Projet en informatique pour les sciences humaines

Johan Cuda

4 juin 2024

Résumé

Ce projet reprend le travail de *threading* des widgets de Orange Textable effectué par Antonin Schnyder en proposant de remonter l'architecture de *threading* d'un étage dans la hiérarchie de classes de Textable.

Table des matières

1 Introduction

1.1 Contexte

Textable est un add-on du logiciel Orange qui permet d'analyser des textes de manière visuelle[1]. Il est composé de widgets développés en Python qui ont récemment été modifiés par Antonin Schnyder pour optimiser leur processus. En effet, l'interface des widgets se bloquait pendant les traitements de données et leurs performances pouvaient être optimisées, c'est pourquoi Antonin Schnyder a modifié ces widgets pour ajouter une logique de *threading*[2] qui suit les recom-

1. Xanthos, Aris (2014). Textable : programmation visuelle pour l'analyse de données textuelles. In Actes des 12èmes Journées internationales d'analyse statistique des données textuelles (JADT 2014), pp. 691-703. Read online
2. À ce sujet voir An Intro to Threading in Python.

mandations de Orange[3].

Dans ce travail, nous proposons de partir du travail effectué par Antonin Schnyder en remontant d'un étage – dans la hiérarchie de classes de Textable – la logique de *threading*. Nous avons identifié qu'une grande partie du code qui permet la mise en place du *threading* se répète dans les widgets, nous avons donc modifié leur architecture pour faire en sorte que les éléments liés au *threading* soient hérités au travers de la super-classe **OWTextableBaseWidget**.

Nous commencerons par discuter rapidement de ce qu'est le *threading*, puis par expliciter la structure du add-on *Textable* ainsi que d'un widget typique pour mieux comprendre leur fonctionnement. Ensuite, nous listerons les éléments que nous avons identifiés comme possiblement remontables, en mentionnant ceux qui ont directement pu être remonter et en décrivant les modifications effectuées sur ceux qui ne le pouvaient pas. Ce faisant, nous discuterons des divers problèmes rencontrés pendant l'implémentation. Nous termienrons ce rapport en proposant un tutoriel de développement de widget qui permet d'exploiter cette nouvelle architecture ainsi que d'assurer en partie l'uniformité des futures widgets qui seront ajoutés à *Textable*. Ce tutoriel reprendra en partie le tutoriel fourni par Antonin Schnyder.

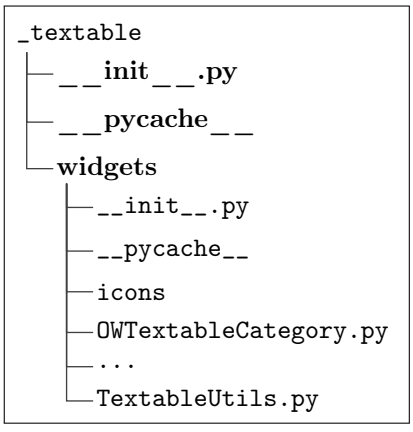
1.2 Qu'est-ce que le *threading* ?

Ce projet n'a pas pour but d'expliquer ce qu'est le *threading* mais il est quand même nécessaire de fournir une explication succincte du concept pour mieux appréhender le travail proposé dans ce rapport. L'idée est de répartir le travail entre différent *threads* et de simuler une forme de *multiprocessing*[4]. À la création de chaque widget, un *main thread* est créé et s'occupera de différents *worker threads* qui seront créés par chaque widget. Le *multithreading* consiste ensuite à faire s'enchaîner l'exécution des *worker threads* pour simuler une sorte de parallélisme des processus. Dans le cas particulier de *Textable*, les processus qui sont voués à être exécutés dans un *worker thread* sont les appels aux fonctions de LTTL par exemple. Ce système permet donc d'améliorer les performances d'un *software* mais aussi dans notre cas précis de permettre à l'*User interface* (UI) de notre widget de ne pas se bloquer pendant ses calculs. Ceci est extrêmement utile en considérant que suivant la taille des corpus analysés avec *Textable*, les calculs peuvent prendre plusieurs minutes, il est donc pratique de pouvoir annuler le processus en cours grâce au bouton `Cancel` par exemple.

2 Structure de *Textable* et des widgets

2.1 Structure de `TextableUtils.py` et OWTextableBaseWidget

Textable peut être visulisé de la manière suivante :



Pour bien comprendre la structure de *Textable* nous allons commencer par considérer le script `TextableUtils.py` (voir Figure 1). Celui-ci contient plusieurs classes qui définissent des éléments utilisables par tous les widgets comme `InfoBox` ou `SendButton`. La classe qui nous intéressera le plus sera bien entendu **OWTextableBaseWidget** : elle est déjà héritée par tous les widgets de

3. Le tutoriel de Orange à ce propos est disponible ici.
4. Sur la différence entre m*multiprocessing* et *multithreading* voir ce lien.

