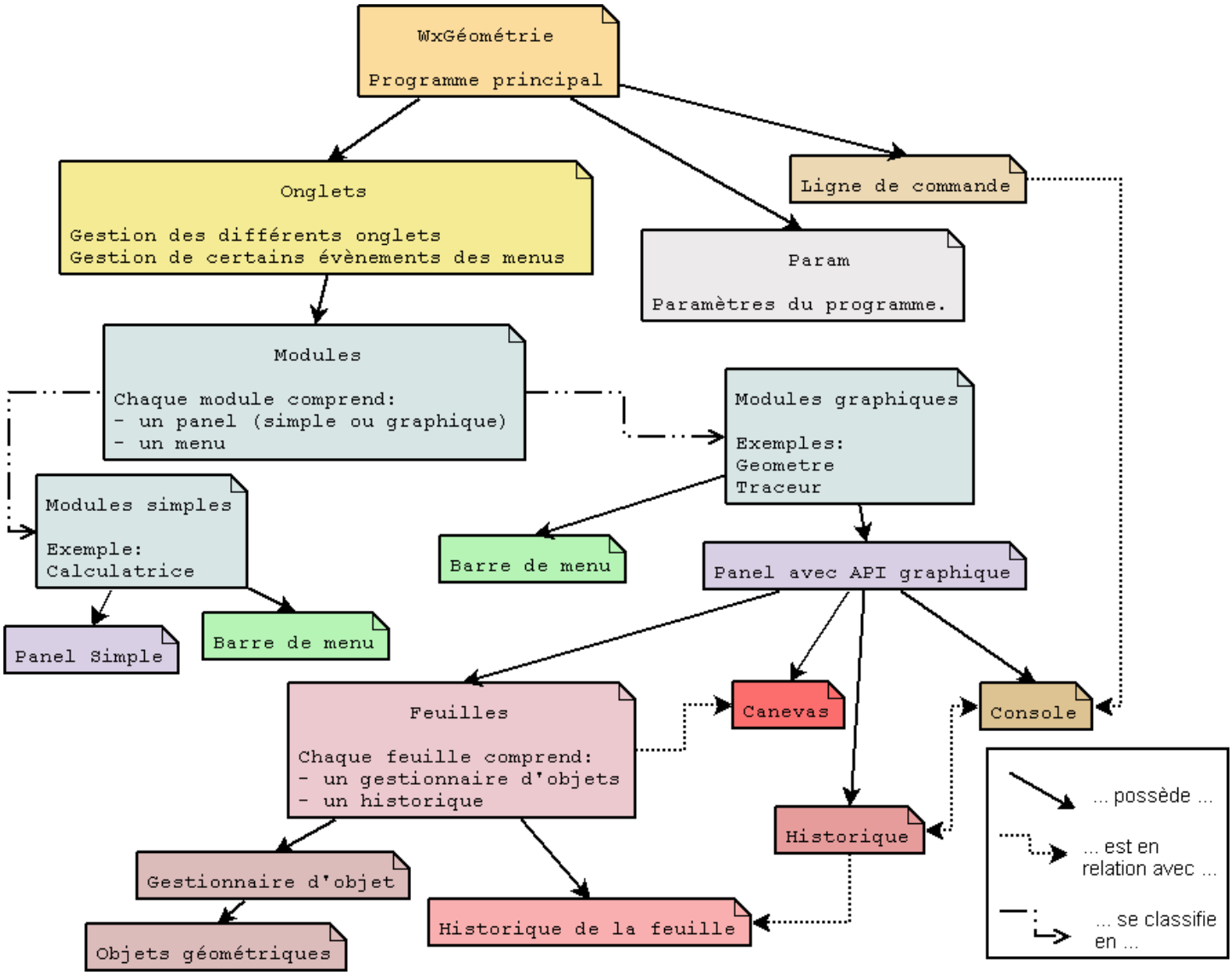
WxGéométrie

WxGéométrie est conçu pour avoir une architecture la plus modulaire possible.
Je suis loin d'être un expert en Python, et encore moins en travail collaboratif.
Je me suis néanmoins efforcé au maximum de rendre mes sources et mon architecture lisible, avec un certain succès semble-t-il pour les échos que j'en ai eu.
Néanmoins, fournir les sources ne dispense pas d'un minimum de documentation.
Selon l'adage, je commencerai mon long discours par un petit dessin.

