Thermo-Calc Property Model Framework Documentation

Release 2018a

Thermo-Calc Software AB

Feb 27, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Thermo-Calc Custom Property Models

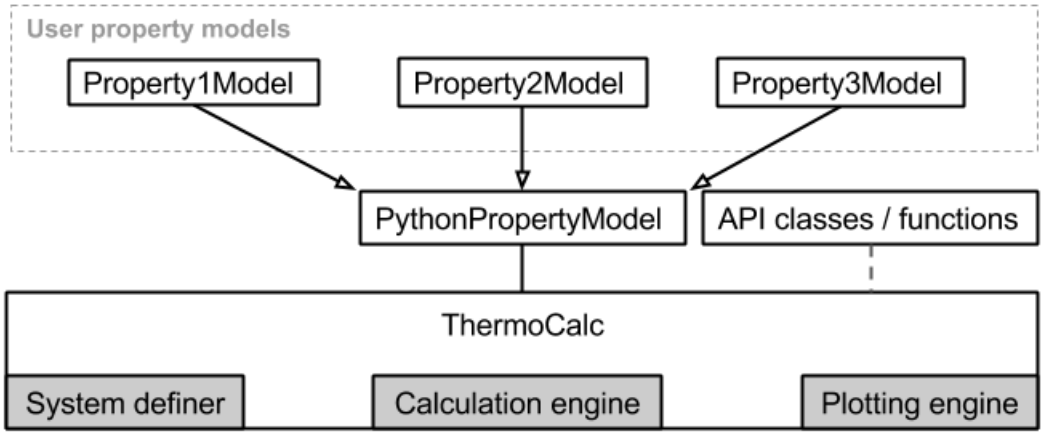
The *Python Property Model API* extends the Thermo-Calc software to enable you to create your own property models written in the easy-to-learn programming language **Python** and the API provides seamless integration of the property models into Thermo-Calc. This includes access to thermodynamic and kinetic calculation routines as well as to the Thermo-Calc plotting system.

The API is based on *Jython*, which is a Python implementation running in the Java Virtual Machine. All Python libraries that are compatible to the Jython distribution can be used within the model (but not libraries based on native C-code).

Model development is possible without any additional installation by using a text editor (for example *Notepad++*). However, using a Python Integrated Development Environment (IDE) provides you with advanced features such as debugging and auto-completion.

1.2 Architecture of the Python Property Model API

Every property model needs to contain a class that implements the interface `PythonPropertyModel`. It is the interface to the Thermo-Calc system and calculation engine and provides the calculation results to the Thermo-Calc plotting engine.



The basic building blocks of the property model API are:

- **ResultQuantity**: Defines a calculation result of a property model that will be provided by the property model after a model evaluation

classes: *ResultQuantity*

- **ThermoCalcCalculator**: Thermodynamic calculation engine of Thermo-Calc for the property models

  classes: *ThermoCalcCalculator*

- **ThermoCalcSystem**: Provides all helper functionality within the property models to access the Thermo-Calc / GUI functionality

  classes: *ThermoCalcSystem*

- **UIComponent**: UI components available within the model panel. Different components are available (for example checkboxes, text fields and lists)

  classes:

  - *UIBooleanComponent*
  - *UICompositionComponent*
  - *UIConditionListComponent*
  - *UIFloatComponent*
  - *UIGeneralListComponent*
  - *UIPhaseListComponent*
  - *UITemperatureComponent*

1.3 Call Events for the Property Model Methods

The table lists which events trigger the property model methods (defined in the interface *PythonPropertyModel*):

Interface method	Event triggering the method to be called
`before_evaluations()`	once after the **Perform** button is clicked, before any call to `evaluate_model()`
`evaluate_model()`	once for each set of conditions, always after the call to `before_evaluations()`
`get_thermocalc_system()`	before any other method call - if the system state in Thermo-Calc has changed
`provide_calculation_result_quantities()`	after each call to `evaluate_model()`
`provide_model_category()`	whenever the Thermo-Calc UI needs to (re-) display components containing model information (for example the model tree)
`provide_model_description()`	once when the model panel is painted (this happens also whenever a model gets checked in the model tree)
`provide_model_name()`	whenever the Thermo-Calc UI needs to (re-) display components containing model information (for example the model tree)
`provide_ui_panel_components()`	once when the model panel is displayed (this happens also whenever a model gets checked in the model tree)

The methods of a single property model are always executed sequentially.

CHAPTER

TWO

INSTALLING JYTHON AND SETTING UP AN IDE

2.1 Installing Jython

This section describes the installation of Jython 2.7.0, but you should install the latest available version.

Note: The Jython installation is only required if you want to have full support of your IDE during the model development (autocompletion, debugging, and so forth). The property models will run automatically in Thermo-Calc using the Jython installation already bundled to Thermo-Calc.

- Before installing Jython, you need to have a correctly working Java SDK installation as a prerequisite.
- Download the latest Jython distribution from www.jython.org (the current installer version is `jython-installer-2.7.0.jar`)
- Execute the installer JAR-file.
- Choose the *Standard* installation.

2.2 Setting up the IDE

The Python property models can be developed using any text editor. For full functionality (autocompletion, debugging, etc.) it is recommended to use a Python IDE. The options described are the *IDEs IntelliJ IDEA Ultimate* and *PyCharm Professional* developed by JetBrains. Another suitable example is the *Microsoft Visual Studio Community* using the *Python Tools for Visual Studio*. If **debugging** the property models is a requirement, then it is necessary to use **commercial** versions of *IntelliJ* or *PyCharm*, otherwise the *Community* editions are sufficient.

2.2.1 IntelliJ IDEA

The information in this section is applicable to IntelliJ IDEA 2016. It is primarily a Java IDE and requires a plugin for full support of Python:

- Go to **File** -> **Settings** -> **Plugins**.
- Choose **Search in repositories** to search for the *Python* plugin.
- Install the plugin.

Installing a *Python SDK*:

- Go to **File** -> **Project Structure**.
- Click the **+** to add a new SDK under **Platform Settings** -> **SDKs**.

- Choose **Python SDK** and **Add Local**.
- Select the file `jython.exe` within the Jython installation, typically located in `C:\jython2.7.0\bin\jython.exe`

Setting the *Project SDK* (required for each model):

- Choose the *Jython SDK* within the **Project** tab as your *Project SDK*.
- **Add a dependency to the *Python property model API* in the Modules tab:**
  - Choose your module and navigate to the **Dependencies** tab.
  - Click **+** and choose **JARs or directories . . .** .
  - Choose the Python property model API directory (`PropertyModels/Interface/ThermoCalcModelInterface` within the property model directory).

2.2.2 PyCharm

This is required for each model:

Installing a *Project Interpreter*:

- Go to **File -> Settings**.
- Add a new *Project Interpreter* by navigating to your project and clicking **+**.
- Choose **Add Local**.
- Select the file `jython.exe` within the Jython installation, typically located in `C:\jython2.7.0\bin\jython.exe`

Adding a dependency to the *Python property model API*:

- Click the settings button for the selected Jython interpreter within the *Project Interpreter* and choose **More . . .**
- Choose the Jython interpreter in the list of available Python interpreters and click **Show paths for selected interpreters**.
- Click the **+** to add the Python property model API directory (`PropertyModels/Interface/ThermoCalcModelInterface` within the property model directory).

Note: When you add the Phython property model API directory to the interpreter settings, it changes your Jython installation. Consequently you may want to use a virtual environment.

CHAPTER

THREE

CREATING A NEW PROPERTY MODEL

A new property model can be created and modified at any time during a Thermo-Calc session. The model is automatically reloaded and you do not need to restart Thermo-Calc.