Textable et a donc été rapidement identifiée comme la classe qui recevrait les modifications relatives au *threading*.

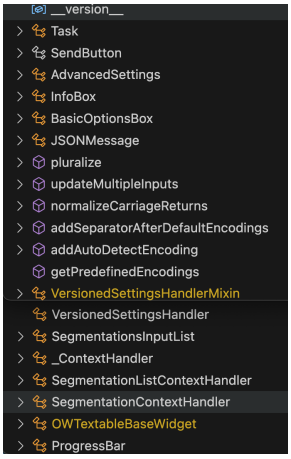


FIGURE 1 – Structure de `TextableUtils.py`

2.2 Structure d'un widget type

Les widgets ont quant à eu une structure assez uniforme et représentent chacun une classe spécifique qui hérite de la classe **OWTextableBaseWidget** (par exemple chaque widget `Count` est une instance de la classe **OWTextableCount**). Ils sont toujours formés des éléments suivants :

1. Imports divers
2. Déclaration d'une série d'*attributs de classes*
3. Déclaration, dans le constructeur de la classe, de tous les *attributs d'instance* ainsi que des éléments d'interface du widget
4. Déclaration des diverses méthodes propres au processus du widget

La structure générale peut varier selon les widgets mais nous avons ici les éléments principaux.

3 Analyse des widgets et implémentation

Nous avons commencé – sur recommandation d'Aris Xantos – par analyser les deux widgets `Count` et `Preprocess` car ils sont assez représentatifs de deux types de widgets courants dans *Textable*. Nous avons donc systématiquement repéré tous les éléments liés au *threading* dans ces deux widgets pour ensuite évaluer s'ils étaient remontables dans la classe parente **OWTextableBaseWidget** ou non. Nous avons continué par évaluer si nos observations se prolongeait aux autres widgets. Dans la suite de cette section, nous adopterons les conventions suivantes :

— `Nom_Du_Widget_Thread` pour nommer la version du widget développée par Antonin Schnyder (par exemple `ConvertThread` dans le cas du widget `Convert`)
— `Nom_Du_Widget_ThreadJohan` pour nommer la version du widget développée dans le cadre de ce rapport (par exemple `PreprocessThreadJohan` dans le cas de `Preprocess`)
— "`# ...`" pour représenter une ellipse dans un extrait de code

3.1 Méthodes et attributs remontés mais non-modifiés

Nous allons ici étudier tous les éléments présents dans les widgets créés par Antonin Schnyder qui ont pu être déplacés dans `TextableUtils.py` sans modifications particulières. Nous fournirons aussi l'extrait de code déplacé.

3.1.1 Signaux et slots

Pour afficher certaines informations sur l'interface (par exemple des messages dans l'`InfoBox` ou l'avancée de la `ProgressBar`) pendant le processus d'un widget *threadé*, il est nécessaire de définir des signaux (comme *attributs de classe*) et de les connecter à des slots (qui sont des méthodes qui

seront explicitées dans la prochaine section), ce qui permettra aux *worker threads* de communiquer avec l'interface du widget d'une manière *thread safe* (i.e. qui prend en compte l'exécution des *threads* et qui ne tente pas de forcer l'*update* de l'UI).

Exemple du code présent dans le widget `CategoryThread` :

```
class OWTextableCategory(OWTextableBaseWidget):
    """Orange widget for extracting content or annotation information"""

    # ...

    # Signals
    signal_prog = pyqtSignal((int, bool))       # Progress bar (value, init)
    signal_text = pyqtSignal((str, str))        # Text label (text, infotype)
    signal_cancel_button = pyqtSignal(bool)     # Allow to Deactivate cancel
                                                # button from worker thread

    # ...

    def __init__(self):

        """Initialize a Category widget"""

        # ...

        # Connect signals to slots
        self.signal_prog.connect(self.update_progress_bar)
        self.signal_text.connect(self.update_infobox)
        self.signal_cancel_button.connect(self.disable_cancel_button)
```

Tous les widgets n'ont pas forcément besoin de tous les signaux [5], nous avons pourtant décidé de remonter les trois signaux comme attributs de la classe **OWTextableBaseWidget** en partant du principe que de faire hériter les trois signaux à tous les widgets ne crée pas de problème particulier et permet de standardiser le code de tous les widgets. Le résultat est donc le code suivant dans

