different dataset names and two insureds 42](#_Toc393894217)

[2. How to make Billing transaction (write off transaction) 45](#_Toc393894218)

[3. How to check Billing transaction appeared in the transactions table 45](#_Toc393894219)

[4. How to make endorsement action 47](#_Toc393894220)

[5. How to fill only Applicant tab 47](#_Toc393894221)

[6. How to fill only one field from asset list 49](#_Toc393894222)

[7. How to verify policy status 49](#_Toc393894223)

# Introduction

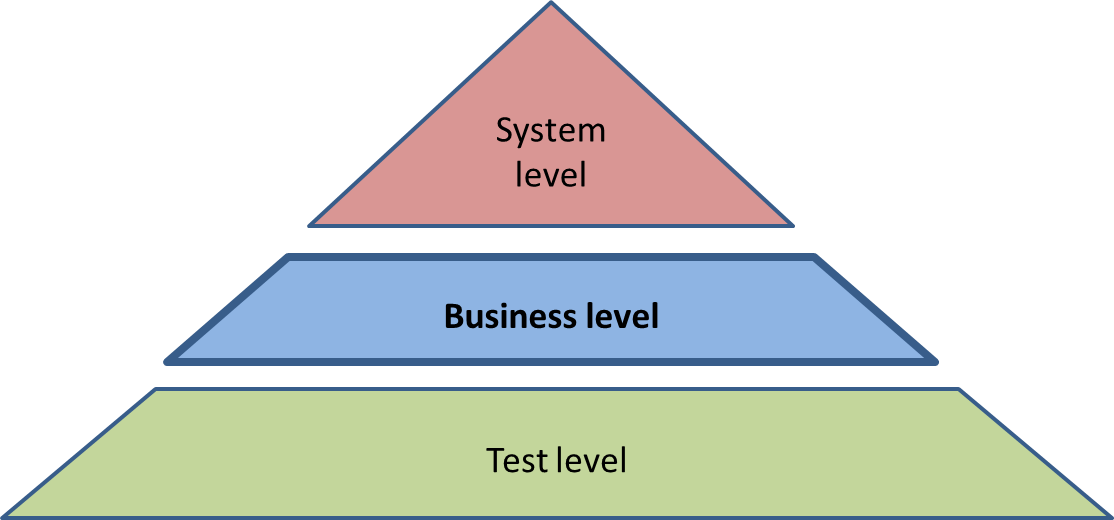
AAA Property automation framework is based on ISTF (Insurance Solutions Test Framework).

ISTF was designed especially for testing EIS based products, but could be used for other web applications too.

ISTF Core library suppors the following functionality:

* Selenium Controller and UI primitives
* AssetList functionality.
* DataProvider for XLS input data processing.
* Custom Asserts
* DB connectors
* Reporting
* Logging

Generally test automation project can be devided into 3 layers:



ISTF solution covers the only system level of the automation project.

Under the business term we mean:

* UI page mark-ups (maps, views)
* Test data (default data, data templates)
* Some kind of meta-information
* The instruments for automation tests implementation etc.

Business level is covered by AAA Property automation framework.

Test level is covered by AAA Property automation framework.

So the main topic of this guide will be Business and Test layers.

# Project Structure

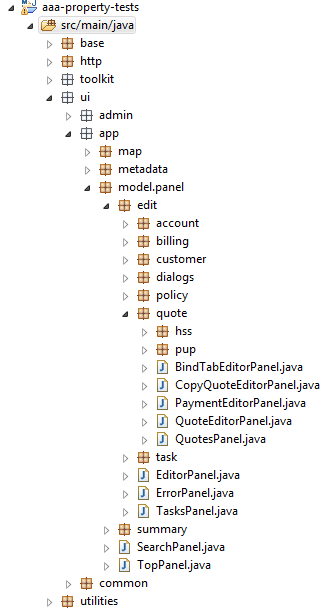
## Packaging

### Business Layer

All business and configuration entities are stored in *src/main/java* folder and have the following packaging structure:

* base – storage for all project configuration related classes
* http – storage for http related entities
* toolkit – storage for ISTF extended entities
* ui – UI based entities
  + admin – admin related entities
    - map – storage of locators for admin type
    - metadata – storage of constants, names for admin type
    - model.panel –panel management implementation
  + app – application related entities
    - map - storage of locators for app type
    - metadata – storage of constants, names for app type
    - model.panel – common panel management implementation
      * edit – edit panel management implementation (all panels that need to be filled)
      * summary – summary (consolidated) panels management implementation
  + common – common for both types entities
* utilities – storage of different utilities, that can be used in tests.

Screenshot:



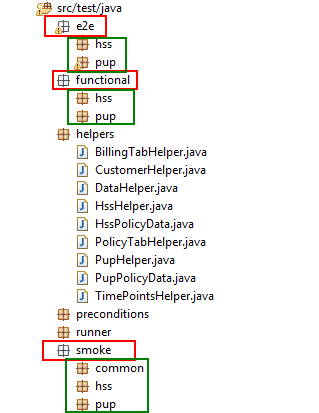
### Test Layer

All helpers and tests are stored in *src/main/test* folder. Tests are grouped according to their type and product.

Package *helpers* contains different useful features and functions:

* BillingTabHelper – is used for operations with tables and sections on Billing summary tab.
* CustomerHelper – in this class are implemented the most often used operations with customer (create, open)
* FileHelper – is used to perform the most often used operations with property file (store, get data)
* HssHelper - in this class are implemented the most often used operations with HSS product (create, open, copy, etc.)
* HssPolicyData – is used to create data instance for HSS policy.
* PolicyTabHelper – is used to perform the most often used operations with policy summary view.
* PupHelper - in this class are implemented the most often used operations with PUP product (create, open, copy, etc.)
* PupPolicyData - is used to create data instance for PUP policy.
* TimePointsHelper – is used for manipulations with time points and time shifts for e2e tests.

Screenshot:



## Maps – UI elements id, xpath

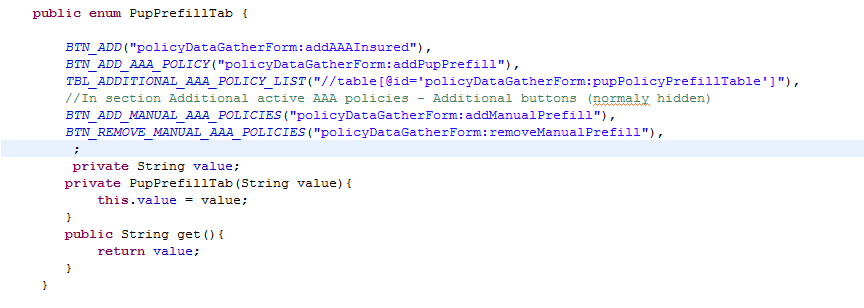
As mentioned above all UI maps stored in packages marked as *map*.

UI map it’s generally UI element locators’ container.

It’s organized as a customized enumeration to achieve

* Code unification and segregation
* Ability of processing tricky cases with dynamically generated IDs

Example:



**Important:** Map for Table and AssetList should be always represented as xpath.

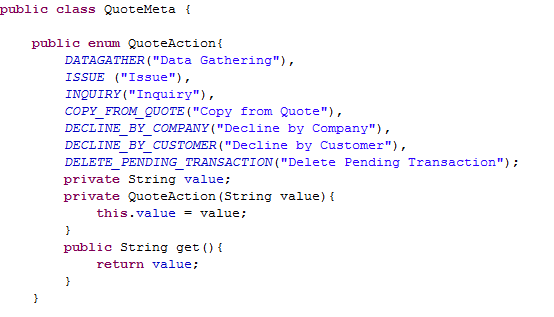
## Metadata

UI Metadata is stored in packages marked as *metadata.*

Under the metadata term we mean all constants (tab names, actions, etc.)

Metadata is also organized as a customized enumeration.

Example:



## Panel

Panel description is stored in packages marked as *model.panel*

Under the panel term we mean description of elements on web pages (elements declaration).

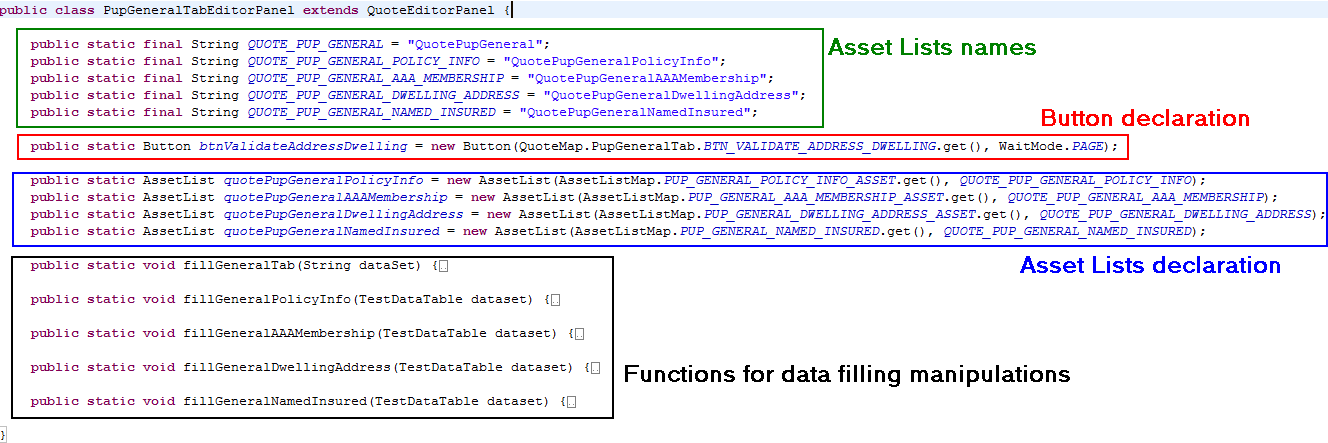
### Edit panels

Under the edit panel term we mean all tabs where could be entered some input data.

Edit Panel description is stored in packages marked as *model.panel.edit*

In these classes are stored elements declaration of the web page and a possibility to manipulate with data for input forms.

Example:



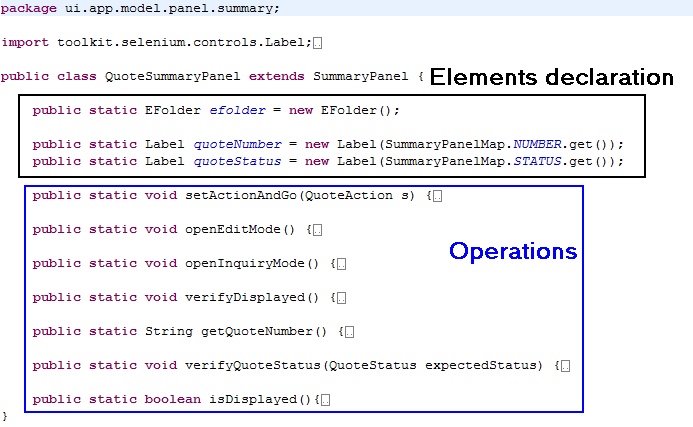
### Summary panels

Under the summary panel term we mean all summary (consolidated) view tabs.

Summary Panel description is stored in packages marked as *model.panel.summary*

In these classes are stored elements declaration of the web page and main operations that can be performed with them.

Example:



# Element Types

All UI elements have their own type and these types are implemented in ISTF core.

The main element types:

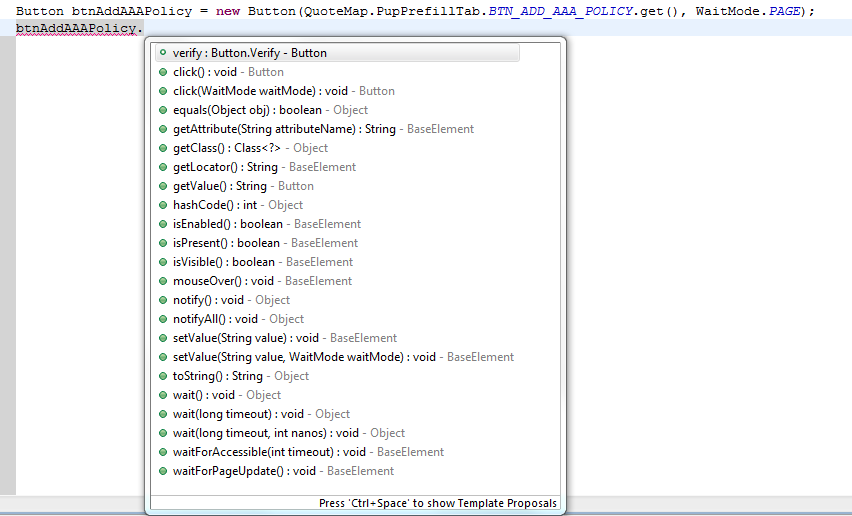
* Base element primitives
* AssetLists
* Customized composite elements.
* Table

## Base elements

Base element primitives:

* + Button
  + CheckBox
  + ComboBox
  + Label
  + Link
  + RadioButton
  + TextBox … etc.

Each element type have its own set of operations that can be performed with it.

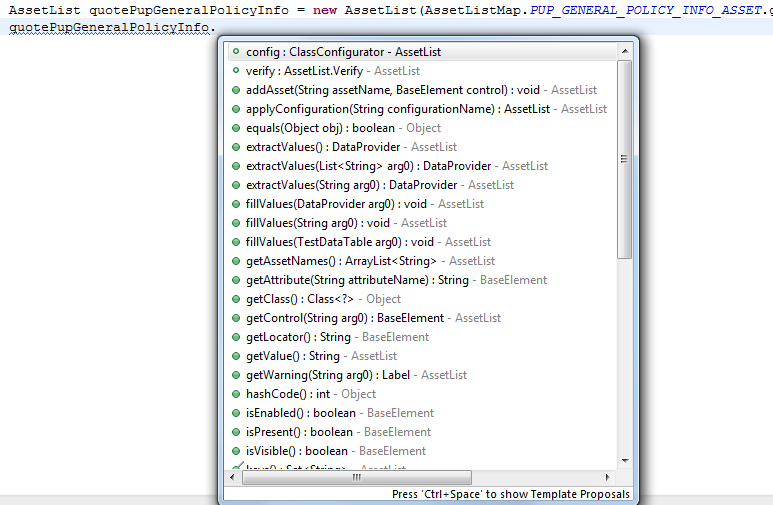
Example:

## AssetList

AssetList–elements on the page that can be grouped and have the same block parent. (Visit [wiki](https://wiki.exigenservices.com/display/CoEI/Asset+lists) for more information). Attention: one AssetList can’t include 2 elements with the same name (UI label).

AssetList has its own set of operations.

Example:



## Composite elements

Customized composite elements –complex elements, usually are composed from primitives (common panels/pop-up’s/etc.) Example: Yes/No pop-up, E-folder. For more information see package: *ui.common.model.composite*

## Table

Table – provides set of methods to manipulate with the table structures: Cell, Row, Column primitives.

# Element Declaration

Almost all elements and methods are static. This modificator permits to get access to the element without creating class instance.