Note: All property models must be located in a subdirectory of the property model directory. It is located within the Thermo-Calc home directory, which differs for the supported platforms.

Operating system	Property model directory
Windows	C:\Users\Public\Public Documents\Thermo-Calc\ 2018a \PropertyModels
Linux	/home/user/Thermo-Calc/2018a/PropertyModels
Mac	/Users/Shared/Thermo-Calc/2018a/PropertyModels

Make sure that the *Model directory* path in the Thermo-Calc dialog *Tools -> Options -> General* points to the correct property model directory path.

How to Create a New Model:

- Start Thermo-Calc.
- Add a Property Model Calculator to a project in Thermo-Calc.
- Create a new folder within the model directory (as per the installed location). The folder can be given any name.
- Create a new Python model file or copy a file from another model, such as a predefined one. The file extension is *.py. For now, ignore any error messages in the **Event Log** in Thermo-Calc stating that the file does not fulfill the requirements for a property model.
- Give the model a valid file name **identical to the model class name** ("`XYModel.py`")
- Assign the Jython interpreter and the dependency to the property model API within your IDE (see *Setting up the IDE*)
- Extend the model class *XYModel* to fully implement the interface `PythonPropertyModel`
- Implement the model (see *Tutorial*)

Thermo-Calc automatically updates the model whenever it is changed in an editor. If UI components are changed you need to reload the model. Do this by deselecting the check box next to the model and then selecting it again to reload the model.

The model directories can be moved and renamed during the Thermo-Calc session without limitation. When moving or deleting a model from a directory, it is required to move or delete the `XYModel.py` **and** the `XYModel.py.encrypted` file. If the encrypted file is left in its original place, that model will only be switched from the *debuggable* to the *non-debuggable* model (see *Debugging*).

Example models demonstrating the most important features can be found in the chapter *Tutorial* and in the property model directory.

Note: The Python file as well as the model class need to follow the syntax "XYModel". **This name needs to be unique within all property models**. You can define other classes or methods within the model file as long as these do not interfere with the required syntax for a model.

Note: It is possible to store several models within one model directory. These are distinguished based on their model name.

CHAPTER

FOUR

TUTORIAL

4.1 Example 1: A Basic Property Model with Thermodynamic Properties

This tutorial demonstrates the most relevant tasks to create a property model. A basic property model is created that provides you with the fraction of a certain phase in dependency of any defined condition.

Let's first take a look at the complete model code. The source code **must** be saved in a file (e.g. `Example1Model.py`) in a subdirectory of the property model directory for your operating system (see *Creating a New Property Model* for details). You can copy the code below directly to your editor, but make sure to remove all line break characters.

```
from PythonPropertyModel import PythonPropertyModel
from ThermoCalcSystem import *
from ThermoCalcCalculator import ThermoCalcCalculator


class Example1Model(PythonPropertyModel):
    """Example 1"""
    _PHASE_FRACTION_QUANTITY = "PHASE_FRACTION_QUANTITY"
    _PHASE_LIST = "PHASE_LIST_COMPONENT"

    def __init__(self):
        """Constructor"""
        self._thermocalc_system = None  # type: ThermoCalcSystem

    def get_thermocalc_system(self, thermocalc_system):
        """Called by Thermo-Calc whenever an updated state of the Thermo-Calc core␣
↪system needs to be injected into the model"""
        self._thermocalc_system = thermocalc_system

    def before_evaluations(self, calculator):
        """Called by Thermo-Calc immediately before the first model evaluation (using␣
↪evaluate_model). Use this method for any required preparations."""
        pass

    def provide_model_category(self):
        """Called by Thermo-Calc when the model should provide its category (shown in␣
↪the Thermo-Calc model tree)."""
        return ["Examples"]

    def provide_model_name(self):
        """Called by Thermo-Calc when the model should provide its name (shown in the␣
↪Thermo-Calc model tree)."""
        return "Example 1"
```

```
    def provide_model_description(self, locale):
        """Called by Thermo-Calc when the model should provide its detailed␣
↪description."""
        description = "This is an example model."
        return description

    def provide_calculation_result_quantities(self):
        """Called by Thermo-Calc when the model should provide its result quantity␣
↪objects."""
        result_quantities = [create_general_quantity(Example1Model._PHASE_FRACTION_
↪QUANTITY, "Phase fraction")]
        return result_quantities

    def provide_ui_panel_components(self):
        """Called by Thermo-Calc when the model should provide its UI components for␣
↪the model panel to be plotted."""
        ui_components = [create_phase_list_ui_component(Example1Model._PHASE_LIST,
↪"Phase: ", "Phase list", default_phase="FCC_A1")]
        return ui_components

    def evaluate_model(self, calculator):
        """
        Called by Thermo-Calc when the model should be actually calculated.

        :param calculator:          Calculation engine
        :type calculator:           ThermoCalcCalculator
        """
        chosen_phase = self._thermocalc_system.get_ui_list_value(Example1Model._PHASE_
↪LIST)
        calculator.compute_equilibrium(True)
        phase_frac = calculator.get_single_value("NPM(" + chosen_phase + ")")
        self._thermocalc_system.set_result_quantity_value(Example1Model._PHASE_
↪FRACTION_QUANTITY, phase_frac)
```

Every property model is a Python model containing the model class. This class needs to be derived from *PythonPropertyModel*, which is the interface for all property models. You need to implement all of its abstract methods. Generally, when executing a property model, Thermo-Calc is calling these methods to either obtain information from the model or to transfer data to the model.

Note: The Python file as well as the model class must follow the syntax "XYModel". **This name must be unique within all property models**. You can define other classes or methods within the model file as long as these do not interfere with the required syntax for a model.

4.1.1 Access to the Thermo-Calc System

At any time you can access all data from the Thermo-Calc system and especially from the predecessor activity nodes (for example from the System Definer) via a *ThermoCalcSystem* object. That object is updated when, for example, the configuration in the System Definer changes. If that happens, a new object is provided by Thermo-Calc to the model by calling *PythonPropertyModel.PythonPropertyModel.get_thermocalc_system()*. It is recommended that you define an instance variable, which gives access to the up-to-date object at all times:

```
def __init__(self):
    """Constructor"""
    self._thermocalc_system = None  # type: ThermoCalcSystem

def get_thermocalc_system(self, thermocalc_system):
    """Called by Thermo-Calc whenever an updated state of the Thermo-Calc␣
↪core system needs to be injected into the model"""
    self._thermocalc_system = thermocalc_system
```

Note: Any method within the API provides a docstring, that can be used by the IDE to provide you autocompletion and autodocumentation. But as Python is using duck-typing, the API cannot always derive the correct variable types automatically. The constructor above shows an example on how to give a type hint to the IDE (PyCharm / IntelliJ IDEA) to provide autocompletion (in the example for `self._thermocalc_system`).

4.1.2 Basic Information About the Model

These methods provide the model category, name and description and are used by Thermo-Calc to present the models within a tree component in the Property Model Calculator:

```
def provide_model_category(self):
    """Called by Thermo-Calc when the model should provide its category␣
↪(shown in the Thermo-Calc model tree)."""
    return ["Examples"]

def provide_model_name(self):
    """Called by Thermo-Calc when the model should provide its name (shown␣
↪in the Thermo-Calc model tree)."""
    return "Example 1"

def provide_model_description(self, locale):
    """Called by Thermo-Calc when the model should provide its detailed␣
↪description."""
    description = "This is an example model."
    return description
```

A property model can be included in multiple categories. The description should contain helpful information to the user regarding the scientific background of the model as well a short description about how to use it. You can use usual string formatting ("\n", ...)