`TextableUtils.py` :

```
class OWTextableBaseWidget(widget.OWWidget):
    """
    A base widget for other concrete orange-textable widgets.

    Defines s common `uuid` setting which is required for all Textable
    widgets.

    """

    # ...

    # Signals
    signal_prog = pyqtSignal((int, bool))       # Progress bar (value, init)
    signal_text = pyqtSignal((str, str))        # Text label (text, infotype)
    signal_cancel_button = pyqtSignal(bool)     # Allow to Deactivate cancel
                                                # button from worker thread

    # ...

    def __init__(self, *args, **kwargs):

        """Initialize a Category widget"""
```

5. À ce sujet voir le tutoriel D'antonin Schnyder.

```
        # ...

        # Connect signals to slots
        self.signal_prog.connect(self.update_progress_bar)
        self.signal_text.connect(self.update_infobox)
        self.signal_cancel_button.connect(self.disable_cancel_button)
```

3.1.2 Méthodes liées aux signaux

Lorsqu'un signal est émis depuis un *thread* (par exemple pour faire avancer la `ProgressBar`), le *slot* – une méthode particulière reliée au signal et décorée par `@pyqtSlot()` – correspondant est appelé. Dans la version des widgets d'Antonin Schnyder, il était nécessaire de définir autant de slots que de signaux.
Exemple de slot pour mettre à jour la `ProgressBar` :

```
@pyqtSlot(int, bool)
def update_progress_bar(self, val, init):
    """ Update progress bar in a thread-safe manner """
    # Re-init progress bar, if needed
    if init:
        self.progressBarInit()

    # Update progress bar
    if val >= 100:
        self.progressBarFinished() # Finish progress bar
    elif val < 0:
        self.progressBarSet(0)
    else:
        self.progressBarSet(val)
```

Comme dans le cas de signaux, nous avons décidé de remonter les trois slots `update_progress_bar`, `update_infobox` et `disable_cancel_button` dans `TextableUtils.py` même s'ils ne sont pas tous utilisés par chaque widget. En remontant ces éléments, nous fournissons aussi une toolbox pour la création de futures widgets, avec les signaux et slots correspondants déjà implémentés dans la classe **OWTextableBaseWidget**.

3.1.3 `Task` et `ThreadExecutor`

Tous les widgets ont à la fin de leur constructeur la série de trois opérations suivante :

```
# Threading
self._task = None
self._executor = ThreadExecutor()
self.cancel_operation = False
```

Ces opérations permettent de gérer les *threads* et de définir la *variable d'instance* `cancel_operation` qui est utilisée pour annuler le processus d'un widget. Elles ont été remontées dans le constructeur de la classe **OWTextableBaseWidget** :

```
class OWTextableBaseWidget(widget.OWWidget):
    # ...

    def __init__(self, *args, **kwargs):

        """Initialize a Category widget"""

        # ...

        # Threading
        self._task = None  # type: Optional[Task]
        self._executor = ThreadExecutor()
```

```
        self.cancel_operation = False

        # Connect signals to slots
        self.signal_prog.connect(self.update_progress_bar)
        self.signal_text.connect(self.update_infobox)
        self.signal_cancel_button.connect(self.disable_cancel_button)
```

3.1.4 Méthode `cancel_manually()`

La méthode `cancel_manually()` était présente à l'identique dans tous les widgets *threadés*, nous l'avons donc simplement remontée comme méthode de la classe **OWTextableBaseWidget**. Cette méthode est un *wrapper* de la méthode `cancel()` qui est appelée lorsque l'utilisateur.ice appuie sur bouton "cancel".

```
    def cancel_manually(self):
        """ Wrapper of cancel() method,
        used for manual cancellations """
        self.cancel(manualCancel=True)
```

3.1.5 Importation des modules

Comme nous avons déplacé la plupart des éléments en lien avec le *threading*, nous devons aussi modifier les importations de modules au début de chaque widget et de `TextableUtils.py` pour faire en sorte que tout fonctionne correctement.

Chaque widget conserve donc l'importation suivante ajoutée par Antonin Schnyder :

```
from functools import partial
```

Elle sera utilisée plus bas dans ce travail pour les processus principaux des widgets.

`TextableUtils.py` récupère quant à lui les importations suivantes en lien avec le *threading* :

```
# Threading
from AnyQt.QtCore import QThread, pyqtSlot, pyqtSignal
from Orange.widgets.utils.concurrent import ThreadExecutor, FutureWatcher
from functools import partial
```

3.2 Méthodes et attributs remontés et modifiés

Dans cette section, nous détaillerons les éléments que nous avons identifiés comme remontables mais qui ont demandé des modifications pour être implémentés dans la classe **OWTextableBaseWidget**.