## Single Element declaration.

All primitive elements can be declared and almost all of them have 3 constructors:

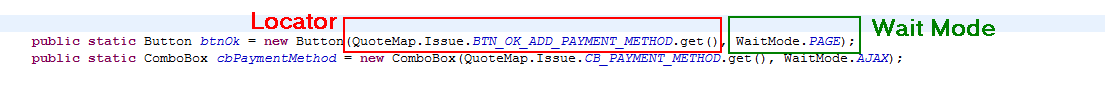
* + Button (locator, waitMode , timeout)
  + Button (locator, waitMode)
  + Button (locator)

*locator* – locator of the element on the page

*waitMode* – wait mode that should be called after click/edit on the element. Wait mode values: PAGE, AJAX, SLEEP, NONE. If not defined wait mode is defaulted to NONE.

*timeout* – timeout of the wait mode type. If not defined default one is used (can be found in config.properties file)

Example:



## AssetList declaration

## AppMetaData.xls

UI meta information – it’s an external storage of the input field attributes required for its localization on ui during data manipulation operations processing.

Having the standard EIS interface oriented to mass data processing we have a huge amount of fields invoked on the document flow. And trying to individually map all these fields – it’s a non-effective practice initially and further in terms of support.

Therefore instead of individual mapping all over the input fields the next scheme was implemented:

- The only holder element locator is mapped

- All the input fields under holder is described into external metadata (out of Java-code)

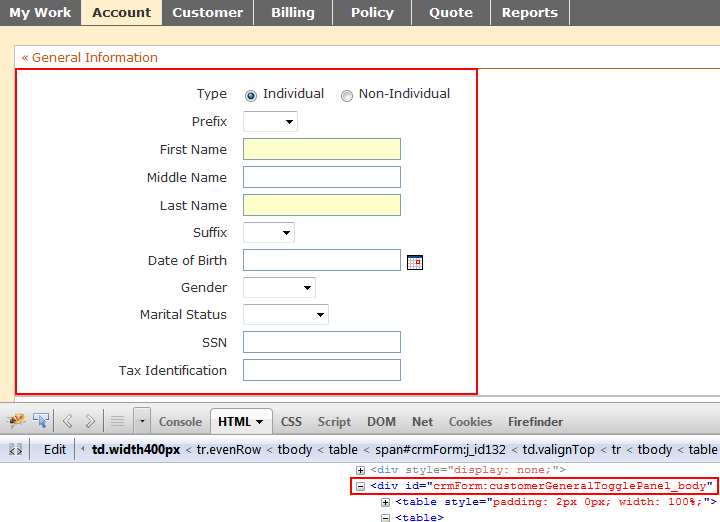
Example:

|  |  |
| --- | --- |
|  | |
|  | |
| **public** **enum** Search {  *SEARCH\_ASSET*("//form[@id='searchForm']/div/div[2]"), |  |

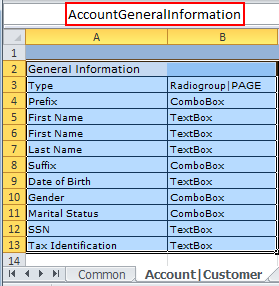
## Mapping between AppMetaData.xls and AssetList

**Constructor:** AssetList(String parentFormLocator, String nameOfAssetTypeRange)

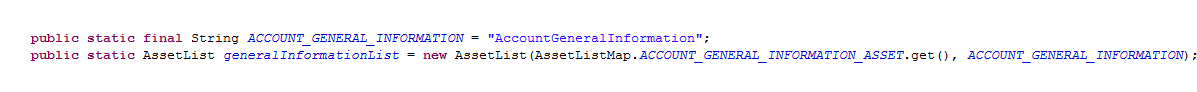
*parentFormLocator* – xpath of the block of elements.



*nameOfAssetTypeRange*– appropriate namespace in AppMetadata.xls



Example:



## Composite Element declaration.

Composite elements are located in package “ui.common.model.composite”

* YesNoDialog(String tableLocator) – composite element for operations with Yes/No pop-up dialog.

tableLocator – locator of the pop-up dialog

* Efolder() – Composite element for operations with e-folder.
* Note.getInstance() – composite for operations with Alert/Note.

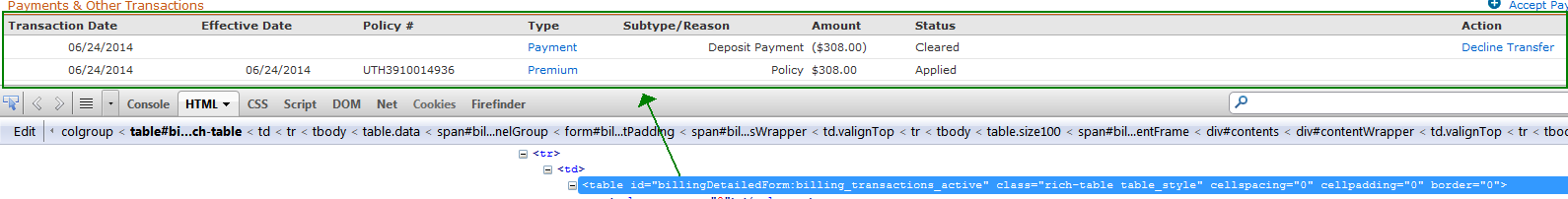
Example:



## Table declaration

**Constructor:** Table(String tableLocator).

*tableLocator –* xpath of table container.

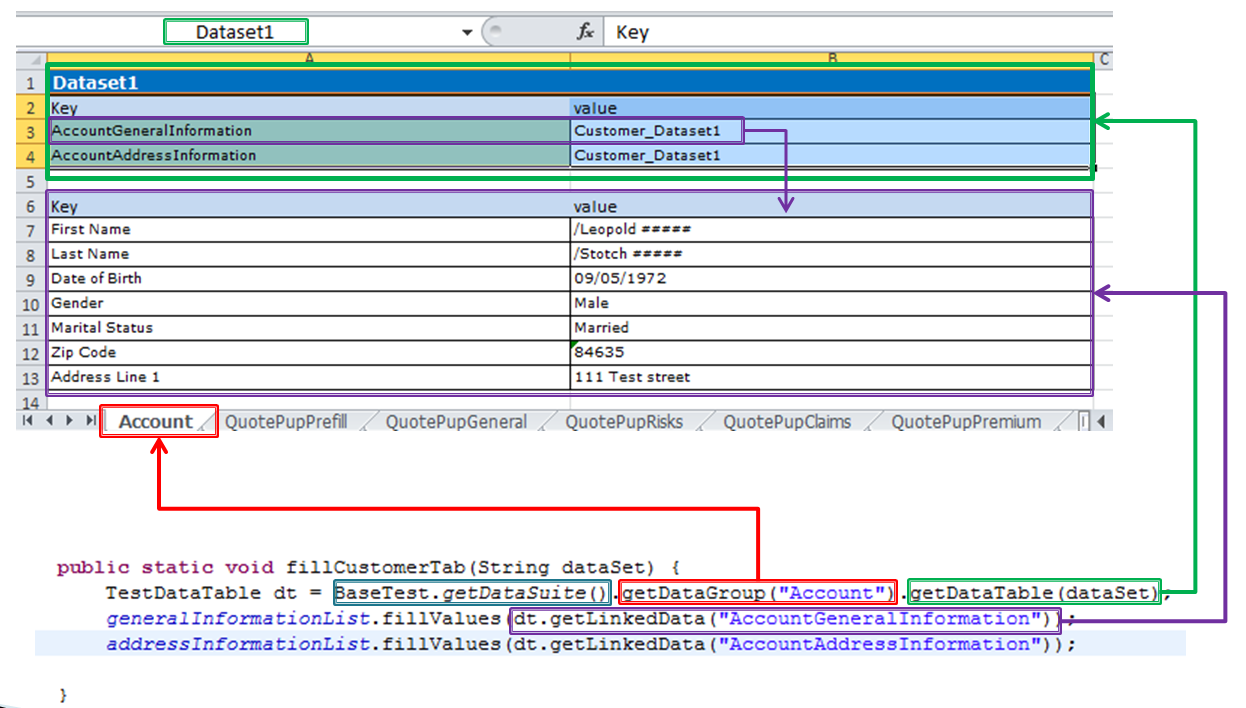


In this case xpath will be: *//table[@id=’* *billingDetailedForm:billing\_transactions\_active’]*

Example:



# Dataset Structure.

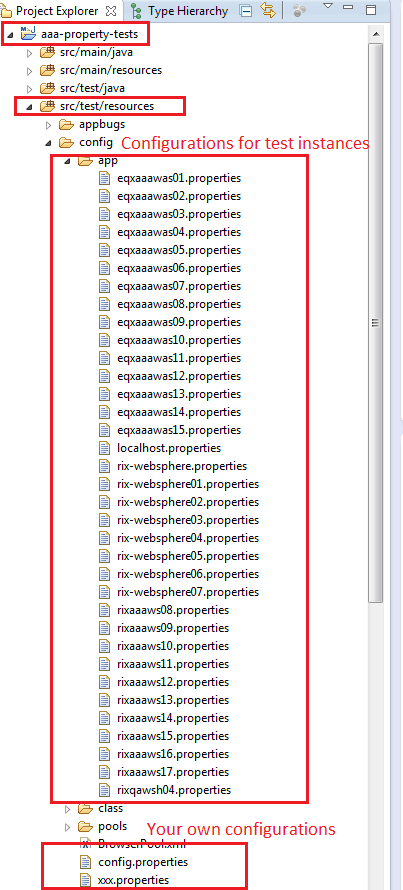


* BaseTest.getDataSuite() – returns excel file name for the current test, defined in suite parameter.
* getDataGroup("Account") – returns excel sheet. Almost each ui tab has its own sheet.
* getDataTable(dataSet) – returns table with dataset for the current test and current sheet.
* dataset – namespace of the dataset table for the current sheet.
* getLinkedData("AccountGeneralInformation") – returns test data table for the selected asset list for the current dataset.
* getValue(labelName) – returns specific value from the test data table for the selected asset list for the current dataset.
* labelName – label name of the element.

# How to execute test

## Configuration

To be able to execute tests you need to know what configuration to use. Below screenshots showing where all configuration files are stored. Off course it is possible to create your own and use them.

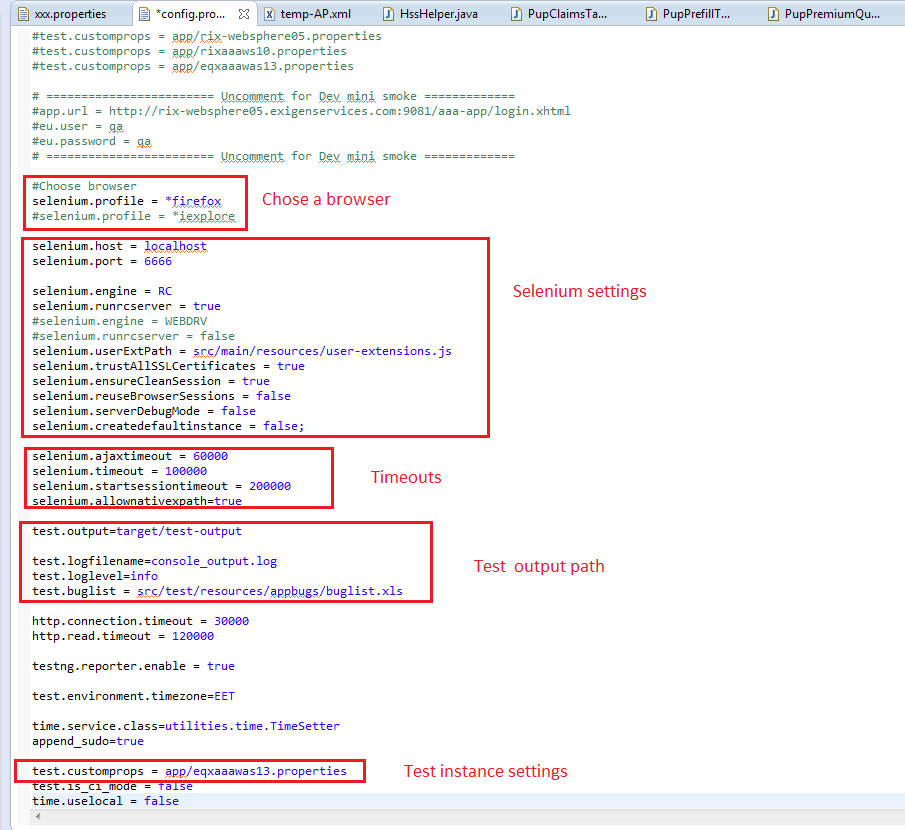


Let’s investigate a file named *config.properties*.

*Selenium.profile* parameter indicates what browser will be used during test execution. Usually it is *firefox.* This is proven to be the most stable test execution approach. Also Internet Explorer can be used. It is very useful for making some operations. There is possibility of switching between [*FireFox and IE*](#_HSS_test_creation).

*Selenium.engine* gives the possibility of switching between Selenium RC and Selenium Webdriver.

Wait times are considered as part of configuration and are measured in seconds. In *selenium.ajaxtimeout* property we specifythe maximum time our test will wait ajax request completion. In *selenium.timeout* property we specifythe maximum time our test will wait for new page loading.



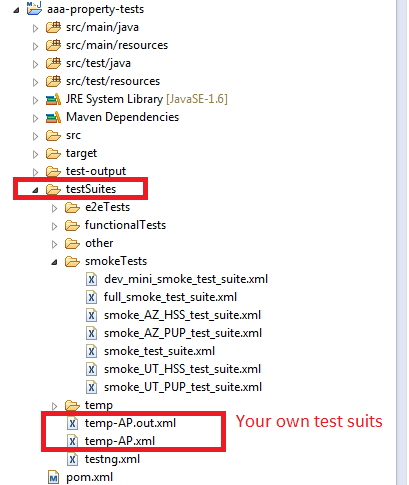
All paths for output and names of output files (log files, screenshots, bug lists) are written in test output section.

Path to instance settings is written in *test.customprops* property.

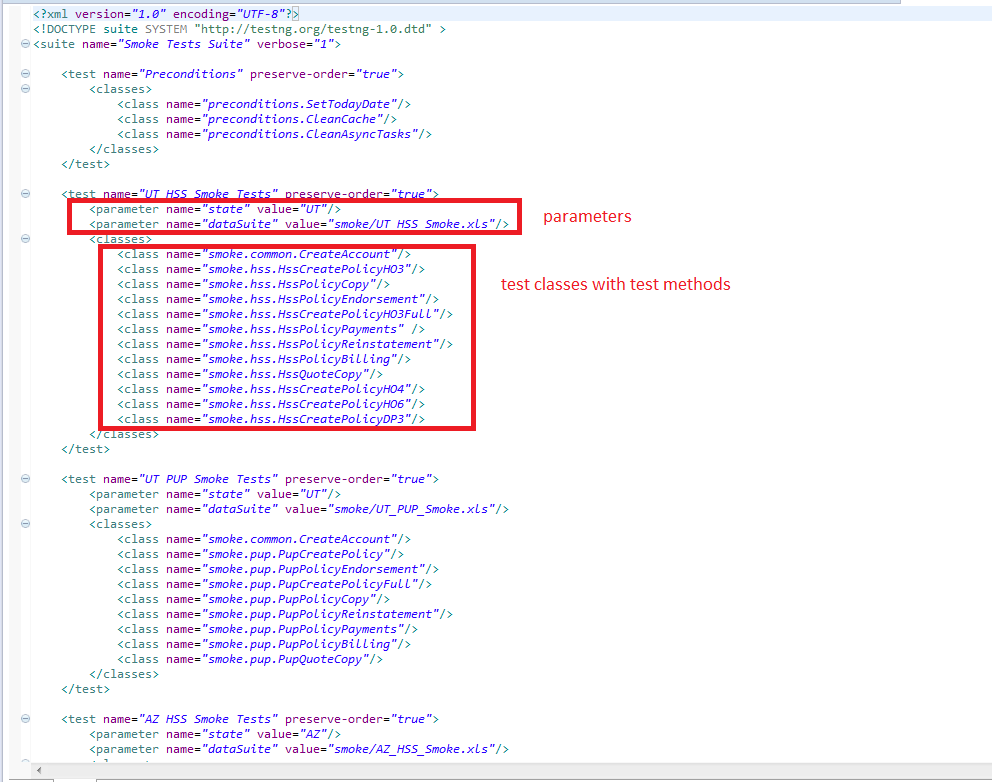
## Xml suits

If you want to execute tests you need to create test suits. There is variety of test suites created for different test execution approaches.