4.1.3 Defining Input and Output Parameters

All required input parameters are defined by creating components for the User Interface (UI). Every output parameter is defined as a *ResultQuantity*.

```
def provide_calculation_result_quantities(self):
    """Called by Thermo-Calc when the model should provide its result␣
↪quantity objects."""
    result_quantities = [create_general_quantity(Example1Model._PHASE_
↪FRACTION_QUANTITY, "Phase fraction")]
    return result_quantities
```

The result quantities need to be provided within a list. Several result quantity types are available that provide different predefined properties, for example for a temperature.

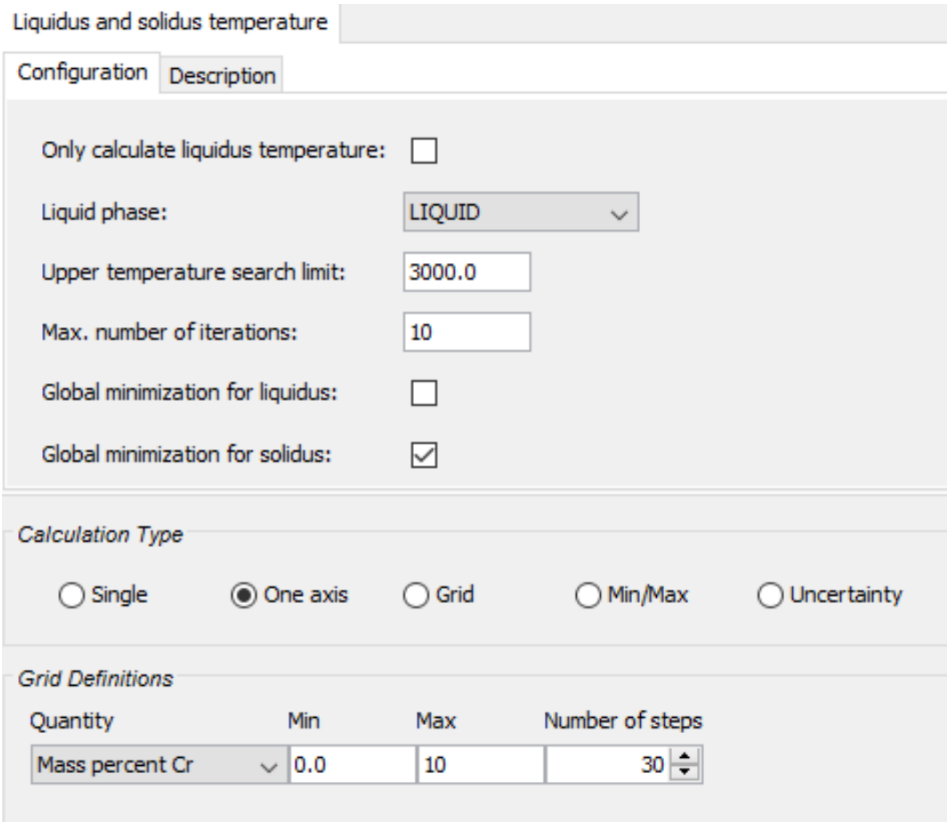
> **Caution:** Every result quantity is expected to be provided in SI units.

The purpose of the result quantity objects is to provide a container for transfering calculation results from the model to Thermo-Calc for plotting and other tasks. You will later assign values to the result quantities in the *PythonPropertyModel.PythonPropertyModel.evaluate_model()* method.

```
def provide_ui_panel_components(self):
    """Called by Thermo-Calc when the model should provide its UI components␣
↪for the model panel to be plotted."""
    ui_components = [create_phase_list_ui_component(Example1Model._PHASE_
↪LIST, "Phase: ", "Phase list", selected_phase="FCC_A1")]
    return ui_components
```

Each input parameter is represented by a UI-component. The property model API provides the most common ones. The order in the list defines the order of appearance in the Property Model Calculator window. You can later read the values set by the user in *PythonPropertyModel.PythonPropertyModel.evaluate_model()*.

Example of a more complex property model UI:



4.1.4 Performing the Calculations

Any setup that is required before the calculations needs to be provided in the method *PythonPropertyModel.PythonPropertyModel.before_evaluations()*:

```
def before_evaluations(self, calculator):
    """Called by Thermo-Calc immediately before the first model evaluation␣
↪(using evaluate_model). Use this method for any required preparations."""
    pass
```

The model is evaluated separately for every set of conditions. This means that in case of varying, for example two component concentrations, the *PythonPropertyModel.PythonPropertyModel.evaluate_model()* method is called separately for each composition:

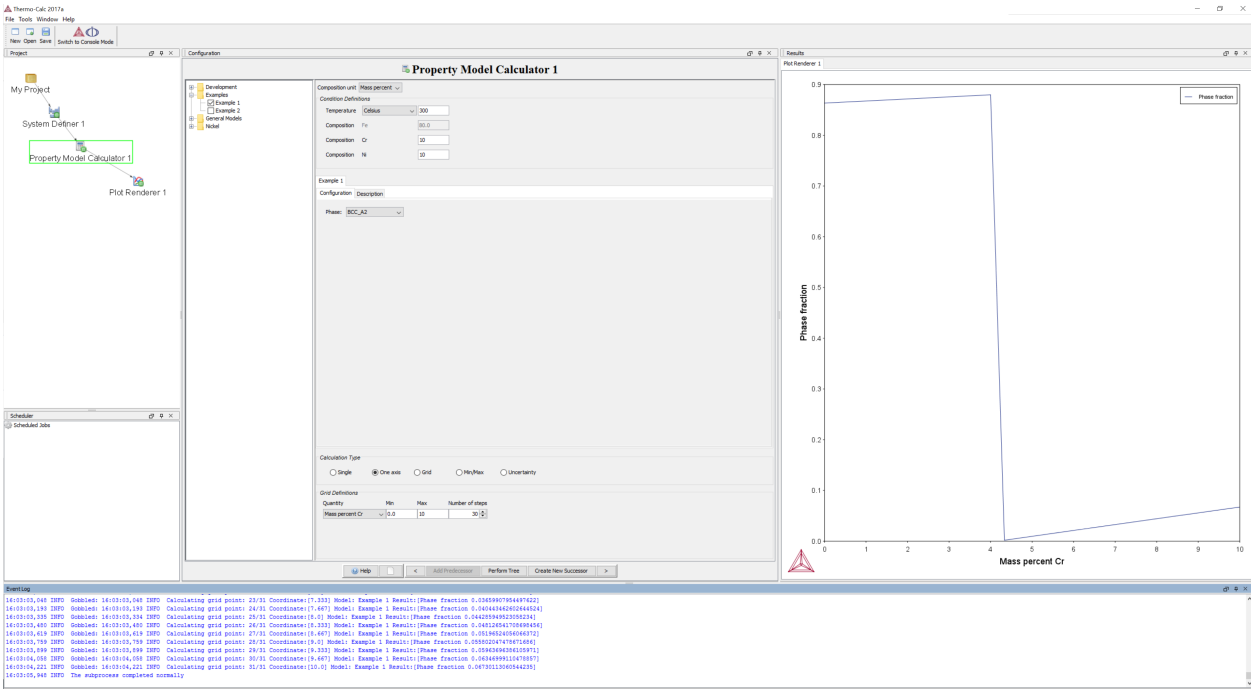
```
def evaluate_model(self, calculator):
    """
    Called by Thermo-Calc when the model should be actually calculated.

    :param calculator:          Calculation engine
    :type calculator:           ThermoCalcCalculator
    """
    chosen_phase = self._thermocalc_system.get_ui_list_value(Example1Model._
↪PHASE_LIST)
    calculator.compute_equilibrium(True)
    phase_frac = calculator.get_single_value("NPM(" + chosen_phase + ")")
    self._thermocalc_system.set_result_quantity_value(Example1Model._PHASE_
↪FRACTION_QUANTITY, phase_frac)
```

Note how type hinting for providing autocompletion is implemented in the example.

