urbanotopus Documentation

urbanotopus

Dec 13, 2018

Contents

Documentation générée

La documentation doxygen générée du code C#, est disponible ici.

Téléchargement et démo

Les liens de téléchargement sont disponibles ici, vous pouvez aussi directement lancer le jeu depuis votre navigateur web ici.

Contenu

CHAPTER 1

Le jeu

Urbanotopus est un jeu où le joueur se fond dans l'histoire d'un urbaniste avec une multitude de choix menant à une infinité de dénouements de l'histoire.

L'objectif de ce jeu est d'apprendre au joueur les implications de chaque choix et donc l'important qu'à et que les choix, parfois difficiles de l'urbaniste ont.

1.1 Le fonctionnement

Chaque jour, le joueur est présenté à de nouvelles situations ou des situations en continues où le joueur doit prendre une ou des actions qui modifient les variables de sa partie. Ces situations et décisions peuvent influencer en bien ou en mal la partie du joueur en fonction de ses décisions.

Le joueur peut très facilement juger de ses performances à partir des conséquences, mais aussi en lisant ce que les gens postent sur le flux Twitter. Mais encore en regardant les statistiques de la partie.

1.2 Le déroulement d'une partie

Après avoir été introduit au jeu, le joueur retourne dans *son* bureau d'urbaniste, où, il est présenté à de nouvelles tâches et décisions dans l'onglet `Histoire`, parmi lesquels il doit choisir sa prochaine destination.

1.2.1 L'interface

L'interface est très simple, les boutons principals sont au milieu du bas de l'écran :

- **Histoire**, permet de sélectionner le chapitre suivant ou relire un précédent ;
- **Statistiques**, permet de voir l'avancement et l'état de la ville ;
- **Twitter**, permet de voir les réactions des citoyens face aux décisions et à l'état de la ville.

Puis, de boutons dans le coin du haut à droite, permettant de :

- Sauvegarder une partie ;
- Charger une partie.

Et enfin, un bouton dans le coin du haut à gauche, permettant de retourner au menu principal.

Fig. 1: Capture d'écran de l'interface principale

1.2.2 Le début de la partie

-

1.2.3 La pleine partie

-

1.2.4 La fin de la partie

-

1.3 Le paramétrage d'une partie

-

CHAPTER 2

Conventions du projet

Vous retrouverez ci-dessous toutes les informations sur comment nous travaillons sur notre projet. Pour les conventions de codage, voyez *Style de codage*.

2.1 La répartition des tâches

2.1.1 Les dates limites

Une date limite (deadline) est définie pour chaque ensemble de tâches. Par exemple, dans le tableau de bord d'octobre à novembre, il est défini que cet ensemble doit être prêt pour le 5 novembre 2018.

À partir de la date d'échéance d'un ensemble de tâches, nous imposons que tous les sous-ensembles de tâches aient leur date limite qui soit estimée par rapport à la date limite de l'ensemble parent, en essayant de garder une marge de retard entre les deux dates en cas d'imprévues.

2.1.2 Les tâches

Ensuite, maintenant que les dates des ensembles sont définies, nous arrivons aux tâches des sous-ensembles.

Elles n'ont pas forcément une date limite associée tant quelles sont à faible priorité, donc que d'autres qu'il y a plus de cinq tâches prioritaires non dépendantes. Cela nous permet de donner la possibilité aux personnes de l'équipe de pouvoir s'assigner eux-même aux tâches n'ayant personne d'assigner, grâce à leurs variables :

- Disponibilités (Université, santé, etc.) ;
- Difficultés ;
- Apprentissage sur le tas.

L'objectif principal de cette méthode de travail est d'empêcher toute charge écrasante envers l'équipe qui peut avoir des difficultés à sortir le même débit que d'autres membres, ce qui permet donc à chacun de prendre son temps pour apprendre correctement son travail et les technologies utilisées mais en échange, une plus grande qualité est attendue.

Si, avec prise en compte de la difficulté et donc du temps requis, les dates butoirs des tâches sont trop proches, dans ce cas, les tâches se font assigner par le chef de projet à un (ou des) membres qui répondent le mieux au travail attendu.

CHAPTER 3

Guides de contribution

Merci d'utiliser le système de tickets (issues) et de pull requests pour rapporter des problèmes, suggérer et discuter de modifications et de nouvelles fonctionnalités.