Located:



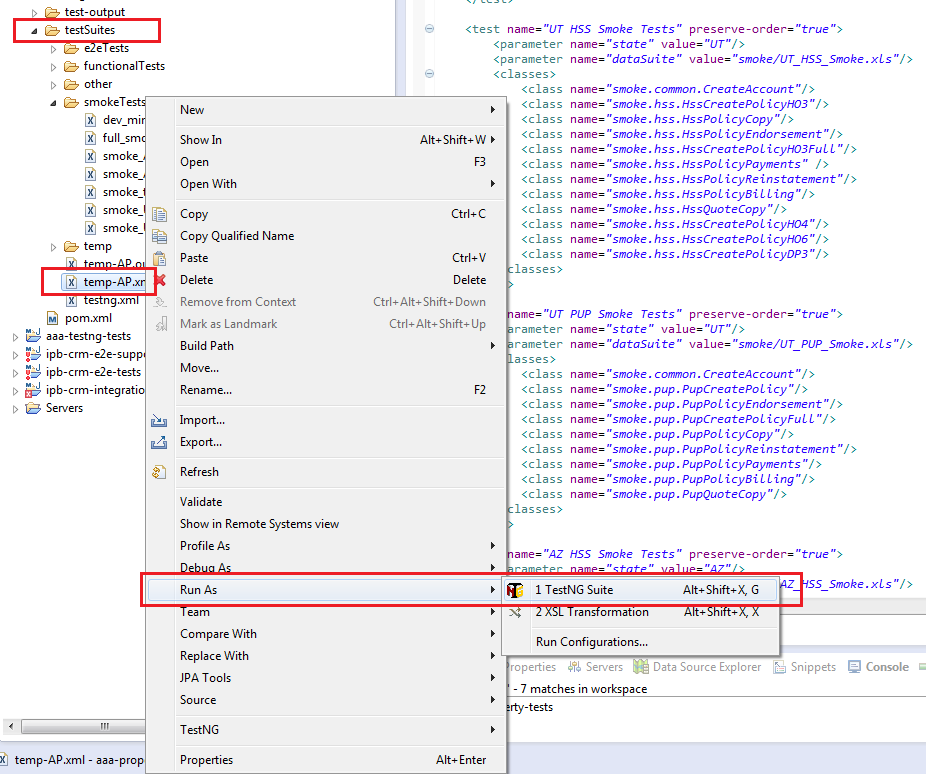
Below is opened *full\_smoke\_test\_suit.xml*



For every test suit we should specify *state* (the name of the state for which new quotes and policies will be created) and *dataSuite* (path to [test data](#_Dataset_creation)) parameters. After that list of test classes should be specified. Every test method from test class will be launched.

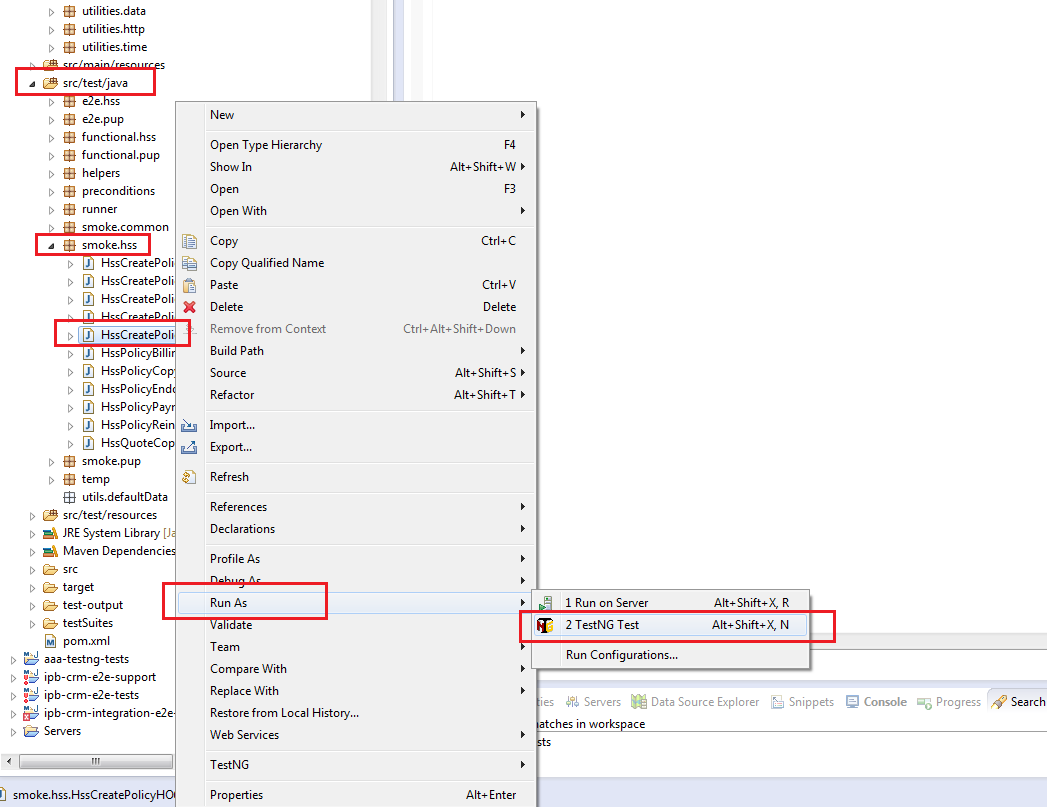
## Test execution using suit file

To run test using suit file you should do right mouse click on suit file *Run As->TestNG Suite.*

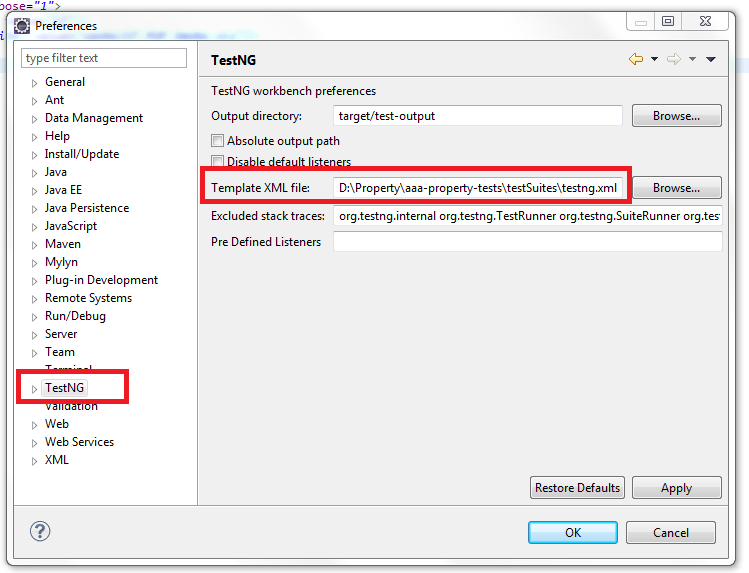
**

## Test execution using testing.xml

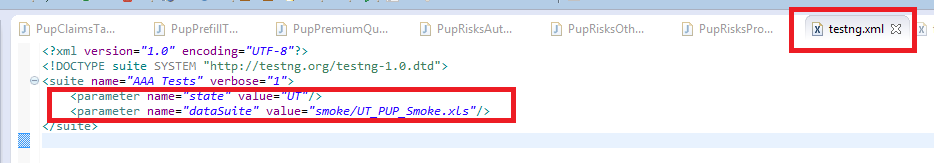
Test can be executed as single entity. To run test using *testing.xml* file you should do right mouse click on test class file *Run As->TestNG Suite.*

**

For using this method of test execution in TestNG settings should be specified path to *testing.xml* that is located in *testSuites* folder. For this open Preferences and change path.

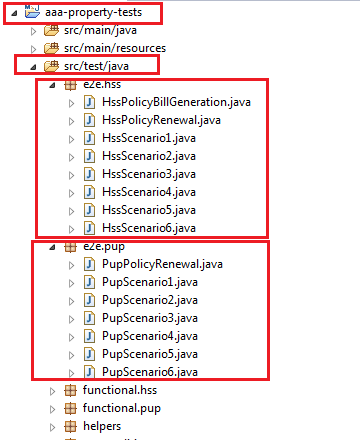


And now if you need to change some launch configuration you should edit testing.xml file.

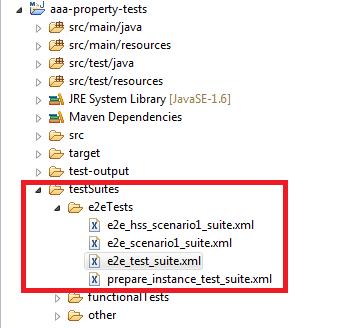


## E2E scenarios execution

End-to-end (E2E) testing is a technique used to test whether the flow of an application right from start to finish is behaving as expected. The purpose of performing end-to-end (E2E) testing is to identify system dependencies and to ensure that the data integrity is maintained between various system components and systems. E2E scenarios represent a scope of tests running one by one. E2E tests are located in *src/tests/java/e2e*.



Before launching E2E tests we should launch preconditions suite. E2E tests won’t work properly without it. E2E test suits and preconditions test suit are located in *testSuits/e2e Tests* folder.



For correct running of E2E scenarios:

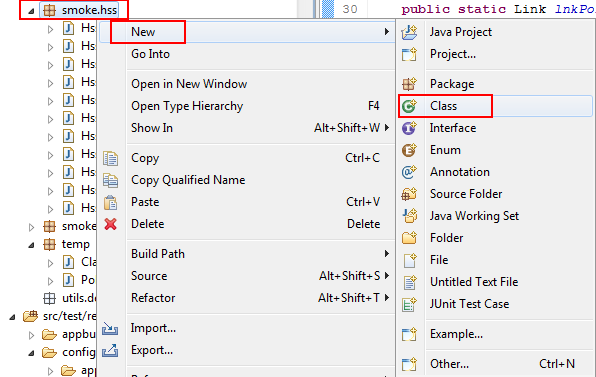
1. Run *prepare\_instance\_test\_suite.xml* (preconditions suite)as was described in paragraph 5.3.
2. Run *e2e\_test\_suite.xml* for launching all e2e tests or run definite e2e class as was described in paragraphs 5.3 and 5.4.

# How to create new test

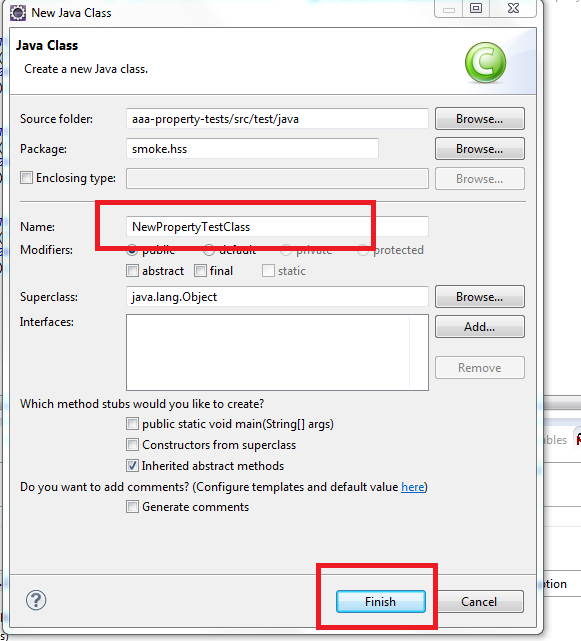
In this paragraph we will try to implement simple test and review some features.

## Class creation

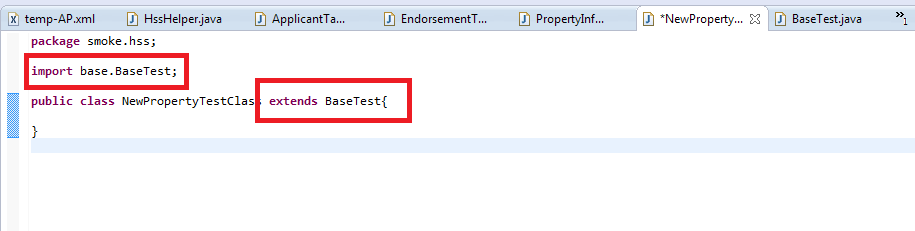
First of all it is necessary to create new java class in appropriate package (smoke.hss).



Mouse right click on necessary package. Select *New->Class*. Then enter class name and click Finish button.

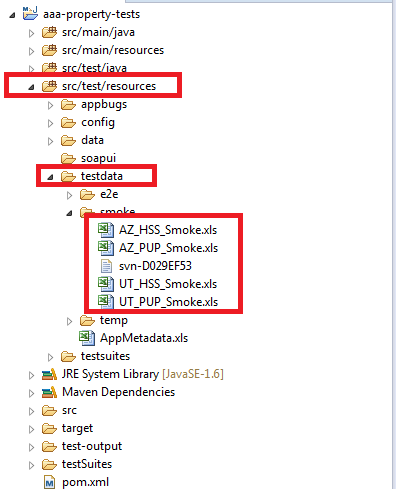


In order to have access to logger, configurations, base test settings, application instances newly created class should be inherited from *BaseTest* class.