Tip: If sophisticated Thermo-Calc functionality is required that is not directly available within the Python API, Thermo-Calc Console Mode commands can be used: You have access to the full set of Thermo-Calc Console Mode commands available in Poly3 by using the methods *ThermoCalcCalculator.ThermoCalcCalculator.send_command_to_poly3()* and *ThermoCalcCalculator.ThermoCalcCalculator.get_single_value()*.

Caution: The lifetime of the *ThermoCalcCalculator* object is *restricted* to the subsequent calls to *PythonPropertyModel.PythonPropertyModel.evaluate_model()* and the initial *PythonPropertyModel.PythonPropertyModel.before_evaluations()* call while performing a model calculation. Consequently you must not store that object within an instance variable.

The screenshot shows an example calculation with the property model for the Fe - 10 Ni - x Cr system using the FEDEMO database, illustrating the functionality of the model:



4.2 Example 2: Property Model Using Apache Math

This example describes how to perform mathematical calculations using the Java Apache Math library. To keep it simple the sine of the temperature is calculated. This has limited practical application and is intended only as a demonstration.

Note: The Python property models in Thermo-Calc are executed as Jython code within the Java Virtual Machine of Thermo-Calc. It means that any Java library can be imported and used directly within the models. This includes the Apache Math library for numerical calculations.

Caution: Within Jython it is *not* possible to use any Python library based on native code (typically C-code). Consequently the property models **cannot use the numerical Python libraries NumPy and SciPy**. **Instead use the Java Apache Math library**.

The Apache Math library is automatically added to the classpath when executing the property model within Thermo-Calc. However, you will need to add a reference to the file `commons-math3-version.jar` in your IDE if you want to have autocompletion support. This file is located in the Thermo-Calc installation directory. For example IntelliJ IDEA supports referencing that Java-JAR, while this is not possible using PyCharm. However, this does not prevent you from running the property model in Thermo-Calc.

This is the code for the property model. Remember that it needs to be stored in the file `Example2Model.py` in a subdirectory of the property model directory:

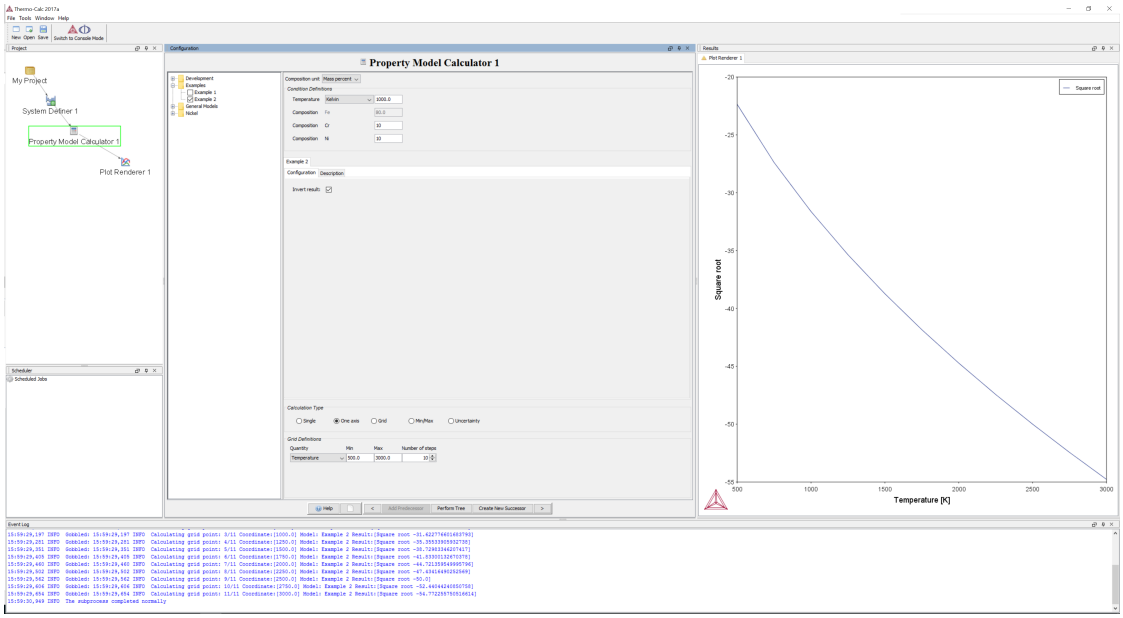
```
# Python libraries
from PythonPropertyModel import PythonPropertyModel
from ThermoCalcCalculator import ThermoCalcCalculator
from ThermoCalcSystem import *
```

```
# Java libraries
from org.apache.commons.math3.analysis.function import Sqrt


class Example2Model(PythonPropertyModel):
    """Example 2"""
    _SQUARE_ROOT = "SQUARE_ROOT"
    _INVERT_RESULT_CHECKBOX = "INVERT_RESULT_CHECKBOX"

    def __init__(self):
        """Constructor"""
        self._thermocalc_system = None  # type: ThermoCalcSystem
        self._sqrt_provider = Sqrt()

    def get_thermocalc_system(self, thermocalc_system):
        """Called by Thermo-Calc whenever an updated state of the Thermo-
↪Calc core system needs to be injected
        into the model"""
        self._thermocalc_system = thermocalc_system

    def before_evaluations(self, calculator):
        """Called by Thermo-Calc immediately before the first model␣
↪evaluation (using evaluate_model). Use this
        method for any required preparations."""
        pass

    def provide_model_category(self):
        """Called by Thermo-Calc when the model should provide its category␣
↪(shown in the Thermo-Calc model tree)."""
        return ["Examples"]

    def provide_model_name(self):
        """Called by Thermo-Calc when the model should provide its name␣
↪(shown in the Thermo-Calc model tree)."""
        return "Example 2"

    def provide_model_description(self, locale):
        """Called by Thermo-Calc when the model should provide its detailed␣
↪description."""
        description = "This is an example model."
        return description

    def provide_calculation_result_quantities(self):
        """Called by Thermo-Calc when the model should provide its result␣
↪quantity objects."""
        result_quantities = [create_general_quantity(Example2Model._SQUARE_
↪ROOT, "Square root")]
        return result_quantities

    def provide_ui_panel_components(self):
        """Called by Thermo-Calc when the model should provide its UI␣
↪components for the model panel to be plotted."""
        ui_components = [create_boolean_ui_component(Example2Model._INVERT_
↪RESULT_CHECKBOX, "Invert result: ",
                                                     "Determines if the result␣
↪will be inverted ...", True)]
        return ui_components
```

```
    def evaluate_model(self, calculator):
        """
        Called by Thermo-Calc when the model should be actually calculated.

        :param calculator:          Calculation engine
        :type calculator:           ThermoCalcCalculator
        """
        temp = calculator.get_temperature()
        square_root = self._sqrt_provider.value(temp)
        if self._thermocalc_system.get_ui_boolean_value(Example2Model._
↪INVERT_RESULT_CHECKBOX):
            square_root = -square_root

        self._thermocalc_system.set_result_quantity_value(Example2Model._
↪SQUARE_ROOT, square_root)
```

In more practical examples, the numerical calculations would be coupled to thermodynamic calculations.