3.2.1 Méthode `cancel()`

La méthode `cancel()` est appelée lorsque l'utilisateur.ice annule l'opération en cours. Le but de cette fonction est de :

— Annuler les boucles de calculs en cours dans le *thread* correspondant
— Annuler la tâche (du *thread* en cours)
— Déconnecter la tâche du *slot* qui surveille sa complétion [6]
— Envoyer "None" dans tous les *outputs* du widget
— Envoyer un message d'annulation à l'utilisateur.ice
— Rendre l'UI du widget à nouveau modifiable [7]

Dans le cas du widget `CountThread`, la méthode `cancel()` se présente comme suit :

```
    def cancel(self, manualCancel=False):
        # Make loop break in LTTL/ProcessorThread.py
        self.cancel_operation = True
```

6. Nous reviendrons sur ce point dans les sections consacrées à `sendData()` et `task_finished()`.
7. Nous reviendrons sur ce point dans la section sur `manageGuiVisibility()`.

```
        # Cancel current task
        if self._task is not None:
            self._task.cancel()
            assert self._task.future.done()

            # Disconnect slot
            self._task.watcher.done.disconnect(self._task_finished)
            self._task = None

            # Send None to output
            self.send('Textable pivot crosstab', None) # AS 10.2023: removed self
            self.send('Orange table', None) # AS 10.2023: removed self

        # If cancelled manually
        if manualCancel:
            self.infoBox.setText(u'Operation cancelled by user.', 'warning')

        # Manage GUI visibility
        self.manageGuiVisibility(False) # Processing done/cancelled
```

Comme cet extrait de code était présent dans tous les widgets presque à l'identique, nous avons rapidement identifié qu'il devait être remonté. Pourtant un élément posait problème : les widgets n'ayant pas tous les mêmes *outputs*, la partie qui envoie "None" à ces derniers n'est pas identique.

Par exemple dans le cas du widget `SelectThread` :

```
# Send None to output
self.send('Selected data', None) # AS 10.2023: removed self
self.send('Discarded data', None) # AS 10.2023: removed self
```

Pour remédier à ce problème, nous avons implémenté une nouvelle méthode nommée `sendNoneToOutputs()` qui envoie automatiquement "None" à chaque output du widget en utilisant l'*attribut de classe* "outputs". Un appel à cette méthode vient donc remplacer les envois individuels à chaque *output*. Le code finalement placé dans `TextableUtils.py` est comme suit :

```
def sendNoneToOutputs(self):
    # Sends none to all widget outputs
    for output in self.outputs:
        self.send(output.name, None)

# ...

def cancel(self, manualCancel=False):

    # ...

        # Send None to output
        self.sendNoneToOutputs()

    if manualCancel:
        self.infoBox.setText(u'Operation cancelled by user.', 'warning')

    # Manage GUI visibility
    if manualCancel:
        QTimer.singleShot(250, partial(self.manageGuiVisibility, processing=False)) # Process
    else:
        QTimer.singleShot(250, partial(self.manageGuiVisibility)) # Processing done/cancelled
```

La dernière partie de cette méthode s'occupant de l'affichage du widget a été récupérée du widget `SelectThread`. Ces lignes ajoutent un petit timer nécessaire dans le cas de ce widget particulier, nous l'avons donc ajouté à la méthode `cancel()` de `TextableUils.py` après discussion avec Antonin Schnyder.

3.2.2 Méthode `manageGuiVisibility()`

La méthode `manageGuiVisibility()` permet de désactiver ou d'activer certaines parties de l'interface suivant ce que fait le widget (par exemple, les paramètres du widget sont désactivés pendant les calculs, ou encore le bouton `cancel` est grisé pendant sa phase de configuration). Présente dans chaque widget, elle diffère pourtant selon les occurrences. En effet, les widgets ne sont pas tous construits de la même manière et leur interface peut varier. Il est évident que vu le nombre de lignes redondantes que cette méthode comporte, il était nécessaire de la remonter dans `TextableUtils.py`.

Extrait de la méthode `manageGuiVisibility.py` dans le widget `SegmentThread` :

```
def manageGuiVisibility(self, processing=False):
    """ Update GUI and make available (or not) elements
    while the thread task is running in the background """

    # Thread currently running, freeze the GUI
    if processing:
        # Buttons and layout
        self.sendButton.cancelButton.setDisabled(0) # Cancel: ENABLED
        self.sendButton.mainButton.setDisabled(1) # Send: DISABLED
        self.sendButton.autoSendCheckbox.setDisabled(1) # Send automatically: DISABLED
        self.basicRegexBox.setDisabled(1) # Basic regex box: DISABLED
        self.regexBox.setDisabled(1) # Regex box (advanced settings): DISABLED
        self.optionsBox.setDisabled(1) # Option box (advanced settings): DISABLED
        self.advancedSettings.checkbox.setDisabled(1) # Advanced options checkbox: DISABLED

    # Thread done or not running, unfreeze the GUI
    else:
        # If "Send automatically" is disabled, reactivate "Send" button
        if not self.sendButton.autoSendCheckbox.isChecked():
            self.sendButton.mainButton.setDisabled(0) # Send: ENABLED
        # Buttons and layout
        self.sendButton.cancelButton.setDisabled(1) # Cancel: DISABLED
        self.sendButton.autoSendCheckbox.setDisabled(0) # Send automatically: ENABLED
        self.basicRegexBox.setDisabled(0) # Basic regex box: ENABLED
        self.regexBox.setDisabled(0) # Regex box (advanced settings): ENABLED
        self.optionsBox.setDisabled(0) # Option box (advanced settings): ENABLED
        self.advancedSettings.checkbox.setDisabled(0) # Advanced options checkbox: ENABLED
        self.cancel_operation = False # Restore to default
        self.signal_prog.emit(100, False) # 100% and do not re-init
        self.sendButton.resetSettingsChangedFlag()
        self.updateGUI()
```