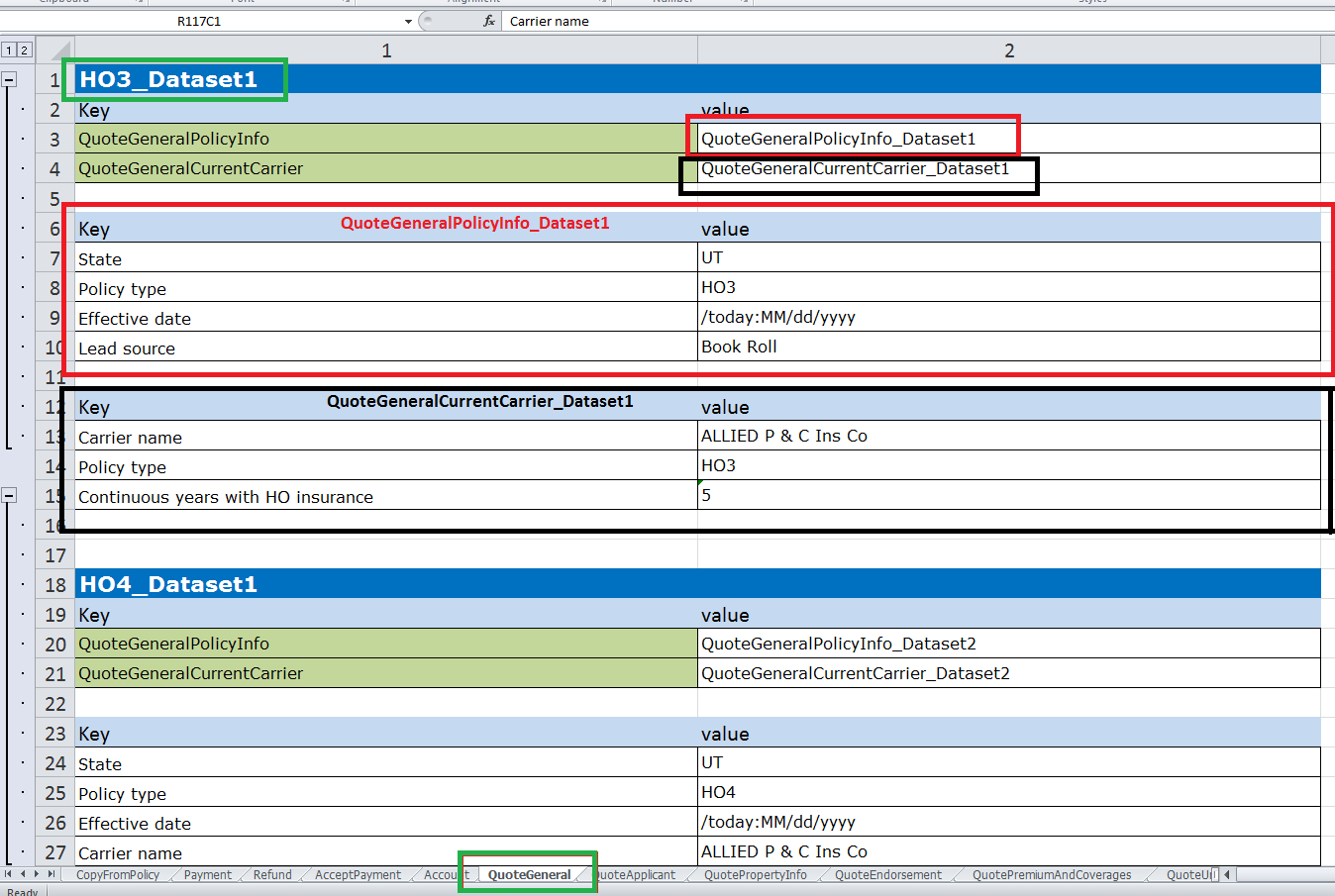


## Dataset creation

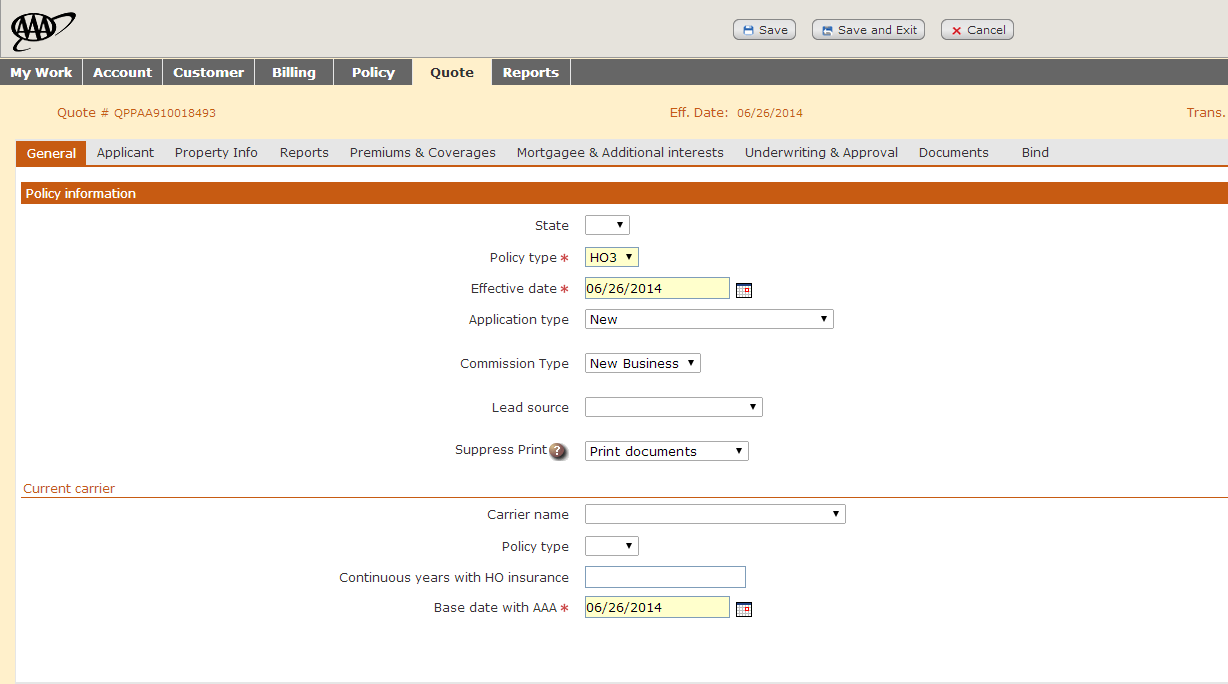
To avoid test data description in test scripts was decided to use datasets. Dataset is a named area in xls file. Those files are located in *src/test/resources/testdata* folder.



Let’s take a look at dataset for HO3 policy creation. For this open *UT\_HSS\_Smoke.xls* and open *QuoteGeneral* sheet. At this sheet you can find *HO3\_Dataset1*.



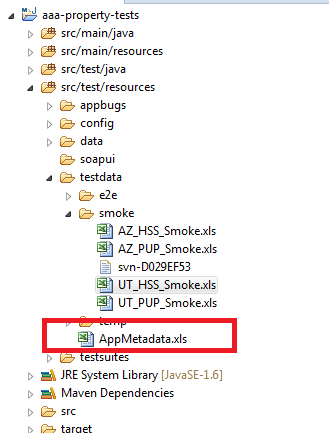
This dataset is used for filling data on General tab during HSS quote creation.

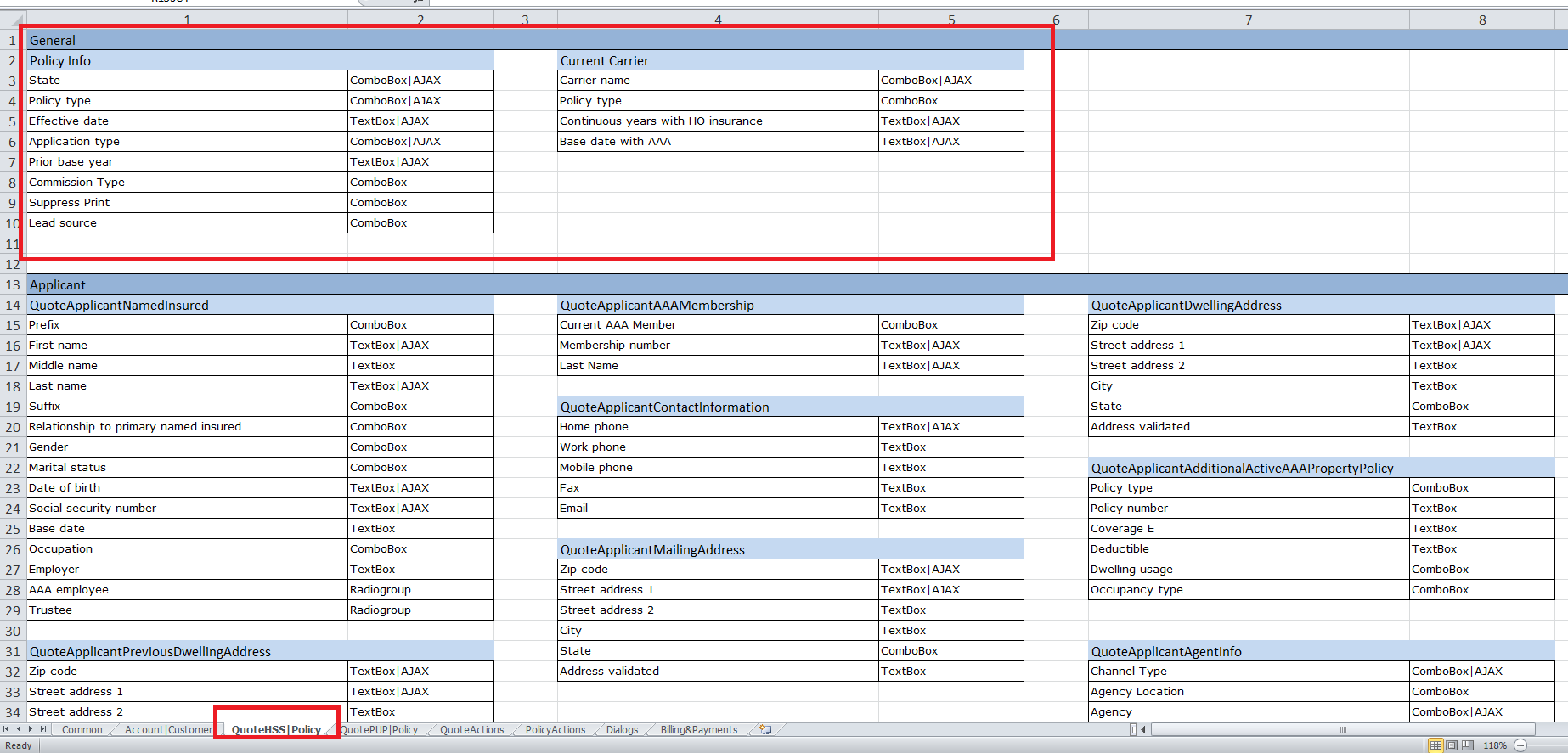


As you can see all tables consist of two columns Key and Value. Names of the filling areas are written in Key column in accordance with their names in application. For example: *QuoteGeneralPolicyInfo* and *QuoteGeneralCurrentCarrier.* Names of data (table) in xls file that will be used for filling fields in application are written in column Value. For example: *QuoteGeneralPolicyInfo\_Dataset1* and *QuoteGeneralCurrentCarrier\_Dataset1.*

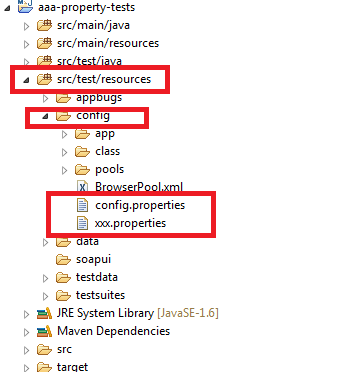
This datasets also consist of Key and Value columns. Key column represents a set of controls (text areas, textboxes, comboboxes, radio buttons and etc.). Value column contains data for filling this controls. For example: In combobox *State* should be set value *UT*.

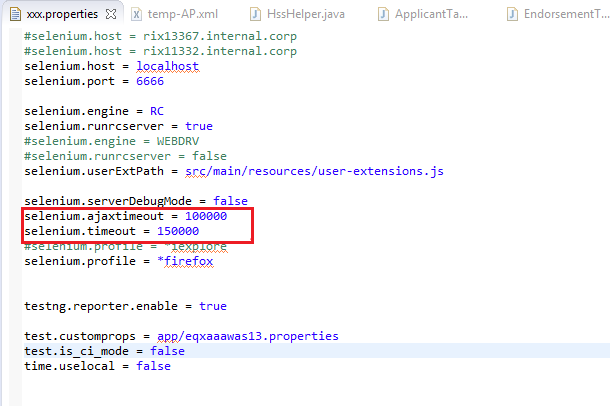
All controls are described in *AppMetadata.xls* file.



If we need to find out information about controls on General tab for HSS quote we should open AppMetadata.xls file, open *QuoteHSS|Policy* sheet and find area named *General*.

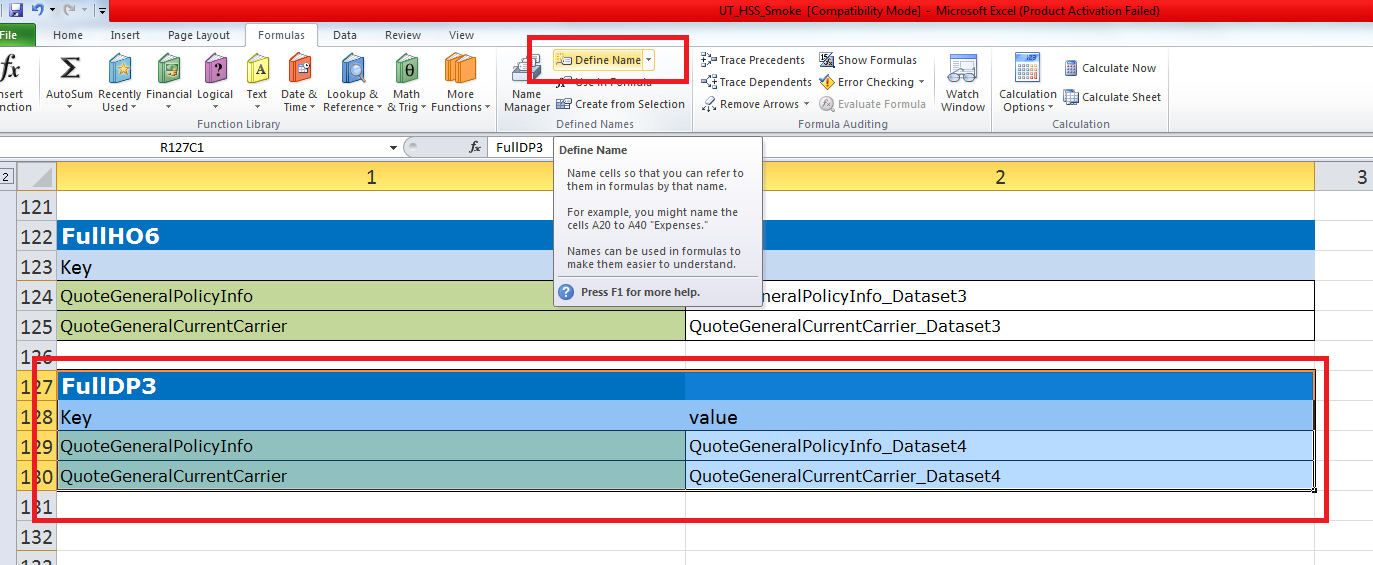
Here you can find description of all the controls on the General tab. Description contains element type and wait mode type (AJAX, PAGE, NONE). If there is no wait mode it means that there is no timeout before filling next control. Timeouts for AJAX and PAGE are set in milliseconds in configuration files. Those files are located in *src/test/resources/config* folder.



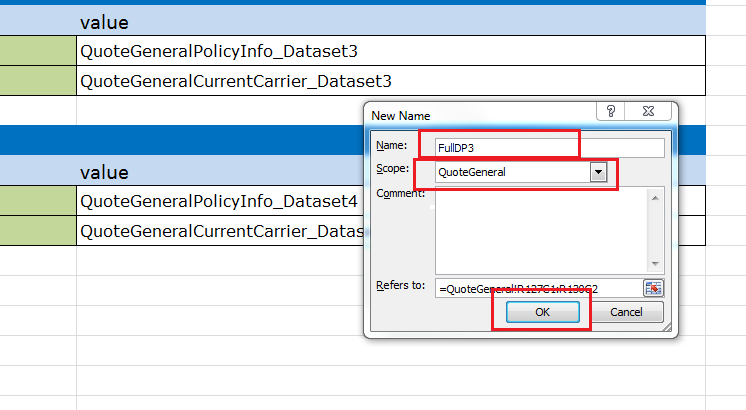


On *QuoteHSS|Policy* sheet you also can find description of controls for other quote creation tabs (Aplicant, Property Info, etc.).