The screenshot shows an example of a calculation with the dummy property model:



CHAPTER

FIVE

DEBUGGING

All property models that are available as Python source code can be debugged from an IDE using a Python remote debugger.

Note: Most built-in Thermo-Calc property models are only available as encrypted files and cannot be debugged.

A property model is in the *debuggable* state, if its Python-file (`XYModel.py`) is available in the model directory in Thermo-Calc. If only the encrypted file (`XYModel.py.encrypted`) is present, it is in the *non-debuggable state* and can be executed but not debugged.

Remote debugging should be possible with any remote debugging library for Python that works with Jython. Thermo-Calc has tested *IntelliJ IDE* **Ultimate** / *PyCharm* **Professional** to work with the Thermo-Calc property models. You can use any editor or the Community edition of IntelliJ IDEA / PyCharm, however debugging of the models is not available.

5.1 Setting up IntelliJ IDEA / PyCharm for Debugging Models

It is expected that the IDE is set up completely according to *Setting up the IDE*.

The Thermo-Calc application is running in a Java Virtual Machine and is invoking the Python code of the model via Jython. The following steps are required to prepare remote debugging in IntelliJ IDEA Ultimate (note that the IntelliJ Community Edition does not provide remote debugging functionality):

- **Place the following lines in the Python model in front of the part you want to debug:**
  - Typically these are placed at the beginning of the *PythonPropertyModel.PythonPropertyModel.evaluate_model()* method:

```
import sys
sys.path.insert(0, "C:\\Users\\UserName\\.IntelliJIdea2016.
↪3\\config\\plugins\\python\\pycharm-debug.egg")
import pydevd
self._thermocalc_system.log_warning("Attach now to the process to proceed ...
↪")  # optional to inform the user that the model waits for the debug server
pydevd.settrace('localhost', port=5678, stdoutToServer=True,␣
↪stderrToServer=True, suspend=True)
```

Warning:

- Currently you must not use the `pydevd`-library available via `pip` on PyPy. It is outdated and does not work as expected. Use the version bundled to IntelliJ IDEA as shown above.

- You may use any port; the only requirement is that it is unused by other programs.

- **Create a remote debugger configuration in IntelliJ:**
  - Go to the menu "*Run*", choose "*Edit configurations*" and add a configuration by clicking "+".
  - Choose "*Python Remote Debug*".
  - Give that configuration a good name, for example "*Remote Debug Server*".
  - Set the host name to "*localhost*" and the port to that defined in your Python source code (*5678* in our example).
- **Start debugging:**
  - Choose the remote debugging configuration defined above and start it in debug mode (*Crtl + F9*). It should give now a wait message in the output window.
  - Start your desired user-defined property model calculation in Thermo-Calc by clicking **Perform tree**. The model should now connect to the debug server and the IDE should stop directly after the `settrace()` call in the Python model.
  - Now you can inspect your variables and step through the code. You can also use the "*Debug Command Line*" in the menu "*Tools*" for interactive testing.
  - You can safely ignore warnings that are occurring.

Note: Thermo-Calc executes the model in a temporary subprocess. This means that the model opens a new connection to the debug server during each execution. This is intended behavior and implies that the connection does not need to be closed manually.

5.2 Editing Your Model during a Thermo-Calc Session

You can edit your model at any time, once it is saved. It is **automatically reloaded** by Thermo-Calc and can immediately be used without restarting Thermo-Calc. You can also debug the edited model.

Note: If IntelliJ IDEA misses a source file during remote debugging, it opens an empty file `<string>` or emits error messages similar to "*Error in file "<string>", line unknown*" if the source file where the error occurred cannot be found. In that case you can still proceed debugging by using "*Resume Program*" (*using the F9 key)*.

Warning: Thermo-Calc does not trigger reloading the model if another imported module than the model itself has been edited.

CHAPTER

SIX

PROVIDING TRANSLATIONS

The Property Model Framework supports multi-language property models based on Java Properties-files. All property models released by Thermo-Calc are using that mechanismn to provide a user interface in all languages supported by Thermo-Calc. This functionality can be as well applied by any user in their own property models.

6.1 Example

The following code snippet shows how to implement localized property models using the class *JavaPropertyManager* (the model needs to have the source file name `MyModel.py`):

```
from JavaPropertyManager import JavaPropertyManager
# further imports to come here

def __init__(self):
    """Constructor"""
    self._properties = JavaPropertyManager("MyModel")

# all other methods to come here

def provide_model_name(self):
    """Called by Thermo-Calc when the model should provide its name (shown in the␣
↪Thermo-Calc model tree)."""
    return self._properties.get("modelName.text")
```

The corresponding snippet of the Properties-file `MyModel.properties` could look like that:

```
modelName.text=MyModel
modelCategory.text=General Models
```

For example a Properties-file `MyModel_de.properties` that is translated to German looks like:

```
modelName.text=Mein Modell
modelCategory.text=Allgemein
```

6.2 Java Properties-files

The class *JavaPropertyManager* automatically chooses the language that is currently set in Thermo-Calc. When using localization, you need to provide at least the default Properties-file (syntax: `XYModel.properties`). In that file the English translations should be provided. Further translations can be provided in Properties-files with the syntax

`XYModel_sv.properties` (or equivalent for other languages). If the required Properties-file or a certain key is missing, Thermo-Calc uses automatically the key in the default Properties-file.

The Properties-files must either be placed in the property model resources directory `jythonModelsResources` that is located in the program file directory of your installation (for example C:\Program Files\Thermo-Calc\ 2018a \jythonModelsResources on Windows) or you can pass a user-defined location to the *JavaPropertyManager* constructor.

Please obtain more general information on Java Properties-files from the Java documentation if required.

CHAPTER

SEVEN

API REFERENCE

7.1 PythonPropertyModel Interface

class PythonPropertyModel.**PythonPropertyModel**

Bases: org.python.proxies.PythonPropertyModel$PythonPropertyModel$14

Customer-available Python property model API interface. The model object is created once ahead of each calculation (in its own subprocess) after the user clicks "Perform".

before_evaluations(*calculator*)

Called by Thermo-Calc immediately before the first model evaluation (using evaluate_model). Use this method for any required preparations.

Parameters **calculator** (ThermoCalcCalculator) – calculation engine, this object exists only during this method call and the following evaluate_model() call and will be destroyed afterwards. Therefore do not store this object outside of the method.

evaluate_model(*calculator*)

Called by Thermo-Calc when the model should be actually calculated.

Parameters **calculator** (ThermoCalcCalculator) – calculation engine, this object exists only during this method call and the previous before_evaluations() call and will be destroyed afterwards. Therefore do not store this object outside of the method. The object contains all required information from the Thermo-Calc core for the calculation.

get_thermocalc_system(*thermocalc_system*)

Called by Thermo-Calc whenever an updated state of the Thermo-Calc core system needs to be injected into the model. Thread-safetey is guaranteed, i.e. the method will only be called directly before but never during other API method calls.

Parameters **thermocalc_system** (ThermoCalcSystem) – Thermo-Calc core system providing access to all information on the thermodynamic system currently defined by the user. This object is guaranteed to live until it is replaced. It can and should be stored in an instance variable within the model class.

provide_calculation_result_quantities()

Called by Thermo-Calc when the model should provide its result quantity objects.

Returns result quantity objects of the model (to be filled later with results in evaluate_model())

Return type list of ResultQuantity

provide_model_category()

Called by Thermo-Calc when the model should provide its category (shown in the Thermo-Calc model tree).

Returns category of the model, it may be present in serveral categories

Return type list of str

provide_model_description(*locale*)
Called by Thermo-Calc when the model should provide its detailed description.