Pour remonter cette méthode, nous avons commencé par identifier les parties du code qui se répétaient. Nous avons déterminé que les éléments concernant les parties communes à (presque) tous les widgets étaient identiques, c'est à dire les éléments concernant : le bouton `cancel`, le bouton `main` et la checkbox `autoSend`. La fin de la méthode, qui remet le widget dans son état initial, est aussi identique. Les éléments variants, quant à eux, avaient la particularité d'être tous insérés dans un partie de l'interface du widget nommée "controlArea". Ces éléments représentent des parties particulières de l'UI comme les `AdvancedSettings` ou l'`optionsBox`.

Exemple de déclaration d'un élément variant (ici `optionsBox`) :

```
# (Advanced) options box...
    self.optionsBox = gui.widgetBox(
        widget=self.controlArea,
        box=u'Options',
        orientation='vertical',
        addSpace=False,
    )
```

La solution que nous avons choisie pour généraliser `manageGuiVisibility()` est de créer une nouvelle méthode `create_widgetbox()` qui fait office de *wrapper* de la méthode originale

`widgetBox` utilisée dans l'exemple ci-dessus. Ce wrapper ajoute l'élément d'UI – au moment de sa création – à une nouvelle variable d'*instance* de **OWTextableBaseWidget** nommée `guiElements`. Cette étape nous permet de savoir en tout temps quels éléments de l'UI d'un widget doivent être activés/désactivés dans la méthode `manageGuiVisibility()`. Nous avons aussi créé une méthode de création des `AdvancedSettings` basée sur le même principe que `create_widgetbox()` que nous avons nommé `create_advancedSettings()`.

Le code alors remonté dans `TextableUtils.py` se présente comme suit :

```
class OWTextableBaseWidget(widget.OWWidget):

# ...

    def __init__(self, *args, **kwargs):

    # ...

        # Attribute to handle GUI visibility
        self.guiElements = []

    # ...

    def manageGuiVisibility(self, processing=False):
        # Update GUI and make available (or not) elements
        #while the thread task is running in background

        # Thread currently running, freeze the GUI
        if processing:
            for guiElement in self.guiElements:
                if guiElement.__class__.__name__ == "AdvancedSettings":
                    guiElement.checkbox.setDisabled(1)
                else:
                    guiElement.setDisabled(1)

            #self.optionsBox.setDisabled(1) # Options: DISABLED
            self.sendButton.mainButton.setDisabled(1) # Send button: DISABLED
            self.sendButton.cancelButton.setDisabled(0) # Cancel button: ENABLED
            self.sendButton.autoSendCheckbox.setDisabled(1) # Send automatically: DISABLED

        # Thread done or not running, unfreeze the GUI
        else:
            # If "Send automatically" is disabled, reactivate "Send" button
            if not self.sendButton.autoSendCheckbox.isChecked():
                self.sendButton.mainButton.setDisabled(0) # Send: ENABLED
            # Other buttons and layout
            for guiElement in self.guiElements:
                if guiElement.__class__.__name__ == "AdvancedSettings":
                    guiElement.checkbox.setDisabled(0)
                else:
                    guiElement.setDisabled(0)
            #self.optionsBox.setDisabled(0) # Options: ENABLED
            self.sendButton.cancelButton.setDisabled(1) # Cancel button: DISABLED
            self.sendButton.autoSendCheckbox.setDisabled(0) # Send automatically: ENABLED
            self.cancel_operation = False
            self.signal_prog.emit(100, False) # 100% and do not re-init
            self.sendButton.resetSettingsChangedFlag()
            self.updateGUI()
```

La nouvelle méthode itère donc sur tous les éléments de `guiElements` pour les activer/ désactiver, avec une gestion spéciale pour le cas de la checkbox des `Advanced settings`.