I. Architecture du programme



Ce schéma n'est pas du tout exhaustif, mais il permet d'avoir une vision d'ensemble du programme.

II. Comment écrire une extension

La manière le plus simple de collaborer au projet est d'ajouter une extension (ou module).

Pour l'instant, les modules disponibles sont les suivants :

- **Geometre :** un module permettant de faire de la géométrie dynamique.
- **Traceur :** un traceur de courbes.
- **Calculatrice :** une calculatrice scientifique supportant les fractions et les racines carrées.

Les 2 premiers modules sont des modules dits *graphiques*, car ils utilisent l'API de WxGéométrie.
Le 3ème module n'utilise pas l'API graphique, il est dit *simple*.

Pour adapter un programme existant et l'intégrer dans WxGéométrie, il est plus judicieux de créer un module simple.

Pour créer une nouvelle extension de toute pièce, le support de l4API graphique permettra de créer quelque chose de mieux intégré, et de plus puissant à moindres frais.

En plus de la documentation, il est conseillé de bien lire le code des modules déjà existants, et de s'en inspirer – vous y trouverez certaines idées ; et par ailleurs, la documentation est assez incomplète pour l'instant.

1. Les extensions simples

Pour créer une extension simple, il suffit de créer un nouveau dossier *mon_extension* dans le dossier *modules* de WxGéométrie.
Dans ce nouveau dossier, il faut créer un fichier *__init__.py*, qui contient obligatoirement les éléments suivants:

a) L'import des librairies standards de WxGéométrie

(Au moins de *API.py*)

```
from API import *
from GUI import *
```

Bien entendu, ces lignes doivent se trouver au début du fichier.

b) Une classe héritée de MenuBar

Cette classe, définie dans *API/menu.py*, servira à construire le menu.

Voici l'exemple de la calculatrice.

```
class CalculatriceMenuBar(MenuBar):
    def __init__(self, panel):
        MenuBar.__init__(self, panel)
        self.ajouter("Fichier", ["quitter"])
        self.ajouter("Affichage", ["onglet"])
        self.ajouter("Outils",
```

```
                        ["Memoriser le resultat", "Copie le resultat du
calcul dans le presse-papier, afin de pouvoir l'utiliser ailleurs.",
"Ctrl+M", self.panel.vers_presse_papier],
                        ["options"])
        self.ajouter("?")
```

On va le décortiquer un peu.

Les 3 première lignes sont à recopier telles qu'elles :

```
class CalculatriceMenuBar(MenuBar):
    def __init__(self, panel):
        MenuBar.__init__(self, panel)
```

Les lignes suivantes ajoutent des éléments au menu.

```
self.ajouter("Fichier", ["quitter"])
```

On crée une liste « Fichier » dans le menu, qui contiendra l'élément prédéfini « quitter ».

Il faut savoir qu'un certain nombre d'éléments prédéfinis existent[1] pour le menu.

Cela dit, dans la plupart des cas, il faudra créer sa propre entrée dans le menu :

```
        self.ajouter("Outils",
                        ["Memoriser le resultat", "Copie le resultat du
calcul dans le presse-papier, afin de pouvoir l'utiliser ailleurs.",
"Ctrl+M", self.panel.vers_presse_papier],
                         ["options"])
```

Ici, "options" correspond à une entrée prédéfinie, tandis que la première entrée est créée de toute pièce.
La syntaxe est la suivante :

```
self.ajouter("Liste d'entrées",
   ["Entrée n°1", "Commentaire de l'entrée n°1", "Ctrl+E", fonction1],
   None,
   ["Entrée n°2", "Commentaire de l'entrée n°1", "Alt+Ctrl+K", fonction1],
   ["Dernière entrée", "Commentaire de l'entrée n°1", None, fonction1])
```

Chaque entrée est donc représentée par une liste.
Elle composée par :

- le **titre** de l'entrée (celui qui apparaît dans le menu) -> type **str**.
- un **commentaire** (qui apparaît en bas de la fenêtre) -> type **str** ou **None**.
- Un **raccourci** clavier -> type **str** ou **None**
- Une **fonction** à lancer quand l'entrée est sélectionnée -> type **func** ou **None**
- Une **variable** ***(facultatif)*** qui indique que l'entrée doit être cochée (ou non) -> type **bool**

Attention : la fonction reçoit un argument event.