For creating dataset table you should open xls file on needed page. Than create table consist of two columns Key and Value. After that highlight this table and click Define name button.



Enter table name in Name field, select sheet name in Scope combobox and click Ok button. And that’s it, new dataset is created.



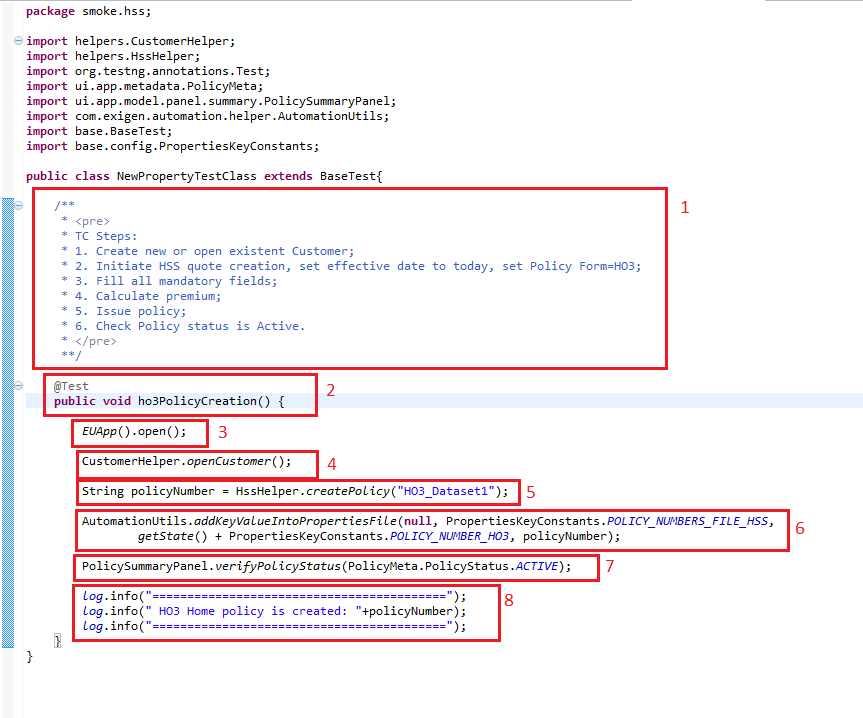
## HSS test creation

We don’t use ISTF functions during tests creation. In test methods we use pre-implemented business functions of Property framework.

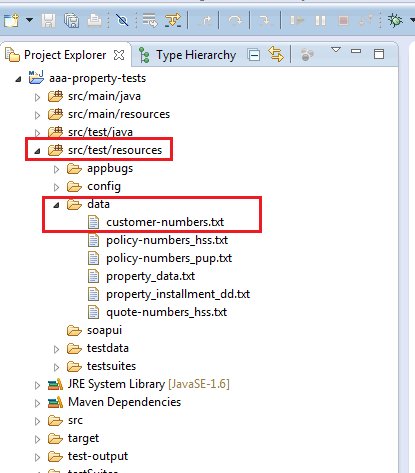
Here is the example of test checking HO3 policy creation.

Steps to implement:

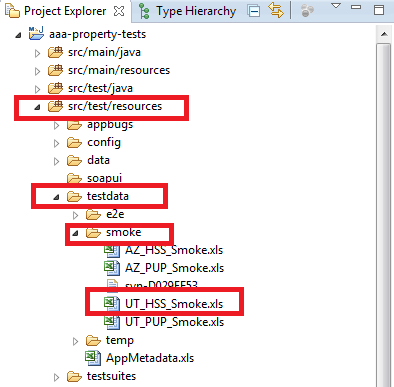
1. Login to app
2. Open existent Customer or create new one
3. Create HSS policy (fill in all tabs)
4. Verify Policy is created and is in status Active



1. Test description contains TC steps implemented in test method.
2. In order the usual method become a testing method you need to add *@Test* annotation.
3. Every test scenario starts from opening application. If it is user application *EUApp().open()* should be used. If it is admin application *ADApp().open()* should be used. According to default settings application will be opened in Mozilla Firefox. If you need to use IE use *IEApp().open().*
4. After application is opened we need to create a Customer. For this we use *CustomerHelper.openCustomer().* This function opens already created customer or creates a new one. If customer has already been created its number has been saved in customer-numbers.txt file.



1. In case Customer was created you can create HO3 policy. For this you should use *HssHelper.createPolicy("HO3\_Dataset1").* "HO3\_Dataset1" is a name of datasets in test data file UT\_HSS\_Smoke.xls



1. After policy was created you can write it into data file policy-numbers\_hss.txt and it can be used in other tests. In this reason *AutomationUtils.addKeyValueIntoPropertiesFile* function should be used.
2. Newly created policy should be in Active status. For checking this *PolicySummaryPanel.verifyPolicyStatus(PolicyMeta.PolicyStatus.ACTIVE)* need to be used.
3. For making logs in console should be used *log.info().*

## PUP test creation

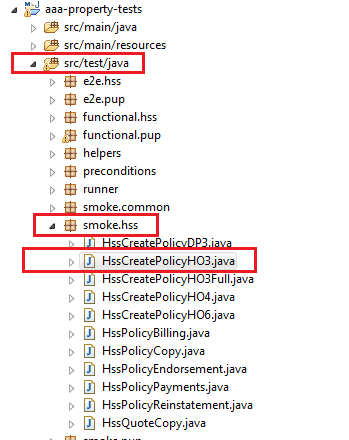
This is an example of how to create PUP policy.

Steps to implement:

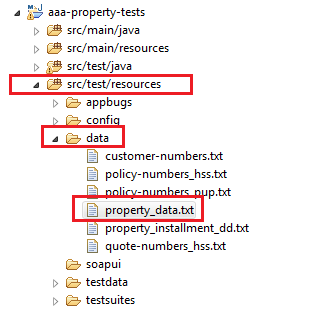
1. Login to app
2. Open existent Customer or create new one
3. Create HSS policy (fill in all tabs)
4. Verify Policy is created and is in status Active

**Important:** Creating policy for PUP product depends on HSS policy creation:

1. Before start PUP policy creation run *HssCreatePolicyHO3* test (how to run tests was described in paragraph 5).

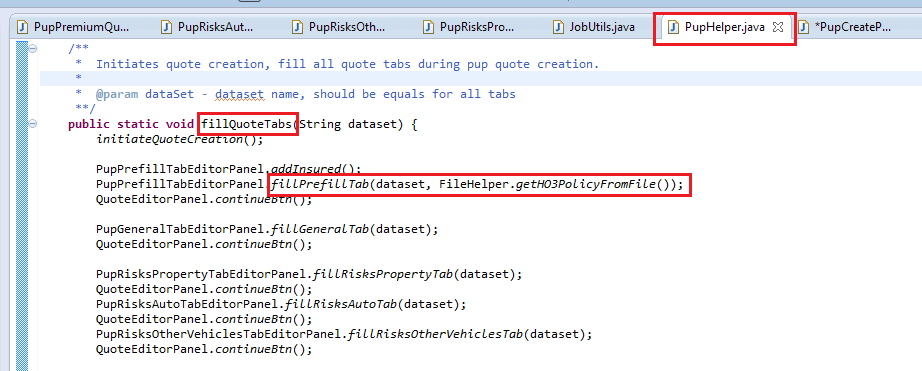


And check that new HO3 policy number was written in *property\_data.txt* file that is located in *src/test/resources/data* folder.

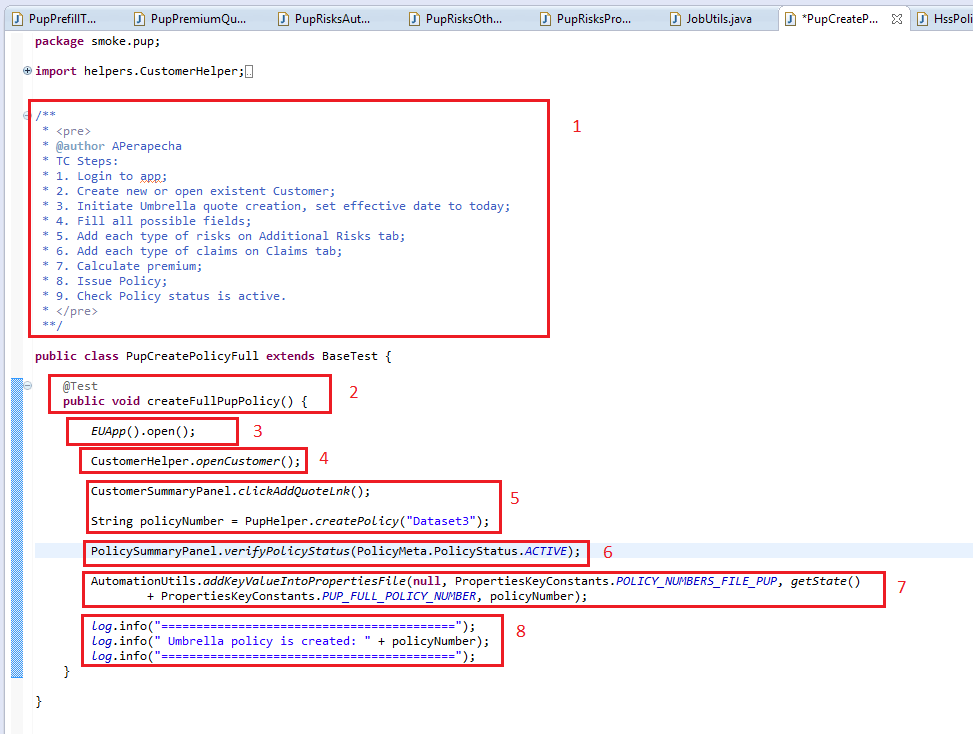


http://i.gyazo.com/a5f7a914663415cfb27ff70552a42708.png

This HSS policy number will be used for PUP quote prefilling in *PupHelper.fillQuoteTabs()* method.



1. Further the creation of PUP test is similar to creation test for HSS product ([paragraph 6.3](#_HSS_test_creation)).

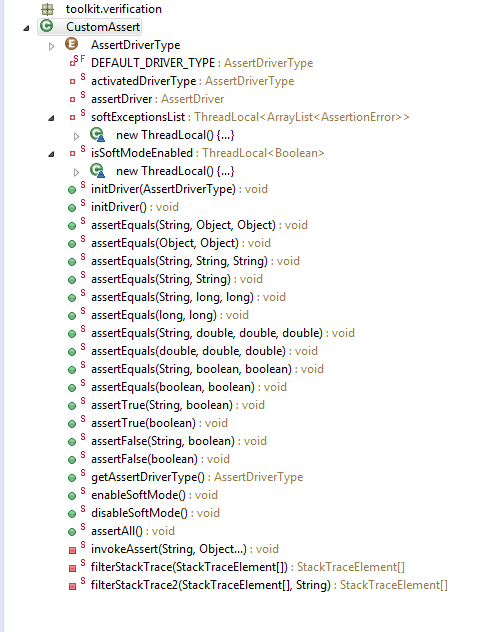


## Results verification

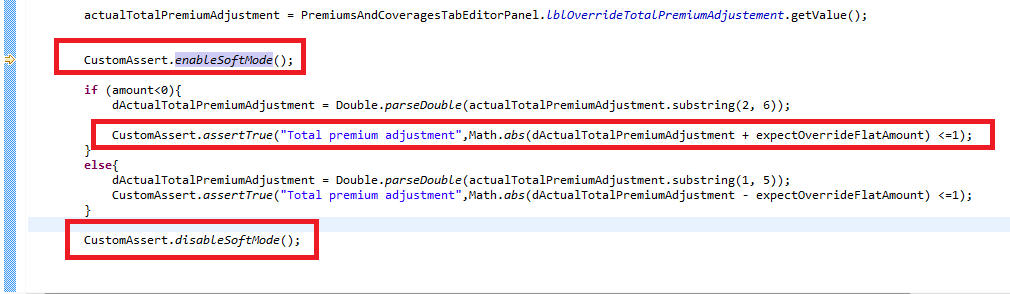
There are three ways for results verification:

1. Custom Assert
2. Verify() function of controls.
3. Special verification functions.
   * 1. Custom Assert

CustomAssert is a special design that allows test assumptions about the values ​​of arbitrary data in an arbitrary point in the program. Here is the list of available functions.



Here is the example of usage.

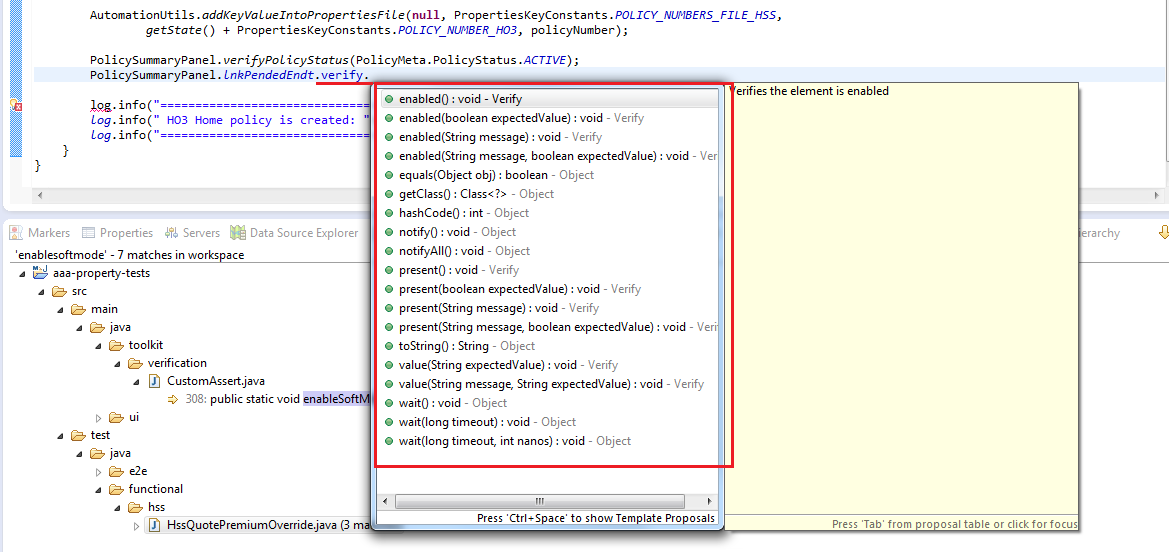


If you need to verify some values, but test must go on even if verification failed, you should use *CustomAssert.enableSoftMode()*. This function allows collect verification errors without stopping test run. To stop *enableSoftMode()* use *CustomAssert.disableSoftMode().*

For checking that condition is true use *CustomAssert.assertTrue().*

* + 1. Verify control

Every control has special function *verify* that can check if control presents, enabled, its value and etc. Here is a list of available features.