3.2.3 Méthodes `sendData()` et `threading()`

La méthode `sendData()` (à travers les différents widgets) a été grandement modifiée par Antonin Schnyder pour conserver toutes les étapes de vérifications des *inputs* du widget et déléguer tous les calculs à un *worker thread*. Effectivement, cela se traduit par différentes vérifications quant aux *tasks* en cours, puis par le démarrage du *threading* (le démarrage du *threading* relie ici la fonction `threaded_function` définie dans `sendData()` à un `_executer` qui s'occupera de la démarrer dès que possible) et l'arrêt de certaines parties de l'interface :

```
def sendData(self):
    """(Have LTTL.Segmenter) perform the actual tokenization"""

    # ...

    # Threading ...

    # Cancel old tasks
    if self._task is not None:
        self.cancel()
    assert self._task is None

    self._task = task = Task()

    # ...

    # Restore to default
    self.cancel_operation = False

    # Threading start, future, and watcher
    task.future = self._executor.submit(threaded_function)
    task.watcher = FutureWatcher(task.future)
    task.watcher.done.connect(self._task_finished)

    # Manage GUI visibility
    self.manageGuiVisibility(True) # Processing
```

Nous avons remplacé ces bouts de codes redondants par l'appel à une nouvelle méthode de `TextableUtils.py` nommée `threading()` qui centralise ces diverses étapes.

Le code ci-dessus est donc remplacé par :

```
# Threading ...
    self.threading(threaded_function)
```

La méthode `threading()` est quant à elle implémentée dans `TextableUtils.py` :

```
def threading(self, threaded_function):
    """ Checks for tasks and start threading"""

    # Threading ...

    # Cancel old tasks
    if self._task is not None:
        self.cancel()
    assert self._task is None

    self.cancel_operation = False

    self._task = task = Task()

    # Threading start, future, and watcher
    task.future = self._executor.submit(threaded_function)
    task.watcher = FutureWatcher(task.future)
```

```
        task.watcher.done.connect(self.task_finished)

        # Manage GUI visibility
        self.manageGuiVisibility(True) # Processing
```

3.2.4 Méthode `task_finished()`

Cette méthode est sans aucun doute la partie la plus complexe de ce travail. Dans la version des widgets d'Antonin Schnyder, la méthode `sendData()` connectait le résultat du processus de la `threaded_function` à un *slot* exécutant une méthode nommée `_task_finished()` qui était chargée de traiter les données des processus du widget et des les envoyer aux différents *outputs*. Cette méthode se présentait par exemple dans le cas de `SegmentThread` de la manière suivante :

```
    @pyqtSlot(concurrent.futures.Future)
    def _task_finished(self, f):
        assert self.thread() is QThread.currentThread()
        assert self._task is not None
        assert self._task.future is f
        assert f.done()

        self._task = None

        try:
            # Data outputs
            try:
                segmented_data = f.result()
            # If operation was started again while processing,
            # f.result() is None and it raises a TypeError
            except TypeError:
                self.infoBox.setText(
                    u'Operation was cancelled.',
                    'warning'
                )
                self.send('Segmented data', None)
                self.manageGuiVisibility(False) # Processing done/cancelled!
                return

            # Processing results
            message = u'%i segment@p sent to output.' % len(segmented_data)
            message = pluralize(message, len(segmented_data))
            self.infoBox.setText(message)
            self.send('Segmented data', segmented_data) # AS 10.2023: removed self

        except IndexError:
            self.infoBox.setText(
                u'Reference to unmatched group in annotation key and/or value.',
                'error'
            )
            self.send('Segmented data', None) # AS 10.2023: removed self

        except Exception as ex:
            print(ex)

            self.infoBox.setText(
                u'Error while segmenting. Please verify your settings.',
                'error'
            )
            self.send('Segmented data', None) # AS 10.2023: removed self

        # Manage GUI visibility
        self.manageGuiVisibility(False) # Processing done/cancelled!
```

Nous pouvons donc repérer dans cette méthode :
— Une série d'assertion pour vérifier que le traitement est bien terminé
— Un `try` qui essaie de récupérer les résultats des processus et de les envoyer aux *outputs*
— Un `except` qui gère les erreurs possibles à la fin du traitement
— Un appel à `manageGuiVisibility()` pour réactiver l'interface du widget

Or, en observant les différentes méthodes décorées [8] `_task_finished()` des différents widgets, nous avons remarqué que les seuls éléments qui diffèrent sont le contenu du premier `try` ainsi que le type d'erreurs géré par le `except`. Nous avons donc décidé, pour pouvoir remonter une partie de ce code dans `TextableUtils.py` et alléger le contenu des widgets, de créer notre propre décorateur qui permet d'entourer la partie de traitement des données (propre à chaque widget) des textes et gestions d'erreurs assez généraux. Ce décorateur est hérité de **OWTextableBaseWidget** et permet de conserver uniquement la partie de `_task_finished()` concernant le traitement du résultat et l'envoi aux *outputs* dans le widget.