3.1 EditorConfig

EditorConfig est un fichier standard de configuration qui a pour but d'assurer un style de codage constant à travers les environnements de développement.

Le projet possède un fichier `.editorconfig` à sa racine qui décrit nos exigences de formatage de notre code source.

La plupart des éditeurs et IDEs supportent ce fichier, que cela soit par défaut ou par le biais d'un add-on. Pour plus d'informations et d'instructions, consultez la liste des éditeurs et IDEs supportés.

Vous retrouverez notamment cet add-on pour Visual Studio. Dans le cas de l'IDE `Rider`, le module est déjà préinstallé.

Merci de vous assurer que votre environnement de développement respecte bien ce fichier et corrigera ou vous avertira de tout problème d'indentation, d'encodage, et de fin de ligne.

3.2 Style de codage

3.2.1 Python

Suivez toujours la convention PEP 8 dans sa totalité (ceci incluant la limite de 80 caractères sur une ligne).

Chaînes de caractères

Utilisez guillemets simples sauf si la chaîne est une docstring.

Les blocs de code

Merci de suivre le style de formatage “hanging grid”. Comme suit :

```
some_dict = {
    'one': 1,
    'two': 2,
    'three': 3}
```

```
some_list = [
    'foo', 'bar', 'baz']
```

Merci d’éviter de laisser pendre des parenthèses, crochets, ou virgules. Par exemple, ne faites pas cela :

```
this_is_wrong = {
   'one': 1,
   'two': 2,
   'three': 3,
}
```

Cassez les lignes de code directement après les parenthèses. Ne faites donc pas cela :

```
also_wrong('this is hard',
           'to maintain',
           'as it often needs to be realigned')
```

Linters

Utilisez :

1. isort pour maintenir une consistance dans vos imports ;
2. pylint pour détecter les erreurs dans votre code ;
3. pycodestyle pour assurer que votre code respecte les conventions PEP 8 ;
4. pydocstyle pour vérifier que vos docstrings sont correctement formatées ;
5. eslint pour vérifier votre code Javascript ;
6. tslint pour vérifier votre code Typescript.

3.2.2 CSharp (C#)

3.2.3 Les blocs de code

Veuillez ne pas retourner à la ligne après l’ouverture d’un crochet puis retourner à la ligne après une fermeture.

Faites :

```
namespace models {
    public class Employee {
        public Employee(int id) {
            if (id == 0) {
                try {
                    id = id / 0;
                }
                catch(System.DivideByZeroException exc) {
                    // We did not stick `catch(...) {` to `}`.
                }
            }
            else {
                // We did not stick `else {` to `}`.
            }
        }
    }
}
```

3.2.4 L'accès à l'instance courante

Veuillez utiliser le mot `this` pour accèder à l'instance courante.

Faites :

```
public class Employee {
    private string _nameAlias;
    private string _name;

    public Employee(string name, string nameAlias) {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this._alias = name;
        this._nameAlias = nameAlias;
        System.Console.WriteLine(this._name);
    }
}
```

Et non pas :

```
public class Employee {
    private string _nameAlias;
    private string _name;

    public Employee(string name, string nameAlias) {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        _name = name;
        _nameAlias = nameAlias;
        System.Console.WriteLine(_name);
    }
}
```

3.3 Convention de nommage

3.3.1 Python

1. Les variables, nom de fonctions et méthodes doivent être écrites en minuscles et les mots séparés par des underscores ;
2. Les constantes doivent être écrites en majuscules et les mots séparés par des underscores ;
3. Les noms de classes et d'objets doivent commencer par une majuscule et suivre le format `CamelCase` ;
4. Les méthodes et variables privées doivent commencer par un underscore.

```
CONSTANT = 'abc'

class HelloWorld(object):
    _private_prop = 'hello'

    def __init__(self):
        pass

    def _private_method(self):
        pass

    def say_hi(self):
        pass

def main():
    var = 123
```

3.3.2 CSharp (C#) et Javascript/ Typescript

1. Les variables, nom de fonctions et méthodes doivent commencer par une minuscles et suivre le format `camelCase` ;
2. Les constantes doivent être écrites en majuscules et les mots séparés par des underscores ;
3. Les noms de classes, d'objets et de propriétés doivent commencer par une majuscule et suivre le format `CamelCase` ;
4. Les méthodes et variables privées doivent commencer par un underscore.

```
const int CONSTANT = 123;

private class HelloWorld {
    public int SomethingPublic = 123;

    private const _PRIVATE_CONSTANT = 123;
    private int _privateProp;

    HelloWorld() {
        // do something...
    }

    private int _privateMethod() {
        int variableWorld = 123;
        return variableWorld;
    }

    public void say_hi() {
        // do something...
    }
}
```