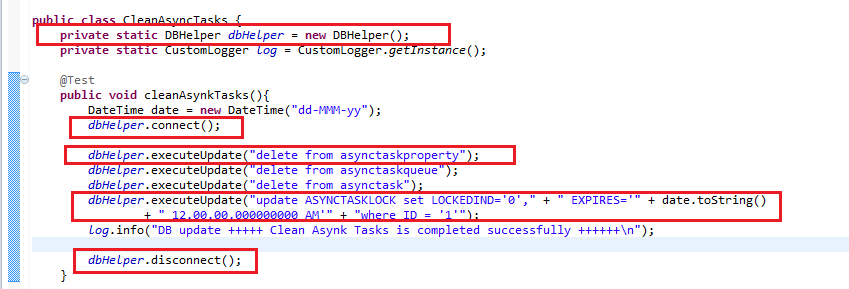
* + 1. Special functions

Some Property Framework Panels contain its own verification functions for different controls. It is done for easier usage and intuitive tests writing.



## Database connector

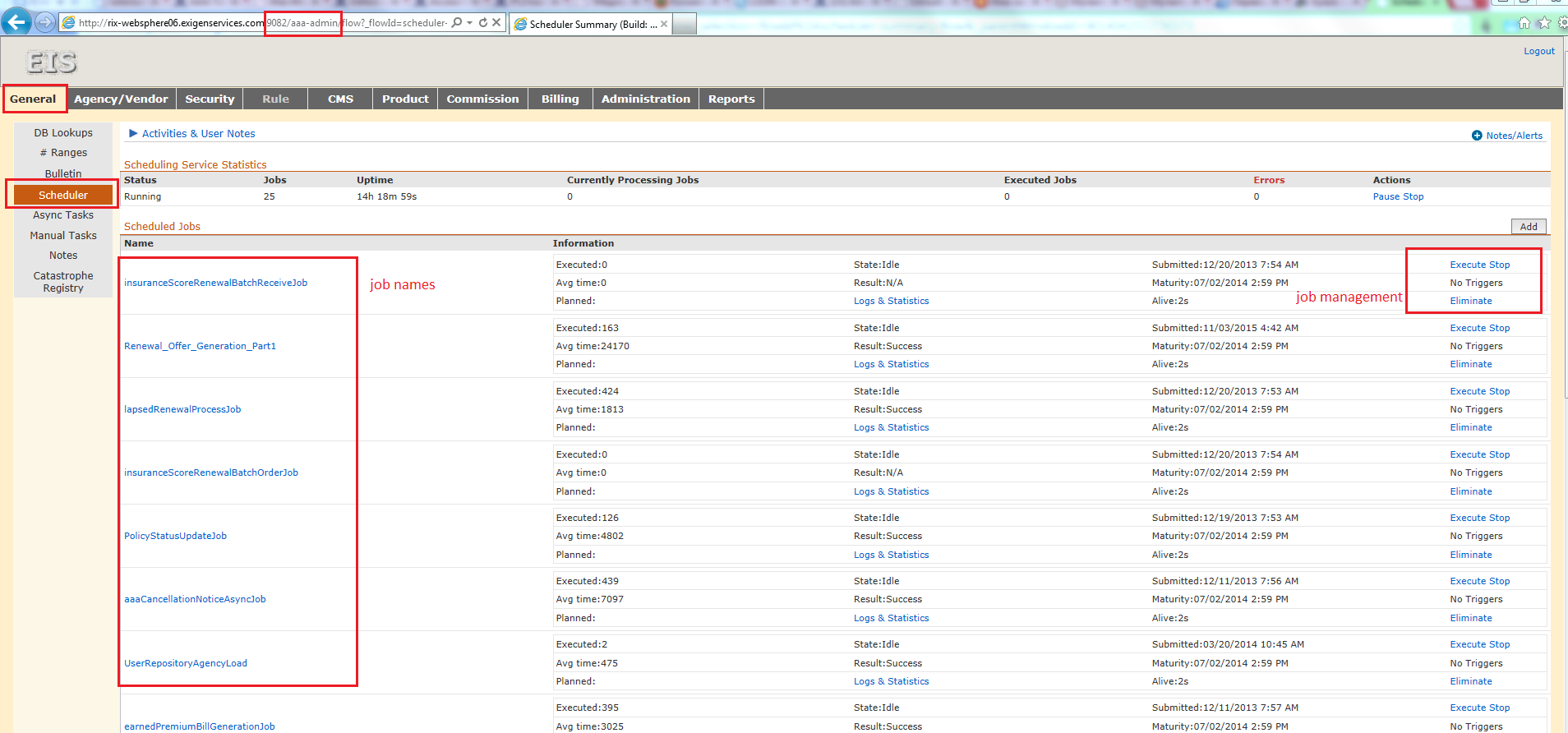
For using application database *DBHelper* class was implemented. It allows to connect to database and to make some changes you need. Here is an example.



For using DBHelper at first you need to create DBHelper object. Than you should connect to database using *connect()* method. After that you can sent sql queries to database using *executeUpdate(String sqlQuery)*. After you finished your work with database close connection – *disconnect().*

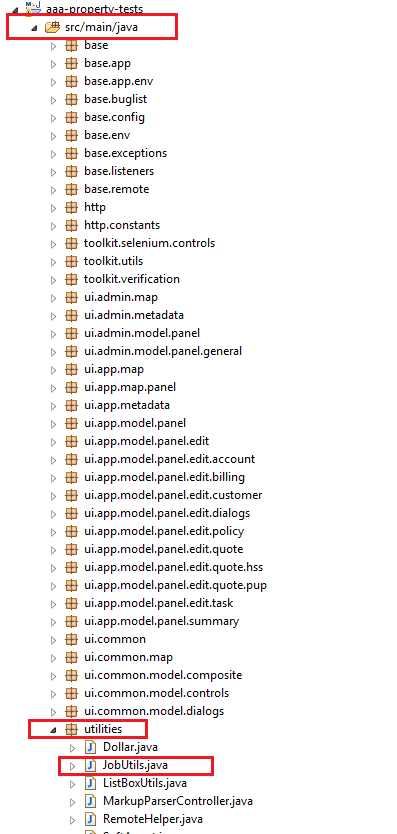
## JobUtils class

For making some automatic payments, perform some operations with insurance policies we need to use jobs. Jobs are located in admin application on *General* tab in *Scheduler* section.



Click *Execute* link to start the job. Click *Stop* link to stop job execution and click *Eliminate* link to delete a job.

There is a JobUtils class in Property framework for job execution. It is located in *src/main/java/utilities* folder.

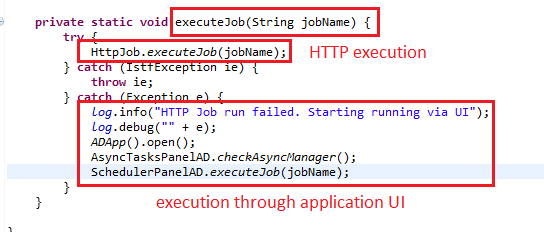


For job execution use:

JobUtils.*executeJob*(Jobs.*billingInvoiceAsyncTaskJob*);

Write down job name into the brackets.

Method *executeJob()* is divided on two parts. At first we try to execute job using HTTP. If it fails than we try to execute job using application UI.



## Policy data

In HssHelper and PupHelper we have 2 ways of policy creation methods. Directly through the Datasets when we set string (Dataset name) as a parameter. The second way is using PolicyData objects as parameter.

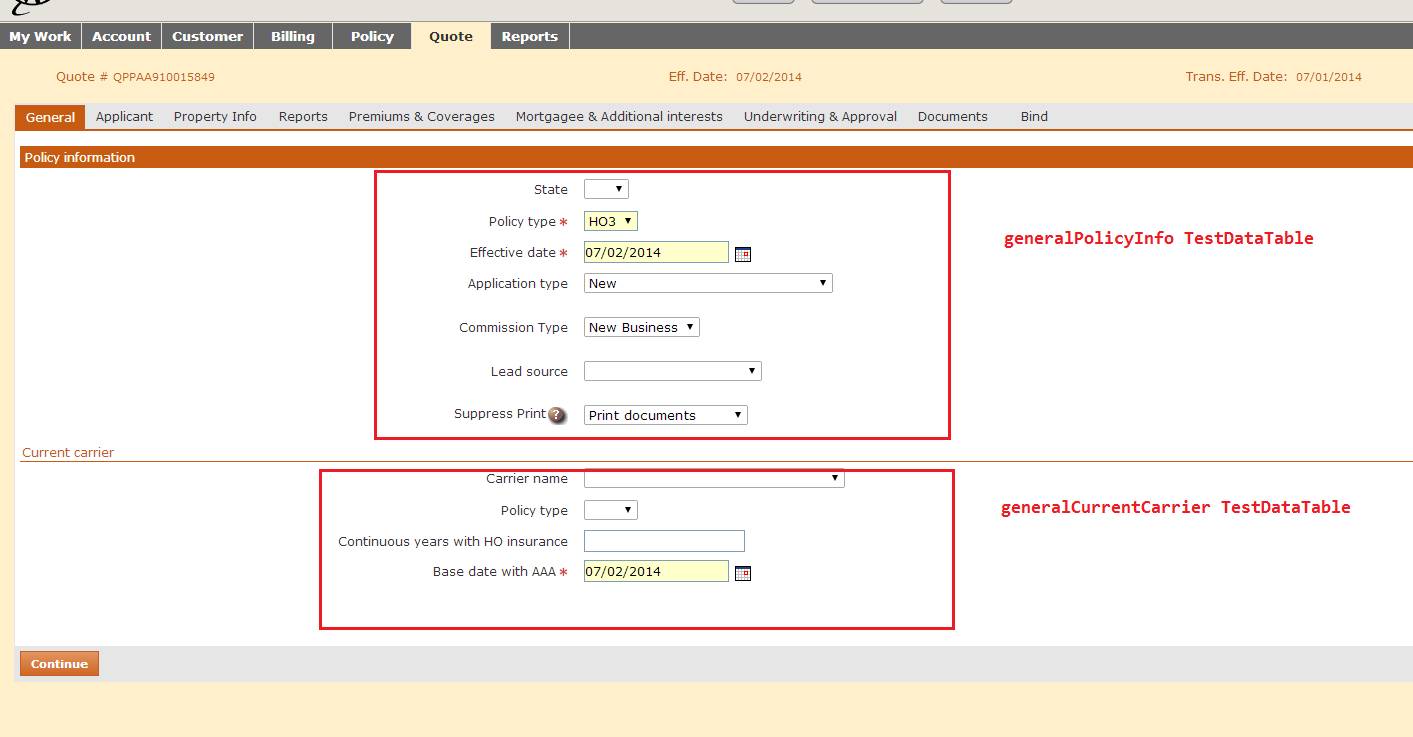
When we use Datasets every time we create new policy or quote with new data in test we should create new Dataset for every tab of this policy.

But if you need to make changes on one tab (or more) and leave all other data without changes you can use PolicyData. You can use old Datasets for all tabs and create new for needed tabs.

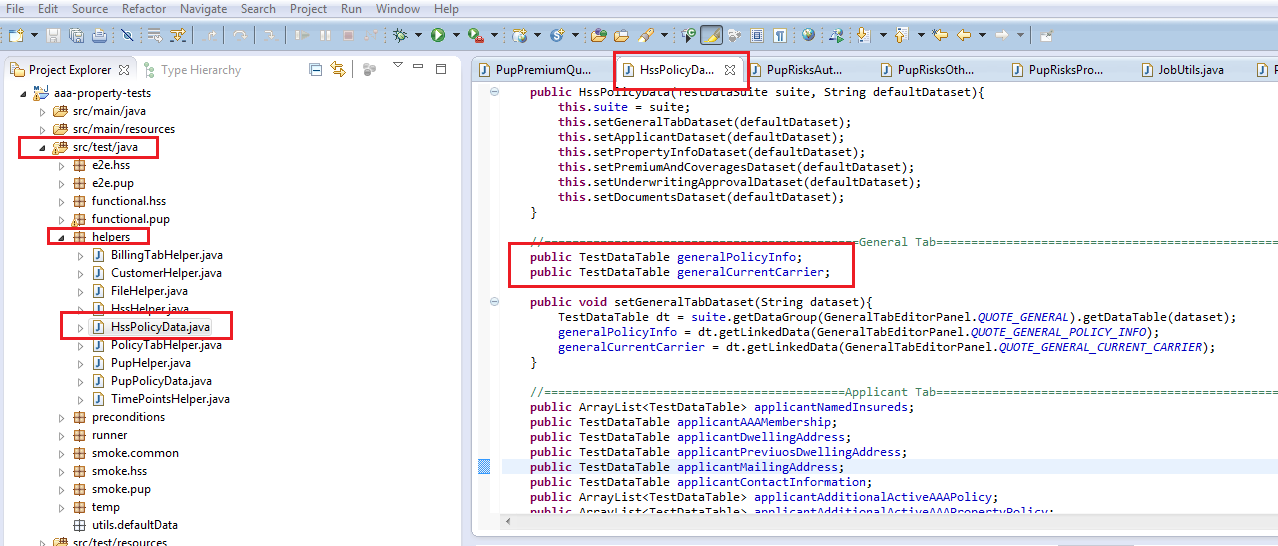
There are two PolicyData classes *HssPolicyData* and *PupPolicyData* (for Hss and Pup products). Those classes are located in *src/test/java/helpers* folder.

PolicyData consists of TestDataTables. Each of TestDataTables is a group of elements on the page that have the same block parent.

In application:

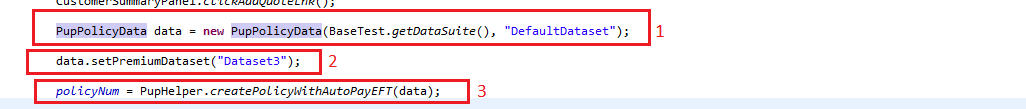


In HssPolicyData:



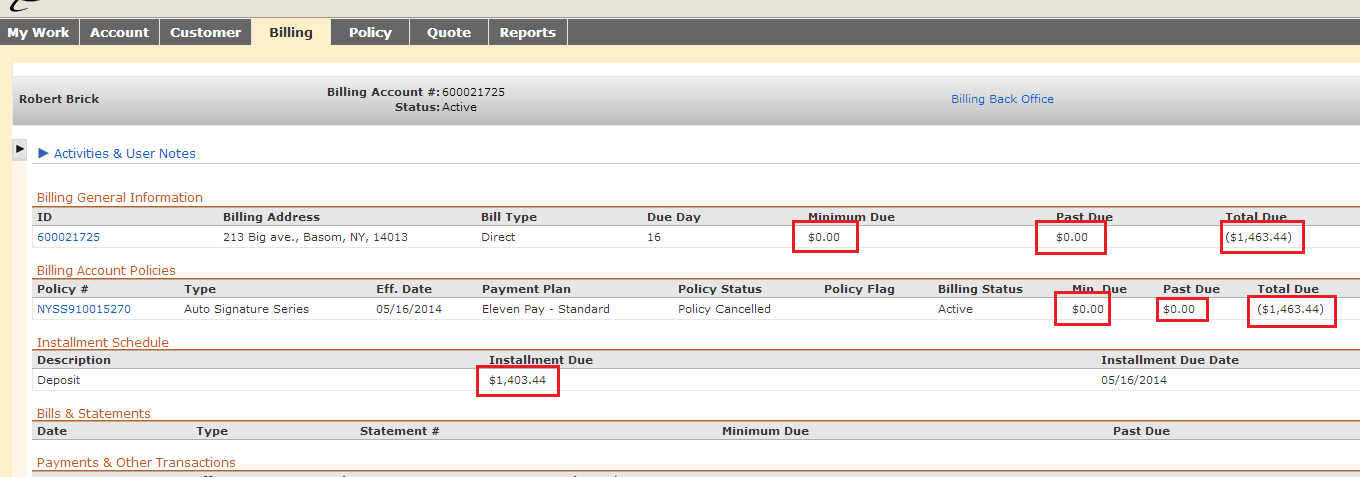
Let’s analyze an example: you need to create new policy and you can use already created Datasets named “DefaultDataset”. By default all tabs will be filled by data named “DefaultDataset”. But you need to fill Premiums and Coverages tab with other data. You create new Dataset in test data file named “Dataset3”. And now you can use PolicyData:

1. Create new PolicyData object and define default dataset name.
2. Change dataset names for tabs you need to be filled in another way.
3. Create new policy using PolicyData.