Concrètement, la méthode `_task_finished()` du widget est remplacée par une nouvelle méthode `task_finished()` (sans le premier *underscore*) :

```
@OWTextableBaseWidget.task_decorator
def task_finished(self, f):
    # Data outputs
    try:
        segmented_data = f.result()
    # If operation was started again while processing,
    # f.result() is None and it raises a TypeError
    except TypeError:
        self.infoBox.setText(
                u'Operation was cancelled.',
                'warning'
            )
        self.send('Segmented data', None)
        self.manageGuiVisibility(False) # Processing done/cancelled!
        return

    # Processing results
    message = u'%i segment@p sent to output.' % len(segmented_data)
    message = pluralize(message, len(segmented_data))
    self.infoBox.setText(message)
    self.send('Segmented data', segmented_data) # AS 10.2023: removed self
```

Cette dernière ne contient plus que le traitement des résultats et est décorée par un nouveau décorateur `task_decorator` implémenté dans la classe **OWTextableBaseWidget** :

```
def task_decorator(task_function):
        """ Decorator for the task_finished function """
        @pyqtSlot(concurrent.futures.Future)
        def _task_finished(self, f):
            assert self.thread() is QThread.currentThread()
            assert self._task is not None
            assert self._task.future is f
            assert f.done()

            self._task = None

            try:
                task_function(self, f)

            # Exceptions handling for different widgets
            except ValueError:
                self.infoBox.setText(
                message=u'Please make sure that input is well-formed XML.',
                state='error',
```

8. Au sujet des décorateurs Python, vous référer à cette page.

```
                )
                self.sendNoneToOutputs()

            except IndexError:
                self.infoBox.setText(
                u'Reference to unmatched group in annotation key and/or value.',
                'error'
            )
                self.sendNoneToOutputs()

            except KeyError:
                return

            except re.error as re_error:
                try:
                    message = u'Please enter a valid regex (error: %s).' % \
                            re_error.msg
                except AttributeError:
                    message = u'Please enter a valid regex.'
                self.infoBox.setText(message, 'error')
                self.sendNoneToOutputs()

            except Exception as ex:
                print(ex)
                self.sendNoneToOutputs()
                self.infoBox.setText(u'An error occured.', 'error')

            finally:
                # Manage GUI visibility
                self.manageGuiVisibility(False) # Processing done/cancelled
        return _task_finished
```

Toute la partie du code relative aux tests et aux gestions d'erreurs est donc remontée dans la classe **OWTextableBaseWidget** et permet une implémentation plus simple dans le widget.
Il reste cependant un point que nous n'avons pas encore abordé, celui de la gestion des erreurs. Ces dernières étaient, à l'origine, gérées par chaque widget dans sa méthode `_task_finished()` originale. Heureusement, les types d'erreurs de chaque widget étant différents, nous avons pu – apprès discussion avec Aris Xantos – tous les réunir dans le décorateur `task_decorator` en standardisant quelque peu les messages d'erreurs.

4 Tutoriel de création de widget

Nous allons proposer dans cette section une liste d'éléments qui doivent être présents dans un widget *threadé* de `Textable`. Ce tutoriel n'a pas pour vocation d'apprendre à son lectorat à créer un widget mais définit les éléments nécessaires pour obtenir un widget *threadé* et s'inspire et reprend des parties du tutoriel fourni par Antonin Schnyder à l'issu de son travail.

4.1 Importation des modules

Le widget doit comporter l'importation de module suivante :

```
from functools import partial
```

4.2 Constructeur `__init__.py`

Le constructeur de la classe doit notre widget doit comporter les éléments suivants :
— Le `sendButton` doit avoir comme *cancelCallback* `cancel_manually` :

```
        self.sendButton = SendButton(
            widget=self.controlArea,
            master=self,
            callback=self.sendData,
```

```
            cancelCallback=manualCancel, # Manual cancel button
            infoBoxAttribute='infoBox',
            sendIfPreCallback=self.updateGUI,
        )
```

— Les `AdvancedSettings` (si nécessaires) doivent être déclarés au moyen de la méthode définie à cet effet :

— Les éléments principaux de l'interface (qui sont censés être affectés à la `controlArea`) doivent être déclarés au moyen de la méthode `create_widgetbox()` (par exemple ici dans le cas de `optionsBox` :

```
# (Advanced) options box...
self.optionsBox = self.create_widgetbox(
    widget=self.controlArea,
    box=u'Options',
    orientation='vertical',
    addSpace=False,
)
```

Les éléments fréquemment créés de la sorte sont unitsBox, contextsBox, optionsBox, etc. Mais cela dépend bien sûr de chaque widgets.