Parameters **locale** (*java.util.Locale*) – locale of the model provided for a localized (language-dependent) description

Returns description text for the model

Return type str

Raises **RuntimeException** – if the model requires a locale, it should throw if the locale is not provided

provide_model_name()
Called by Thermo-Calc when the model should provide its name (shown in the Thermo-Calc model tree).

Returns name of the model

Return type str

provide_ui_panel_components()
Called by Thermo-Calc when the model should provide its UI components for the model panel to be plotted.

Returns model UI panel components in the order to be presented in the model panel

Return type list of UIComponent

7.2 ResultQuantity

class ResultQuantity.**ResultQuantity**(*quantity_id*, *description*, *quantity_type*)
Defines a calculation result quantity of a property model that is identified by a unique ID

Constructor.

Parameters

- **quantity_id** (*str*) – unique ID of the quantity
- **description** (*str*) – description of the quantity (shown in the Thermo-Calc UI)
- **quantity_type** (*"enum" defined in ResultQuantity*) – type of the quantity (defines the unit)

ENERGY_QUANTITY = 2

GENERAL_QUANTITY = 0

TEMPERATURE_QUANTITY = 1

get_description()
Obtains the description of the quantity.

Returns description of the quantity

Return type str

get_id()
Obtains the ID of the quantity.

Returns unique ID of the quantity

Return type str

get_type()
Obtains the type of quantity.

Returns type of the quantity

Return type "enum" defined in ResultQuantity

7.3 ThermoCalcCalculator

class ThermoCalcCalculator.**PhaseStatus**
Bases: object

Defines the status of a phase

DORMANT = 2

ENTERED = 0

FIXED = 1

SUSPENDED = 3

class ThermoCalcCalculator.**SearchDirection**
Bases: object

Defines the search direction in the compute_transition() method

NEGATIVE = 0

POSITIVE = 1

class ThermoCalcCalculator.**ThermoCalcCalculator**(*model_engine*)
Bases: object

Thermodynamic calculation engine of Thermo-Calc for the property models

Constructor. Note: This class should never be instantiated by the model, it will always be provided by Thermo-Calc with the correct and updated data.

Parameters **model_engine** (*se.thermocalc.unite.modelcalculator.domain.ModelEngine*) – Thermo-Calc internal property model engine

compute_chemical_diffusion_cofficient(*phase*, *diffusing_constituent*, *gradient_constituent*, *reference_constituent*)
Computes a chemical diffusion coefficient.

Parameters

- **phase** (*str*) – name of the phase of interest
- **diffusing_constituent** (*str*) – name of the diffusing constituent
- **gradient_constituent** (*str*) – name of the gradient constituent
- **reference_constituent** (*str*) – name of the reference constituent

Returns the chemical diffusion coefficient (in m/s**2)

Return type float

compute_equilibrium(*use_global*)
Performs a thermodynamic equilibrium calculation using the current settings of the calculator

Parameters **use_global** (*bool*) – flag defining if global minimization is used: true for using it, false for not using it

compute_intrinsic_diffusion_coefficient(*phase*, *diffusing_constituent*, *gradient_constituent*, *reference_constituent*)

Computes an intrinsic diffusion coefficient.

Parameters

- **phase** (*str*) – name of the phase of interest
- **diffusing_constituent** (*str*) – name of the diffusing constituent
- **gradient_constituent** (*str*) – name of the gradient constituent
- **reference_constituent** (*str*) – name of the reference constituent

Returns the intrinsic diffusion coefficient (in m/s**2)

Return type float

compute_tracer_diffusion_coefficient(*phase*, *constituent*)

Computes a tracer diffusion coefficient.

Parameters

- **phase** (*str*) – name of the phase of interest
- **constituent** (*str*) – name of the diffusing constituent

Returns the chemical diffusion coefficient (in m/s**2)

Return type float

compute_transition(*phase_to_form*, *condition_to_relax*, *step_length*, *direction*, *use_global*)

Compute transition command identical to that of the Thermo-Calc console. Determines the ocurrence of a phase transition when varying a certain condition. Example: solvus temperature of a phase. Note: After finishing, this command has adjusted the equilibrium present in the calculator to the requested transition. The requested properties such as the transition temperature can be read using the appropriate methods.

Parameters

- **phase_to_form** (*str*) – name of the phase to be formed
- **condition_to_relax** (*str*) – condition to be relaxed in Thermo-Calc console syntax (e.g. "T" or "W(CR)")
- **step_length** (*float*) – step length of the relaxed condition, the minimum possible value is 1.0, the unit is that of the condition
- **direction** (SearchDirection) – direction of search for the relaxed condition (positive or negative)
- **use_global** (*bool*) – flag defining if global optimization is used

delete_condition(*condition_in_classic_syntax*)

Deletes a condition from the calculator.

Parameters **condition_in_classic_syntax** (*str*) – condition to be deleted in Thermo-Calc console syntax

get_conditions()

Obtains the currently defined conditions in the calculator.

Returns list of conditions currently defined in the calculator in Thermo-Calc console syntax

Return type list of str

get_degrees_of_freedom()

Obtains the degrees of freedom in current calculator.

Returns degrees of freedom in the current calculator

Return type int

get_molar_volume(*phase_name*)
Obtains the molar volume of a phase.

Parameters **phase_name** (*str*) – name of the phase

Returns the molar volume of that phase

Return type float

get_phase_status(*phase_name*)
Obtains the current status of a phase.

Parameters **phase_name** (*str*) – name of the phase

Returns status of that phase

Return type *PhaseStatus*

get_single_value(*command_in_classic_syntax*)
Obtains any single value from the calculator using the Thermo-Calc console syntax. Can be used as a workaround for commands not existing in the property model API.

Parameters **command_in_classic_syntax** (*str*) – Thermo-Calc console syntax command specifying the value

Returns the requested value in the same unit as it would be in the Thermo-Calc console mode

Return type float

get_stable_phases_in_equilibrium()
Obtains all phases thare are stable in the current equilibrium.

Returns all phases stable in the current equilibrium

Return type list of str

get_temperature()
Obtains the currently set temperature in the calculator.

Returns the currently set temperature (in K)

Return type float

send_command_to_poly3(*command_in_classic_syntax*)
Sends a console syntax command directly to the Poly3 model. This command accepts all console commands available in POLY3 and can be used as a workaround for commands not existing in the property model API.

Parameters **command_in_classic_syntax** (*str*) – Thermo-Calc console syntax command

set_component_entered(*component*)
Sets a component to status ENTERED in the calculator.

Parameters **component** (*str*) – name of the component

set_component_special(*component*)
Sets a component to status SPECIAL in the calculator.

Parameters **component** (*str*) – name of the component

set_component_suspended(*component*)
Sets a component to status SUSPENDED in the calculator.

Parameters **component** (*str*) – name of the component

set_condition(*condition_in_classic_syntax*)
Sets a new condition in the calculator.

Parameters **condition_in_classic_syntax** (*str*) – new condition in Thermo-Calc console syntax

set_phase_dormant(*phase*)
Sets a phase to the status DORMANT.

Parameters **phase** (*str*) – name of the phase to be set to the status dormant

set_phase_entered(*phase*, *amount*)
Sets a phase to the status ENTERED.

Parameters

- **phase** (*str*) – name of the phase to be set to the status entered
- **amount** (*float*) – phase fraction amount of that phase (0.0 < amount < 1.0)

set_phase_fixed(*phase*, *amount*)
Sets a phase to the status FIXED.