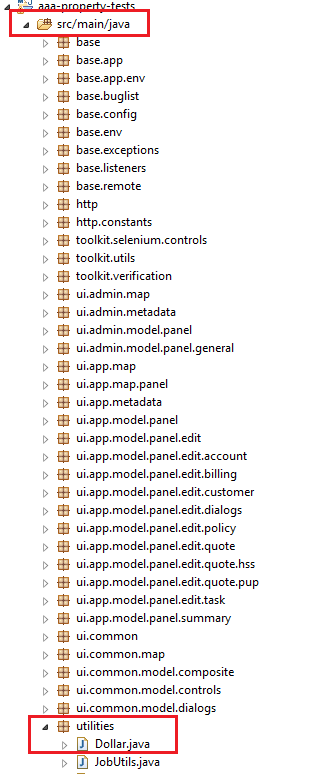


## Dollar

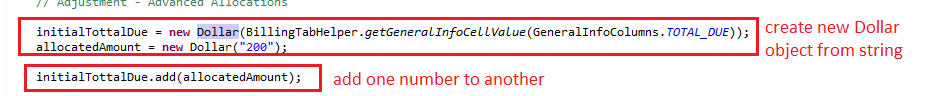
In application all billing amounts are represented by currency in format ($+double).



For operating with this numbers in tests we can use Dollar class that located in *src/main/java/utilities* folder.

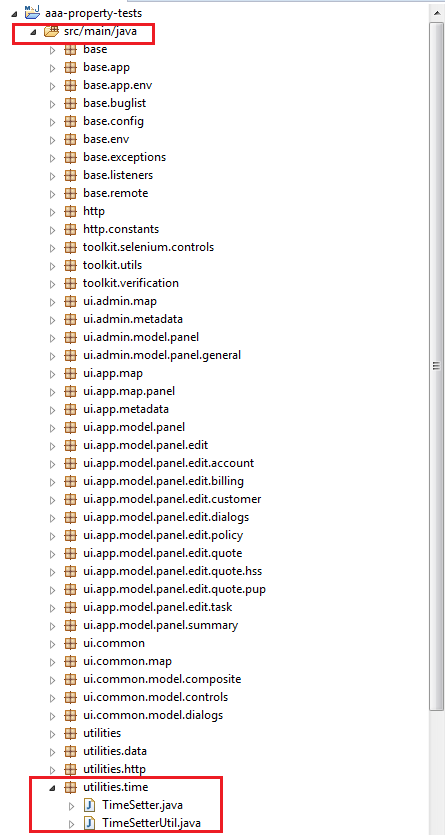


Using Dollar class you can add, multiply, divide and make other actions with billing numbers. Here is an example of using Dollar.



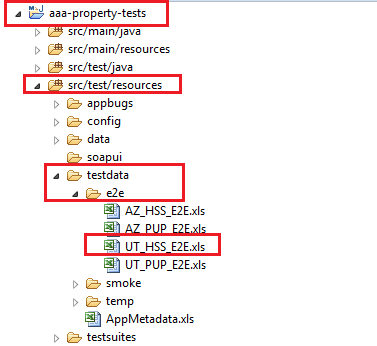
## TimeSetterUtils and TimePointsHelper

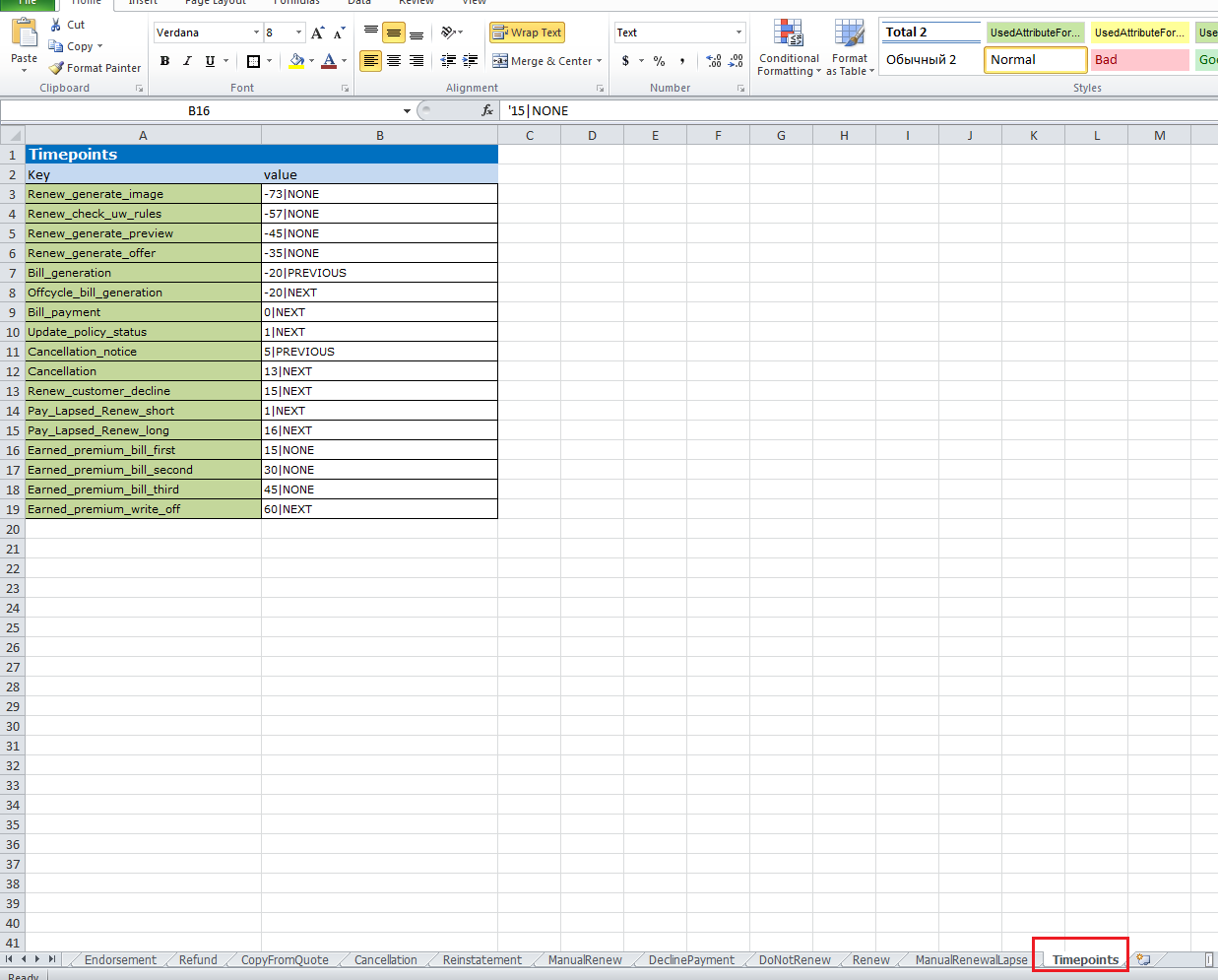
*TimeSetterUtils* class allows us to move time on servers. It’s located in *src/main/java/utilities/time* folder.



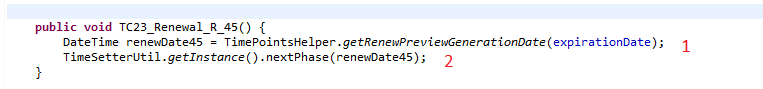
The most useful function is *nextPhase (DateTime time).* It sets time on server according to time in parameter.

*TimePointsHelper* allows to get dates (in DateTime format) of policy events like Generation Date, Renew Image Generation Date, Renew Offer Generation Date and etc. How time will be changed for this events is described in test data on sheet that is named *Timepoints.* For example: open *UT\_HSS\_E2E.xls* and find *Timepoints* sheet. There you can see Key and Value columns. Key is event name, Value is how will be time changed.





The next example shows how to use TimePointsHelper and TimeSetterUtils:



1. Get Renew Preview Generation Date (renewPreviewGenerationDate = policyExpirationDate-45)
2. Set Renew Preview Generation Date on server.

## Appendix 1

## How to create more complex test script

How to create Home policy with Property framework was described in [paragraph 6](#_How_to_create). Let’s analyze more complex example. There is an Example below with detailed comments.

### How to create policy using Policy data with different dataset names and two insureds

**private** **static** String *dataSetAdditional* = "Discounts\_HO3\_Dataset1";

**private** **static** String *dataSetEndorsementFalse* = Discounts\_Endorsement\_HO3\_Dataset1";

**private** **static** String *dataSetEndorsementTrue* = QuoteApplicantAdditionalActiveAAAPolicy\_Discounts\_Endorsement\_2";

@Test

**public** **void** exampleTest() {

Dollar writeOffAmount = **new** Dollar("100");

Dollar initialTottalDue;

Dollar finalTottalDue;

//Open application

*EUApp*().open();

//Open existent customer or create new

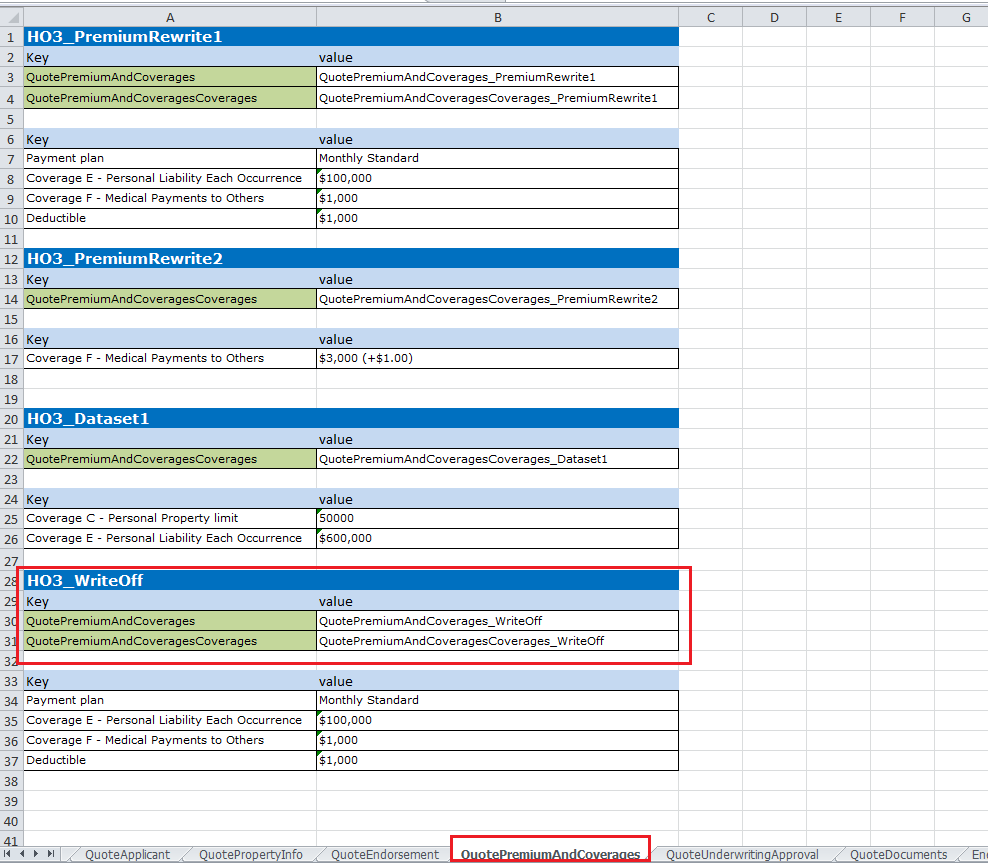
CustomerHelper.*openCustomer*();

//Initiate PolicyData object, HO3\_Dataset1 name will be used by default

HssPolicyData policyData = **new** HssPolicyData(*getDataSuite*(),"HO3\_Dataset1");

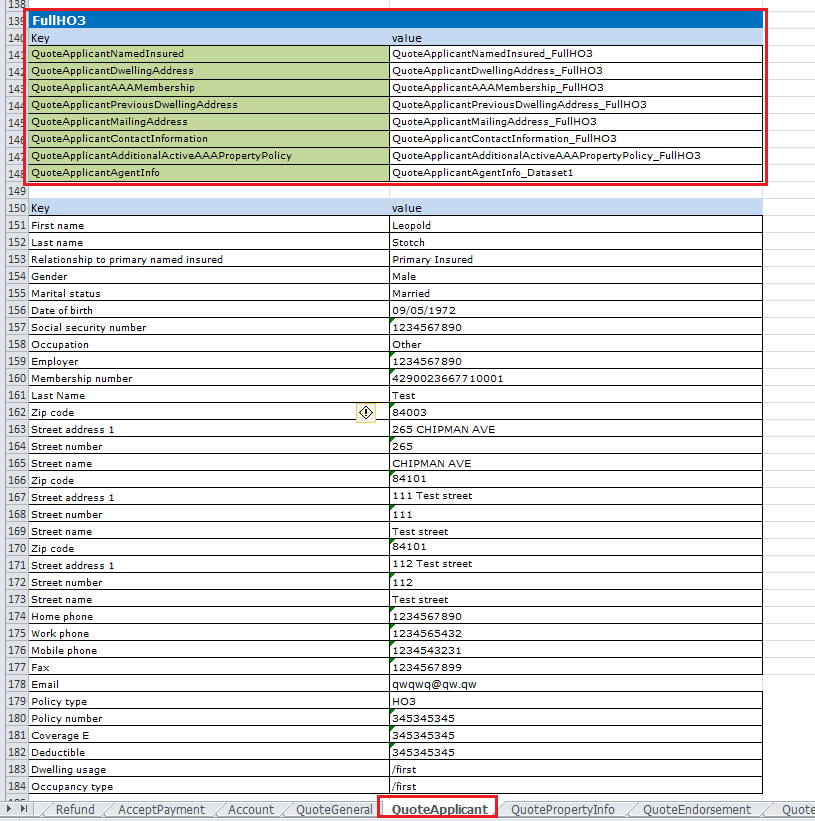
//PremiumAndCoverages tab will be filled with dataset named HO3\_WriteOff

policyData.setPremiumAndCoveragesDataset("HO3\_WriteOff");