3.4 Contribuer au code

3.4.1 Ajouter et modifier du contenu dans nos projets

Si vous souhaitez ajouter ou modifier du code ou n'importe quel autre contenu dans nos projets, vous devez tout d'abord commencer par **créer une issue** avant d'ouvrir une **pull request** (sauf si le changement est vraiment très mineur, comme une ou des typos, dans ce cas **une simple pull request est suffisante**).

L'objectif de l'ouverture d'une issue avant l'ouverture d'une *pull request* est de pouvoir discuter du problème que vous souhaitez résoudre ou de la solution que vous proposez. Cela permet donc d'ouvrir une discussion avant d'ouvrir une *pull request* qui risque fortement d'être refusée car non convenable et donc demandera une ou des modifications, menant à du travail et temps supplémentaire, et cela salira l'historique `git` du projet.

De plus, si votre changement apporte des changements ou des ajouts graphiques, veillez à ajouter une ou des captures d'écran ou une vidéo de démonstration.

> **Warning:** Veillez à avoir un message de commit propre et ne pas créer une masse de commits pour des changements mineurs. Quitte à réécrire ou modifier les commits afin de nettoyer ce que vous avez fait. Plus d'informations.

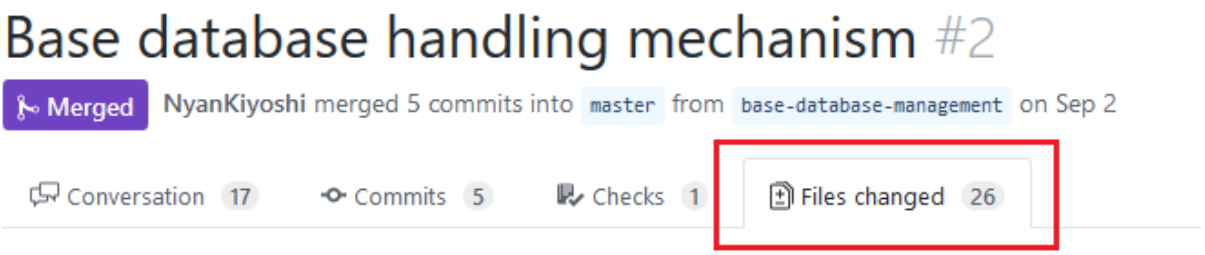
Note: Évitez de *pull* la branche `master` dans le but de mettre à jour votre branche ou afin de résoudre des conflits. Préférez plutôt un `git rebase -i` ce qui permettra de préserver un historique propre.

3.4.2 Comment contrôler (review) les pull requests

Notre méthode standard afin de *review* une *pull request* est :