Parameters

- **phase** (*str*) – name of the phase to be set to the status FIXED
- **amount** (*float*) – phase fraction amount of that phase (0.0 < amount < 1.0)

set_phase_suspended(*phase*)
Sets a phase to the status SUSPENDED.

Parameters **phase** (*str*) – name of the phase to be set to the status SUSPENDED

set_temperature(*temp*)
Sets the temperature in the calculator.

Parameters **temp** (*float*) – new temperature (in K)

suspend_all_phases()
Sets all phases in the present calculator to the status SUSPENDED.

7.4 ThermoCalcSystem

class ThermoCalcSystem.**SpecialListMarkers**
Bases: object

Placeholders for special list elements that are locale-dependent. They will be provided by UI list components if a special marker has been selected.

ANY_LIST_MARKER = u'ANY'

NONE_LIST_MARKER = u'NONE'

class ThermoCalcSystem.**ThermoCalcSystem**(*model_utils*)
Provides all helper functionality required within the property models to access the Thermo-Calc / GUI functionality

Constructor. Note: This class should never be instantiated by the model, it will always be provided by Thermo-Calc with the correct and updated data.

Parameters **model_utils** (*se.thermocalc.unite.modelcalculator.domain.ModelUtils*) – Internal model utils object from the Thermo-Calc property model core

get_components()
Obtains all components currently set in the system.

Returns all defined components :rtype: list of str

get_dependent_component()
Obtains the currently set dependent component of the system, i.e. that component that is typically the main element of an alloy, for example Fe or Ni.

Returns the dependent component

Return type str

get_non_dependent_components()
Obtains the curently set non-dependent components of the system, i.e. all components except the dependent "main" component.

Returns the non-dependent components

Return type list of str

get_object_from_previous_evaluation(*key*)
Retrieves a stored object from a previous evalulation.

Parameters **key** (*str*) – key of the object

Returns the stored object

Return type object

get_selected_phases()
Obtains all currently selected phases in the system.

Returns the names of all selected phases

Return type list of str

get_ui_boolean_value(*component_id*)
Obtains the value from the specified checkbox.

Parameters **component_id** (*str*) – ID of the checkbox

Returns setting of the checkbox

Return type bool

get_ui_composition_value(*component_id*)
Obtains the composition from the specified composition UI component.

Parameters **component_id** (*str*) – ID of the composition UI component

Returns composition (in mole or mass fraction), note that the input unit of the UI is specified in the model panel. If required, the composition is automatically converted to fractions.

Return type float

get_ui_double_value(*component_id*)
Obtains the value from the specified UI component.

Parameters **component_id** (*str*) – ID of the UI component

Returns value

Return type float

get_ui_list_value(*component_id*)
Obtains the selected entry from a UI component list. If a special element (such as ANY, NONE, ...) is selected, the corresponding locale-independent placeholder is provided.

Parameters **component_id** (*str*) – ID of the list UI component

Returns selected entry

Return type str

get_ui_temperature_value(*component_id*)
Obtains the temperature from the specified temperature UI component.

Parameters **component_id** (*str*) – ID of the temperature UI component

Returns temperature (in K), note that input unit of the UI is specified in the model panel. If required, the temperature is automatically converted to K.

Return type float

is_object_saved(*key*)
Determines if an object is stored for the specified key.

Parameters **key** (*str*) – object key

Returns true if an object has been saved for the specified key, false otherwise.

Return type bool

is_phase_gas(*phase_name*)
Determines if a phase is a gas phase.

Parameters **phase_name** (*str*) – name of the phase

Returns true if the phase is a gas phase, false otherwise

Return type bool

is_phase_liquid(*phase_name*)
Determines if a phase is a liquid phase.

Parameters **phase_name** (*str*) – name of the phase

Returns true if the phase is a liquid phase, false otherwise

Return type bool

log_debug(*s*, **arg*)
Logs a debug level message to the Thermo-Calc log message window. Example: self._thermocalc_system.log_debug("Phase {} is invalid", phase_name)

Parameters

- **s** (*str*) – log message
- **arg** (*any*) – string arguments, can be omitted

log_error(*s*, **arg*)
Logs an error level message to the Thermo-Calc log message window. Example: self._thermocalc_system.log_error("Phase {} is invalid", phase_name)

Parameters

- **s** (*str*) – log message
- **arg** (*any*) – string arguments, can be omitted

log_info(*s*, **arg*)
Logs an info level message to the Thermo-Calc log message window. Example: self._thermocalc_system.log_info("Phase {} is invalid", phase_name)

Parameters

- **s** (*str*) – log message
- **arg** (*any*) – string arguments, can be omitted

log_trace(*s*, **arg*)
Logs a trace level message to the Thermo-Calc log message window. Example: self._thermocalc_system.log_trace("Phase {} is invalid", phase_name)

Parameters

- **s** (*str*) – log message
- **arg** (*any*) – string arguments, can be omitted

log_warning(*s*, **arg*)
Logs a warning level message to the Thermo-Calc log message window. Example: self._thermocalc_system.log_debug("Phase {} is invalid", phase_name)

Parameters

- **s** (*str*) – log message
- **arg** (*any*) – string arguments, can be omitted

num_saved_objects()
Determines the number of objects stored between evalulations.

Returns number of objects stored

Return type int

save_object_for_next_evaluation(*key*, *object_to_be_saved*)
Stores any object between evaluations and make it available for the next evalulation. This allows to overcome the limited lifetime of the ThermoCalcCalculator object.

Parameters

- **key** (*str*) – key of the object for later access
- **object_to_be_saved** (*object*) – object to be stored

set_result_quantity_value(*quantity_id*, *value*)
Sets the value of a previously defined result quantity for further usage in the Thermo-Calc property model core for plotting, ...

Parameters

- **quantity_id** (*str*) – unique ID of the result quantity
- **value** (*float*) – value to be set (use float('NaN') in case of an invalid result)

ThermoCalcSystem.**create_boolean_ui_component**(*component_id*, *name*, *description*, *initial_setting*)
Creates a UI checkbox component for a boolean value. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel

- **description** (*str*) – additional description of the component
- **initial_setting** (*bool*) – initial setting of the checkbox

Returns the created component

Return type *UIBooleanComponent*

ThermoCalcSystem.**create_composition_ui_component**(*component_id*, *name*, *description*, *initial_composition*)

Creates a UI text field component for a composition value. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **initial_composition** (*float*) – initial composition to be set in the text field (unit defined by the user in the Thermo-Calc system)

Returns the created component

Return type *UICompositionComponent*

ThermoCalcSystem.**create_condition_list_ui_component**(*component_id*, *name*, *description*)

Creates a UI list component for all conditions defined in the system. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component

Returns the created component

Return type *UIConditionListComponent*

ThermoCalcSystem.**create_energy_quantity**(*quantity_id*, *description*)

Creates a energy result quantity (in J). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

- **quantity_id** (*str*) – unique ID of the result quantity
- **description** (*str*) – additional description of the result quantity

Returns the created result quantity

Return type *ResultQuantity*

ThermoCalcSystem.**create_float_ui_component**(*component_id*, *name*, *description*, *value*)

Creates a UI text field component for a real number. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel

- **description** (*str*) – additional description of the component
- **value** (*float*) – initial setting of the text field

Returns the created component

Return type *UIFloatComponent*

ThermoCalcSystem.**create_general_quantity**(*quantity_id*, *description*)

Creates a general result quantity that can contain any type of result (without a unit). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

- **quantity_id** (*str*) – unique ID of the result quantity
- **description** (*str*) – additional description of the result quantity

Returns the created result quantity

Return type *ResultQuantity*

ThermoCalcSystem.**create_list_ui_component**(*component_id*, *name*, *description*, *entry_list*, *selected_entry=''*)

Creates a UI list component for string entries. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **entry_list** (*list of str*) – entries of the list
- **selected_entry** (*str*) – entry to be initially selected. If omitted, by default the first element is selected.