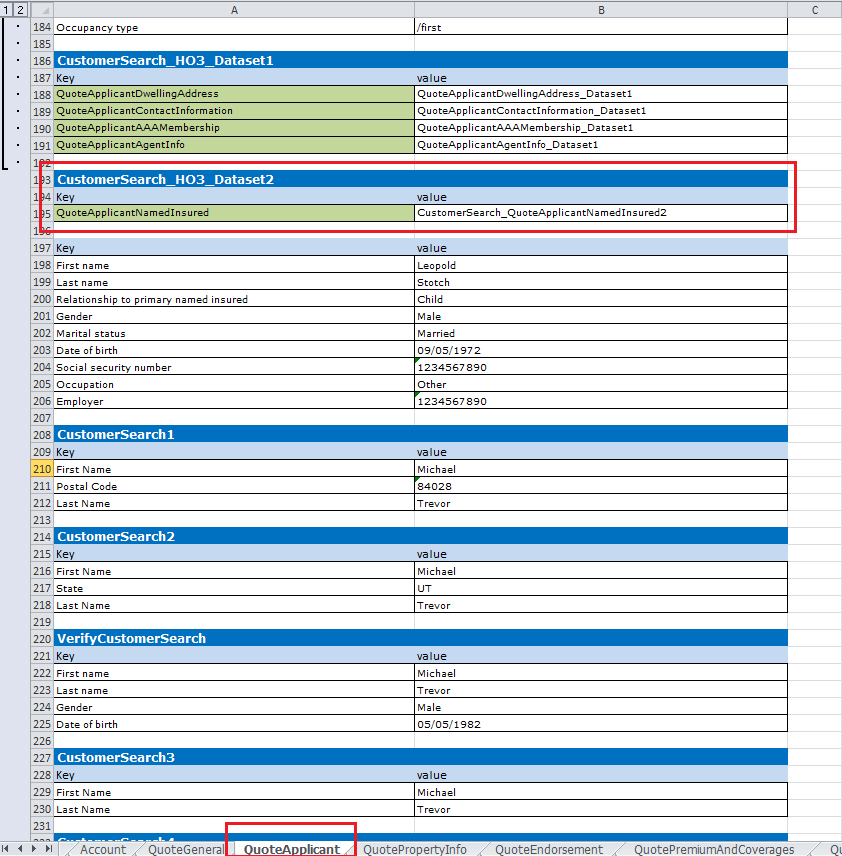
//Applicant tab will be filled with dataset named FullHO3

policyData.setApplicantDataset("FullHO3");



//Add additional insured on applicant tab

policyData.addApplicantNamedInsureds("CustomerSearch\_HO3\_Dataset2");



//create new Policy

HssHelper.*createPolicy*(policyData);

### How to make Billing transaction (write off transaction)

//navigate to Billing view

TopPanel.*navigateTo*(TopPanelTabs.*BILLING*);

initialTottalDue = **new** Dollar(BillingTabHelper.*getGeneralInfoCellValue*(GeneralInfoColumns.*TOTAL\_DUE*));

//Create Write Off 100$ transaction

BillingSummaryPanel.*btnOtherTransactions*.click();

//Fill Other Transactions Tab with HO3\_WriteOff dataset

OtherTransactionsPanel.*fillOtherTransactionsTab*("HO3\_WriteOff");

//click Ok button on Other Transactions Tab

OtherTransactionsPanel.*btnOk*.click();

### How to check Billing transaction appeared in the transactions table

//Check write-off transaction appears

HashMap<PaymentsColumns, String> query = **new** HashMap<PaymentsColumns, String>();

query.put(PaymentsColumns.*TYPE*, "Adjustment");

query.put(PaymentsColumns.*SUBTYPE\_REASON*, "Write-off");

query.put(PaymentsColumns.*AMOUNT*, "("+writeOffAmount.toString()+")");

query.put(PaymentsColumns.*STATUS*, "Applied");

BillingTabHelper.*getPaymentsRow*(query).verify.present();

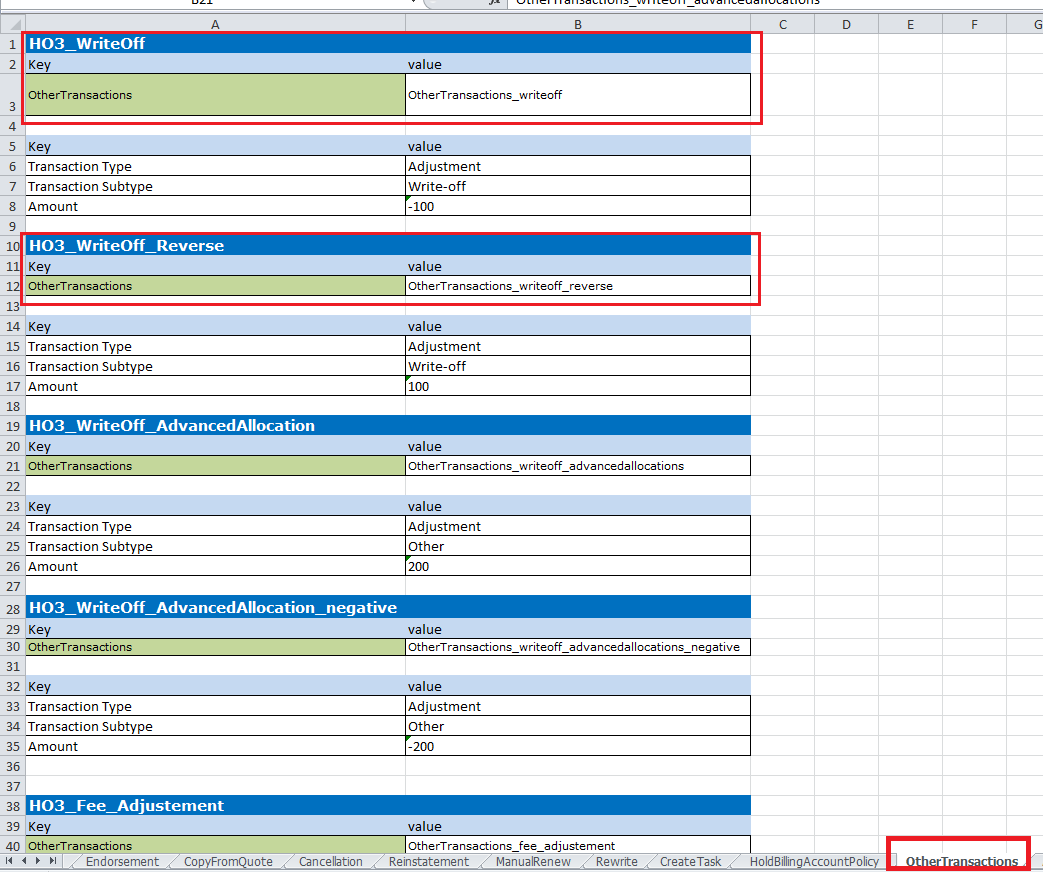
//Create Reversal Write Off 100$ transaction

BillingSummaryPanel.*btnOtherTransactions*.click();

//Fill Other Transactions Tab with HO3\_WriteOff\_Reverse dataset

OtherTransactionsPanel.*fillOtherTransactionsTab*("HO3\_WriteOff\_Reverse");

OtherTransactionsPanel.*btnOk*.click();



//Check reversal write-off transaction appears

query.clear();

query.put(PaymentsColumns.*TYPE*, "Adjustment");

query.put(PaymentsColumns.*SUBTYPE\_REASON*, "Write-off");

query.put(PaymentsColumns.*AMOUNT*, writeOffAmount.toString());

query.put(PaymentsColumns.*STATUS*, "Applied");

BillingTabHelper.*getPaymentsRow*(query).verify.present();

finalTottalDue = **new** Dollar(BillingTabHelper.*getGeneralInfoCellValue*(GeneralInfoColumns.*TOTAL\_DUE*));

//Check Total Due value after write-off/reversal write-off.

CustomAssert.*assertEquals*(initialTottalDue.toString(),finalTottalDue.toString());

### How to make endorsement action

//Initiate Endorsement action

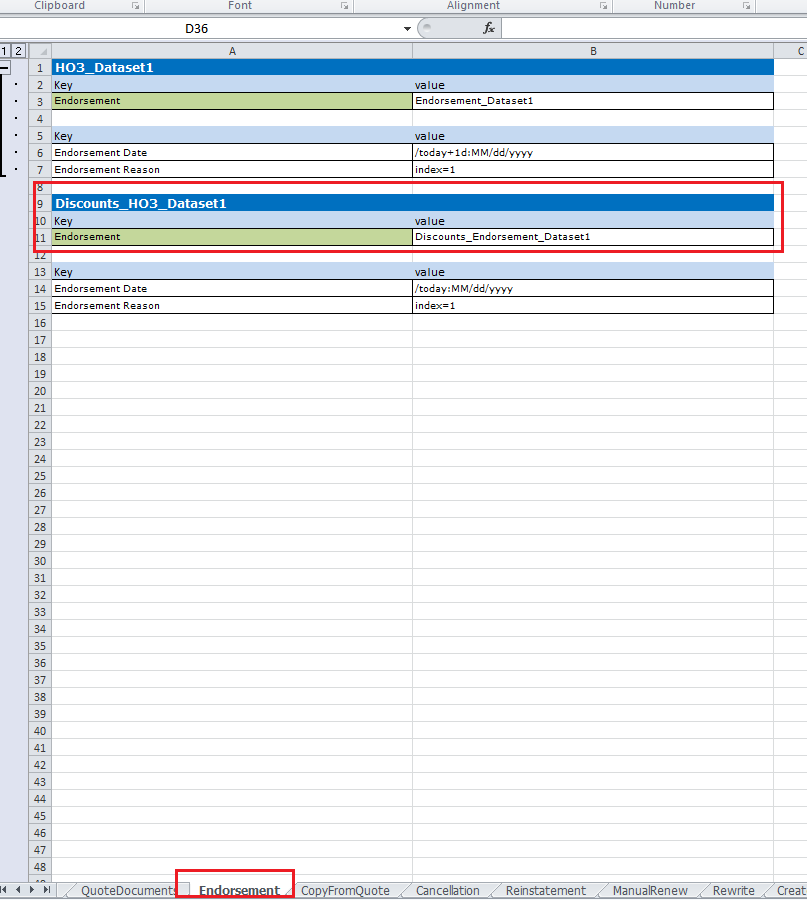
PolicySummaryPanel.*setActionAndGo*(PolicyAction.*ENDORSEMENT*);

//Fill Endorsement tab

EndorsemetEditorPanel.*fillEndorsementTab*(*dataSetAdditional*);

//Click ok button

EditorPanel.*ok*();



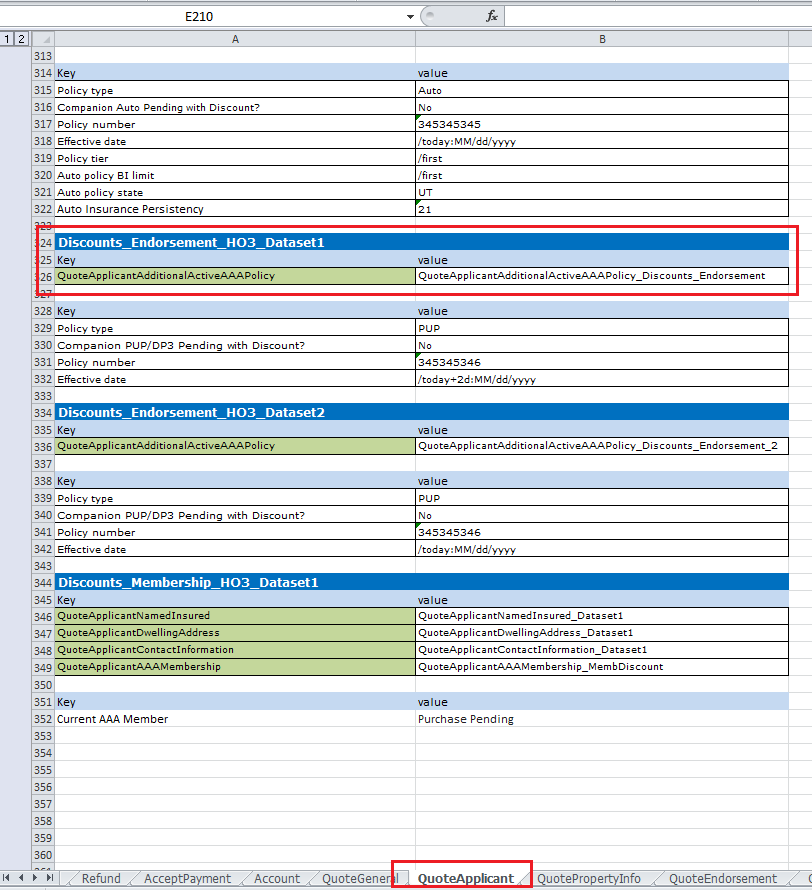
### How to fill only Applicant tab

//Navigate to Applicant tab

QuoteEditorPanel.*navigateTo*(QuoteHSSTabs.*APPLICANT*.get());

//Fill Applicant tab with new data

ApplicantTabEditorPanel.*fillApplicantTab*(*dataSetEndorsementFalse*);



//Navigate to reports tab and order reports

QuoteEditorPanel.*navigateTo*(QuoteHSSTabs.*REPORTS*.get());

ReportsTabEditorPanel.*orderAllReports*();

//Go to Premiums and Coverages tab and rate quote

QuoteEditorPanel.*navigateTo*(QuoteHSSTabs.*PREMIUM\_AND\_COVERAGES\_QUOTE*.get());

//Click Rate button

PremiumsAndCoveragesTabEditorPanel.*rate*();

//Verify error message on error panel

ErrorPanel.*verifyErrorMessage*(1, "Companion policy effective date must be on or before the endorsement date");

//Cancel error panel

ErrorPanel.*cancel*();

//Navigate to Applicant tab

QuoteEditorPanel.*navigateTo*(QuoteHSSTabs.*APPLICANT*.get());

//Fill Applicant tab with new data

ApplicantTabEditorPanel.*quoteApplicantAdditionalActiveAAAPolicyList*.fillValues(BaseTest.*getDataSuite*(). getDataGroup(ApplicantTabEditorPanel.*QUOTE\_APPLICANT*).getDataTable(*dataSetEndorsementTrue*));

//Go to Premiums and Coverages tab and rate quote

QuoteEditorPanel.*navigateTo*(QuoteHSSTabs.*PREMIUM\_AND\_COVERAGES\_QUOTE*.get());

### How to fill only one field from asset list

PremiumsAndCoveragesTabEditorPanel.quotePremiumAndCoverages.getControl("Payment plan").setValue("Monthly Standard");

PremiumsAndCoveragesTabEditorPanel.*rate*();

//Get premium value

Dollar discountPupPolicyPremiumSummary = PremiumsAndCoveragesTabEditorPanel.*getPolicyActualPremium*();

//Open Rating datalis

PremiumsAndCoveragesTabEditorPanel.*viewRatingDetails*();

CustomAssert.*assertTrue*("Discount not applied",!RatingDetailsDialog.*tblDiscounts*.getColumn(4).getCell(14).getValue().equals("0.0"));

//Close Rating details

RatingDetailsDialog.*ok*();

//Bind quote

EditorPanel.*navigateTo*(TabNames.QuoteHSSTabs.*BIND*.get());

BindTabEditorPanel.*purchase*();

### How to verify policy status

//Verify Policy Active

PolicySummaryPanel.*verifyPolicyStatus*(PolicyMeta.PolicyStatus.*ACTIVE*);

//Verify Policy Premium summary same as on premiums and Coverages tab

CustomAssert.*assertEquals*("New Total (Policy premium summary) on policy summary tab",

discountPupPolicyPremiumSummary, **new** Dollar(PolicySummaryPanel.*getPolicyPremiumSummary*()));

}