Typiquement, la fonction est une méthode du Panel (voir plus loin).

Elle sera donc définie comme ceci :

```
def ma_fonction(self, event):
     titre = event.nom_menu
     print "Hello world!"
```

etc...

---

1 La liste exhaustive de ces alias peut-être trouvée à la fin du fichier API/menu.py.
Par exemple, « "ouvrir" » remplace « ["Ouvrir", "Ouvrir un fichier.", "Ctrl+O", self.parent.OpenFile] ».

On notera qu'on peut récupérer le titre de l'entrée sélectionnée grâce à la propriété *nom_menu*.

[à compléter – flux « RSS », etc... **]**

c) Une classe héritée de Panel_simple

Les 3 premières lignes doivent être :

```
class Calculatrice(Panel_simple):
    __titre__ = "Calculatrice"

    def __init__(self, parent):
        Panel_simple.__init__(self,parent)
```

Bien entendu, le titre est à adapter.

Il correspond à ce qui s'affiche en haut de l'onglet correspondant.

Pour le reste, vous pouvez mettre absolument tout ce que vous voulez ; *Panel_simple* se comporte comme la classe *wx.Panel* ordinaire[2].

Remarque : si vous voulez intégrer dans WxGéométrie un programme déjà existant en WxPython, ce dont je vous saurai grand gré, il suffit bien souvent de faire hériter la classe principale du programme de *Panel_simple*, au lieu de *wx.Panel*.

[à compléter – comment récrire les méthodes _ouvrir et _sauvegarder **]**

2. Les extensions utilisant l'API graphique

Quand je parle d'API graphique, il s'agit essentiellement de la gestion des objets géométriques (points, droites, intersections...), du réglage de la fenêtre d'affichage (zoom, etc...), et de l'export en png, eps et svg.

Pour créer une extension graphique, il suffit là encore de créer un nouveau dossier *mon_extension*[3] dans le dossier *modules* de WxGéométrie.
Dans ce nouveau dossier, il faut créer un fichier *__init__.py*, qui contient obligatoirement les éléments suivants:

a) L'import des librairies standards de WxGéométrie

(Au moins de *API.py*)

```
from API import *
from GUI import *
```

Bien entendu, ces lignes doivent se trouver au début du fichier.

2 Typiquement, deux fonctions sont conçues pour être récrites : _sauvegarder(self, fgeo) et _ouvrir(self, fgeo).
Dès que j'aurai un peu de temps, je documenterai la chose...
En attendant, vous pouvez jeter un coup d'oeil dans les trois modules déjà existants.

3 N'utilisez que des caractères alpha-numériques et le tiret bas « _ ».

b) Une classe héritée de MenuBar

Par rapport au panel simple, un plus grand nombre de menus sont prédéfinis.
En particulier, un menu "créer" comprend toute la gestion des objets.

```
self.ajouter("creer")
```

Se reporter au ***1.b)*** pour plus de détails.

c) Une classe héritée de Panel_API_graphique

Les premières lignes sont les suivantes :

```
class Traceur(Panel_API_graphique):
    __titre__ = "Traceur de courbes" # Donner un titre a chaque module

    def __init__(self, parent):
        Panel_API_graphique.__init__(self, parent)
```

Bien entendu, le titre est à adapter en fonction de votre module (il apparaîtra en haut de l'onglet).