Returns the created component

Return type *UIGeneralListComponent*

ThermoCalcSystem.**create_phase_list_ui_component**(*component_id*, *name*, *description*, *default_phase=''*, *any_marker=False*)

Creates a UI list component for all phases defined in the system. It is possible to select a default phase that is supposed to be the **expected phase selection** for that list. The value of that component can later be accessed during the model evaluation.

A **default** phase is the phase that is initially selected and re-selected as soon as a currently selected phase is removed. If the default phase is not available, a NONE-marker will be created and used instead of the default phase. A typical use case for the default phase setting is a phase list that expects to contain the LIQUID-phase of a system.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **default_phase** (*str*) – default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.

- **any_marker** (*bool*) – defines if an entry "ANY PHASE" should be added to the phase list, if set to true this overrides any default phase setting

Returns the created component

Return type *UIPhaseListComponent*

ThermoCalcSystem.**create_temperature_quantity**(*quantity_id*, *description*)

Creates a temperature result quantity (in K). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

- **quantity_id** (*str*) – unique ID of the result quantity
- **description** (*str*) – additional description of the result quantity

Returns the created result quantity

Return type *ResultQuantity*

ThermoCalcSystem.**create_temperature_ui_component**(*component_id*, *name*, *description*, *initial_temp*)

Creates a UI text field component for a temperature value. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **initial_temp** (*float*) – initial temperature to be set in the text field (unit defined by the user in the Thermo-Calc system)

Returns the created component

Return type *UITemperatureComponent*

7.5 UIBooleanComponent

class UIBooleanComponent.**UIBooleanComponent**(*component_id*, *name*, *description*, *setting*)

Bases: *UIComponent.UIComponent*

Checkbox UI component of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **setting** (*bool*) – initial setting of the checkbox

get_setting()

Obtains the initial setting of the checkbox.

Returns initial setting of the checkbox

Return type bool

7.6 UIComponent

class UIComponent.**UIComponent**(*component_id*, *name*, *description*)

Bases: object

Base class for all UI components of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component

get_description()

Obtains the additional description of the component.

Type additional description of the component

Return type str

get_id()

Obtains the unique ID of the component.

Returns unique ID of the component

Return type str

get_name()

Obtains the name of the component.

Type name of the component, will be presented in the model panel

Return type str

7.7 UICompositionComponent

class UICompositionComponent.**UICompositionComponent**(*component_id*, *name*, *description*, *composition*)

Bases: *UIComponent.UIComponent*

Composition text field UI component of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **composition** (*float*) – initial composition to be set in the text field (unit defined by the user in the Thermo-Calc system)

get_composition()
Obtains the initial composition set in the text field in the UI.

Returns initial composition to be set in the text field (unit defined by the user in the Thermo-Calc system)

Return type float

7.8 UIConditionListComponent

class UIConditionListComponent.**UIConditionListComponent**(*component_id*, *name*, *description*)

Bases: *UIComponent.UIComponent*

System condition list UI component of the model panel

Constructor. All conditions are set automatically.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component

7.9 UIFloatComponent

class UIFloatComponent.**UIFloatComponent**(*component_id*, *name*, *description*, *value*)

Bases: *UIComponent.UIComponent*

General real value text field UI component of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **value** (*float*) – initial setting of the text field

get_value()
Obtains the initial setting of the text field.

Returns initial setting of the text field

7.10 UIGeneralListComponent

class UIGeneralListComponent.**UIGeneralListComponent**(*component_id*, *name*, *description*, *content*, *selected_entry=''*)

Bases: *UIComponent.UIComponent*

General list UI component of the model panel that can contain any strings

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **content** (*list of tuple of str*) – entries of the list, they need to contain a locale-independent ID and a localized content string, for example: [("ENTRY_1_ID", "entry 1"), (ENTRY_2_ID", "entry 2")]
- **selected_entry** (*str*) – entry to be initially selected. If omitted, by default the first element is selected.

connect_component_visibility(*dependent_component_id*, *selected_item_to_set_visible*)
Connects the visibility of any other UI component of the model panel to the selection of a certain entry of the list.

Parameters

- **dependent_component_id** (*str*) – ID of the UI element to be dependent on the chosen element
- **selected_item_to_set_visible** (*str*) – entry (locale independent ID) of the list to be chosen to set the dependent component visible

get_content()
Obtains the entries of the list.

Returns entries of the list, they need to contain a locale-independent ID and a localized content string, for example: [("ENTRY_1_ID", "entry 1"), (ENTRY_2_ID", "entry 2")]

Return type list of tuple of str

get_dependent_components()
Obtains a dictionary containing all UI elements currently connected regarding their visibility.

Returns all UI elements currently connected (key: dependent component ID, value: required list entry to set it visible)

Return type dict(str, str)

get_selected_entry()
Obtains the initially selected entry.

Returns initially selected entry. If empty, the first element is selected.

Return type str

remove_component_visibility(*dependent_component_id*)
Removes the visiblity connection to a UI component that has been previously connected.

Parameters **dependent_component_id** (*str*) – ID of the previously connection UI element

7.11 UIPhaseListComponent

class UIPhaseListComponent.**UIPhaseListComponent**(*component_id*, *name*, *description*, *default_phase=''*, *any_marker_setting=False*)

Bases: *UIComponent.UIComponent*

Phase list UI component of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **default_phase** (*str*) – default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.
- **any_marker_setting** (*bool*) – defines if an entry "ANY PHASE" should be added to the phase list, if set to true this overrides any default phase setting

get_any_marker_setting()
Obtains the setting if any entry "ANY PHASE" is added to the phase list.

Returns if an entry "ANY PHASE" is added to the phase list, if set to true this overrides any default phase setting

Return type bool

get_default_phase()
Obtains the default phase.

Returns default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.

Return type str

7.12 UITemperatureComponent

class UITemperatureComponent.**UITemperatureComponent**(*component_id*, *name*, *description*, *temp*)

Bases: *UIComponent.UIComponent*

Temperature value text field UI component of the model panel

Constructor.

Parameters

- **component_id** (*str*) – unique ID of the component
- **name** (*str*) – name of the component, will be presented in the model panel
- **description** (*str*) – additional description of the component
- **temp** (*float*) – initial temperature to be set in the text field (unit defined by the user in the Thermo-Calc system)

get_temp()
Obtains the initial temperature set in the text field.

Returns initial temperature to be set in the text field (unit defined by the user in the Thermo-Calc system)

Return type float

7.13 JavaPropertyManager

class JavaPropertyManager.**JavaPropertyManager**(*class_name*, *resource_dir_path=None*)

Bases: object

Providing localized strings based on Java property file bundles.

Constructor. It is expected that Java *.properties files of the format "class_name_sv.properties" are provided. A default file with the name "class_name.properties" containing the english strings is required.

Parameters

- **class_name** (*str*) – name of the Python class for which the properties need to be provided
- **resource_dir_path** (*str*) – path to directory containing the resource file, if omitted the resources are expected to be loacted in the subdirectory RESOURCES_DIR of the Thermo-Calc program directory

JAVA_PROPERTY_EXTENSION = '.properties'

RESOURCES_DIR = 'jythonModelsResources'

get(*property_key*)

Obtains a property for the class specified in the constructor.

Parameters **property_key** (*str*) – property key (for example "label.text", . . .)

Returns localized string

Return type str