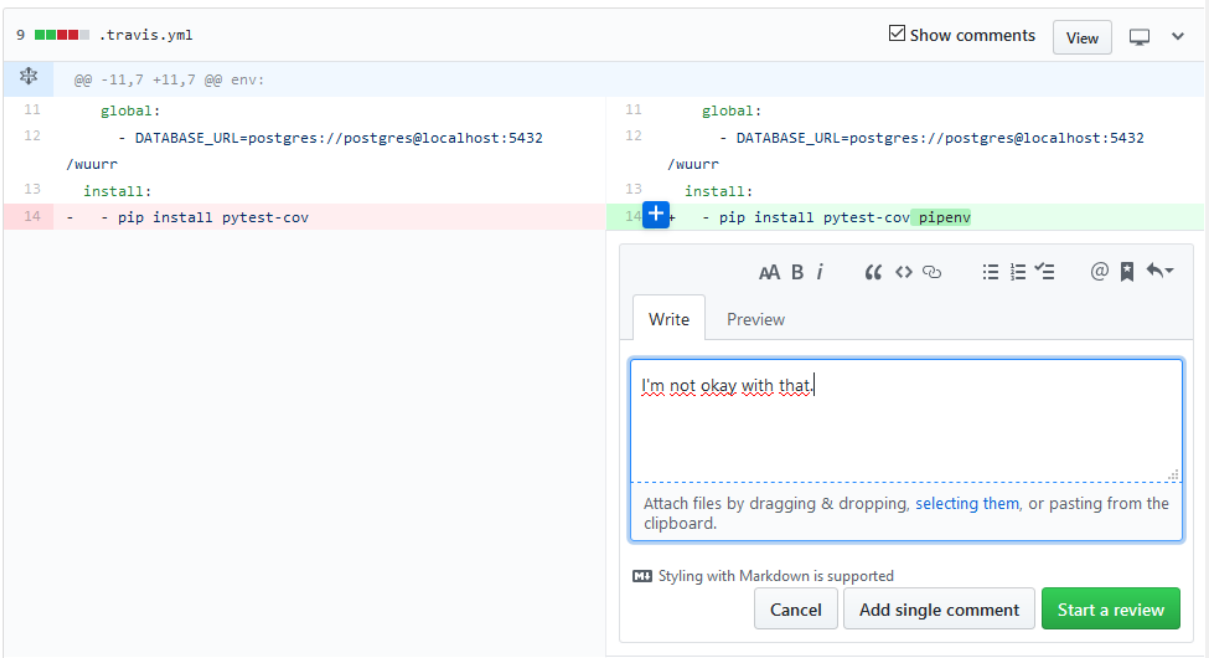
Première étape

Lire le code source et changements à la recherche d'erreurs (bugs ou possibles bugs et typos), de mauvaises pratiques, de violation de convention, de complexités et de code non optimal.



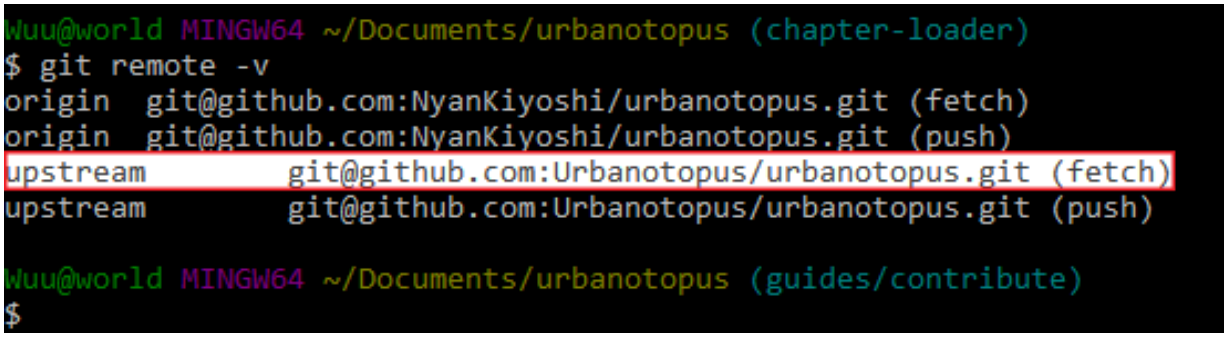
Deuxième étape

Suggérer des changements si certaines choses semblent mauvaises, pourraient être améliorées ou affinées. Le tout avec des commentaires constructifs.



Troisième étape

Récupèrer les changements pour les tester et les vérifier localement. Pour cela, assurez-vous que votre clone local du dépôt git a pour `upstream remote`, la source de base du projet, celui pour lequel la *pull request* est associée. Pour vérifier, faites `git remote -v`, vous devriez avoir en upstream `fetch` le dépôt de source, comme suit :



Si vous avez un remote mais pas le bon, faites : `git remote remove upstream` ;

Si vous n'avez le remote `upstream` **ou vous avez supprimé l'upstream (avec la commande ci-dessus)**, ajoutez via `git remote add URL_DÉPÔT_SOURCE>`. Si le projet est `Urbanotopus`, faites `git remote add git@github.com:Urbanotopus/urbanotopus.git`.

Vérifiez avec `git remote -v`.