4.3 Méthode `sendData()`

Il existe ici deux possibilités[9] :

— La méthode sendData() appelle des fonctions d'un autre module (p.ex. LTTL). Dans ce cas, c'est la fonction de LTTL qui s'exécutera dans un worker thread.

— La méthode sendData() fait une série d'opérations sans appeler de fonctions d'un autre module (p.ex. dans le cas des Widgets TextFiles et URLs). Dans ce cas, il faut créer une nouvelle méthode dans laquelle s'exécuteront les opérations qui prennent du temps. Elle sera exécutée dans un worker thread. Ce cas est aussi le plus fréquent dans la création de widgets par des étudiant.x.e.s. (À ce propos voir la prochaine section)

Dans les deux cas, il est nécessaire de définir dans `sendData()` une fonction `threaded_function` de la manière suivante :

```
threaded_function = partial(fonction_worker_thread, caller=self param1 = param1, param2= param
```

`partial` prend comme premier argument une fonction et ensuite les paramètres de cette fonction à remplir. `fonction_worker_thread` contiendra donc ici soit l'appel à une fonction de LTTL, soit la méthode qui contient la série d'opérations du widget.

Il est ensuite nécessaire d'appeler la méthode `self.threading` :

```
self.threading(threaded_function)
```

C'est cet appel qui lance effectivement le processus du widget.

4.4 Utilisation des signaux

Dans le cas le plus courant pour des widgets `Prototypes` (i.e. des widgets développés par des étudiant.x.e.s), il est nécessaire de créer une fonction supplémentaire qui sera appelée dans `sendData()` et qui contiendra le processus effectif du widget. Il faut faire attention ici à quelques éléments.

Le premier est de bien utiliser les couples *signaux/slots* pour mettre à jour régulièrement la barre de progression ou les messages de l'`InfoBox` depuis cette fonction. Ceci permet de garantir une mise à jour *thread safe* de ces informations. Un deuxième élément important est de vérifier à intervalles réguliers dans les processus longs (typiquement les boucles) si le booléen `cancel_operation` a été modifié. Cela permet d'éviter que la boucle tourne trop longtemps avant de s'arrêter si l'utilisateur.ice a stoppé l'exécution du programme. Un bon exemple de ceci est le widget `ConvertThreadJohan` :

Nous pouvons voir dans cet exemple les tests effectués à intervalles réguliers ainsi que l'utilisation des signaux pour envoyer des messages depuis le *worker thread*.

9. Cette partie du tutoriel est en grande partie reprise du travail d'Antonin Schnyder avec quelques modifications.

4.5 Méthode `task_finished()`

Il faut définir une fonction `task_finished` de la manière suivante :

```
@OWTextableBaseWidget.task_decorator
def task_finished(self, f):

    # Récupère les outputs

    outputs = f.result()

    # Effectue les traitements nécessaires avant

    # d'envoyer les résultats en output,

    # comme par exemple compter les caractères

    # ...

    # Envoie les outputs et met à jour l'interface

    self.send(..., outputs) # Outputs du Widget

    self.infoBox.setText(...) # Message de réussite
```

Cette fonction doit être décorée comme l'extrait ci-dessus et doit avoir deux arguments : `self` et `f`. Elle contiendra ensuite le traitement des données et l'envoie vers les différents *outputs*.

5 Prochaines étapes et conclusion

Dans ce rapport, nous avons donc détaillé les modifications apportées au add-on *Textable* dans le but de déplacer les éléments de *threading* pour qu'ils soient hérités au travers de la classe **OWTextableBaseWidget**.

Nous avons réussi à extraire des widgets la plupart des éléments en lien avec le *threading* : ceci nous permet d'alléger le processus de développement de nouveaux widgets ainsi que de faciliter la maintenance des widgets existants. Nous n'avons malheureusement pas pu inclure dans ce travail toutes les pistes explorées mais nous espérons avoir fourni un bon aperçu du travail effectué.

Ce travail pourrait bien sûr être prolongé, par exemple en implémentant un nouveau système de gestions des erreurs. Il serait aussi intéressant de tester notre nouvelle architecture sur les widgets `Prototypes` retravaillés par Olivia Verbrugge dans le cadre de son Projet en informatique pour les sciences humaines. Finalement, certains widgets n'ayant pas été *threadés*, il serait très intéressant – et assurément ardu – d'essayer de le faire dans la nouvelle architecture que nous proposons.

6 Liens

— La page GitHub du projet
— Orange
— Textable
— Xanthos, Aris (2014). Textable : programmation visuelle pour l'analyse de données textuelles. In Actes des 12èmes Journées internationales d'analyse statistique des données textuelles (JADT 2014), pp. 691-703. Read online