La dernière ligne crée 3 attributs importants :

- **self.canvas** : un espace de dessin supportant les options avancées de WxGéométrie : zoom, dessin sans crènelage, export en png, eps et svg, et support des objets géométriques en particulier.
  Une bonne partie de ces fonctionnalités proviennent de la librairie matplotlib, dont il peut être utile de consulter la documentation.
- **self.historique** : enregistrement des commandes passées, gestion des annulations.
  Pour plus de détails, consulter le ***3.b*** à ce sujet.
- **self.commande** : la console. Elle joue plusieurs rôles : filtrer et reformuler les commandes, stocker les commandes passées dans l'historique (principalement).

d) Comment rajouter des fonctions au panel ?

En développant le nouveau module, vous aurez besoin de rajouter de nouvelles fonctions.

La manière la plus naturelle de procéder est d'ajouter une nouvelle méthodes à Panel_API_graphique, et une nouvelle entrée au menu.

Pour le Panel, il faut penser à rajouter une 2ème méthode, qui sera appelée par le menu, et qui servira à faire passer la commande par la console.

Enfin, il faut que la console reconnaisse cette nouvelle méthode comme valable, pour qu'on puisse la lancer avec avec un niveau de sécurité élevé (qui doit, à moyen terme, devenir le mode par défaut) .

Prenons un exemple concret. Vous avez créé une fonction pour dessiner de jolies coccinelles[4].

- Vous rajouter dans la méthode *__init__* de votre panel la ligne suivante :

```
self.commandes.commandes_panel.append( "creer_coccinelle(" )
```

Ceci permet à la commande de passer à la ~~casserole~~ console sans encombre, en étant répertoriée comme une méthode du Panel accessible à l'utilisateur final.

- Vous rajouter à la fonction *__init__* de votre barre de menu la ligne suivante :

```
self.ajouter("fonctions inutiles", ["créer une coccinelle", "Créer une
jolie coccinelle sur la feuille.", "Alt+Ctrl+C",
self.panel.creer_coccinelle])
```

4 Et pourquoi pas ? C'est sympathique, une coccinelle. Hum... revenons à notre sujet.

- Vous définissez **deux** nouvelles méthodes pour votre panel :

```
def creer_coccinelle(self, event = None):
     self.commande.executer("creer_cocci()")

def creer_cocci(self):
     Placez votre code ici
     ...
```

Tout ceci n'est pas 100% obligatoire, mais aidera à ce que votre fonction s'intègre bien dans le programme.

[à compléter – décrire l'API de l'objet canevas **]**

3. Quelques remarques importantes pour finir

a) détection des modules

Pour qu'un module soit lancé au chargement du programme, il faut éditer le fichier de configuration param.py, et modifier la ligne :

```
# Modules a importer
# ----------------

modules = ["geometre", "traceur", "calculatrice"]
```

Supposons que votre module s'appelle *mon_module* (le nom du sous-répertoire que vous avez créé dans le répertoire *modules*).
Vous remplacez donc cette ligne par :

```
modules = ["geometre", "traceur", "calculatrice", "mon_module"]
```

Le fichier *modules.py* va ensuite analyser le module, pour y chercher une classe héritée de *Panel_simple* (*Panel_API_graphique* hérite lui-même de *Panel_simple*), et une classe héritée de *MenuBar*.
Ces classes doivent **obligatoirement** être présentes[5], et être uniques.
(Rien ne vous empêche par contre d'avoir un certain nombre de classe héritant de *wx.Panel*, et de *wx.MenuBar*)

b) Gestion de l'historique (pour les modules graphiques)

L'annulation et la restauration fonctionne selon un principe assez rudimentaire :

à chaque fois qu'une commande est passée à la console du module, la feuille de travail enregistre son état actuel[6].
A chaque fois qu'on annule, la feuille de travail restaure l'avant-dernier état enregistré.

Autrement dit, seules les commandes se rapportant à la feuille peuvent actuellement être annulées :

- création, déplacement, etc... d'objet géométriques,
- changement de la fenêtre d'affichage (zoom, ...).

5 Si vous n'utilisez pas leurs fonctionnalités spécifiques, vous pouvez toujours vous en servir comme de simples *wx.Panel* et *wx.MenuBar*, dont elles héritent.

6 Par ailleurs, la commande est stockée par l'historique du module, qui garde trace de toutes les commandes passées.