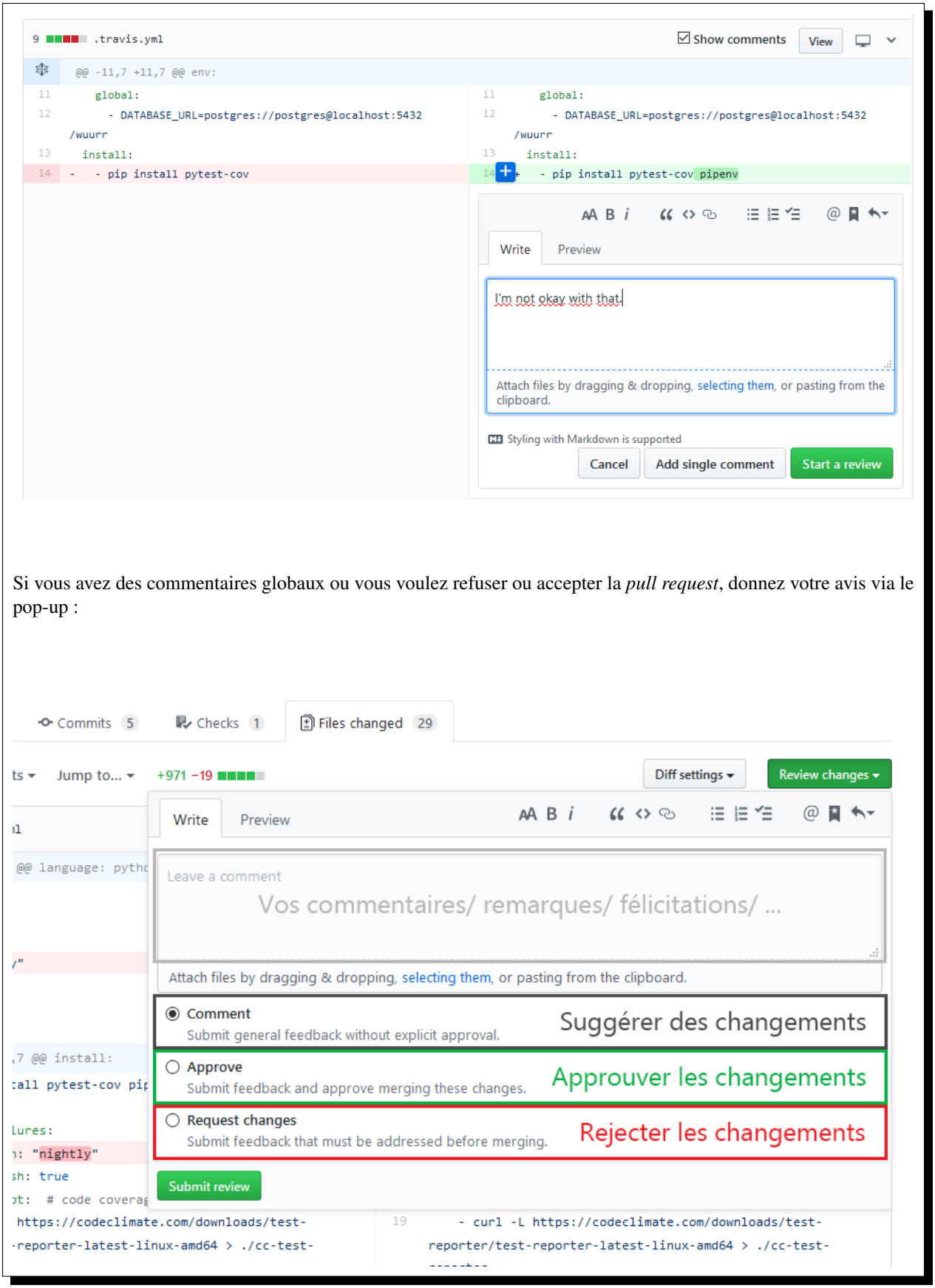
Ensuite, récupérez le contenu de la *pull request* en faisant un `git fetch upstream pull/ID_DE_LA_PULL_REQUEST/head:NOM_DE_BRANCH` puis `git checkout NOM_DE_BRANCH`.

Par exemple, si la *pull request* est `#2`, faites `git fetch upstream pull/2/head:test-database` puis `git checkout test-database`.

Testez les changements qui ont été récupérés.

Quatrième étape

Maintenant que vous avez testé les changements, vous pouvez valider ou non les changements. Si vous trouvez un bug ou souci à une ligne donnée ajouter un review à la ligne concernée :



Si vous avez des commentaires globaux ou vous voulez refuser ou accepter la *pull request*, donnez votre avis via le pop-up :

3.5 Documenter le code source

Pour chaque méthode et classe et veillez à ce que les documentations en commentaires restent valides et à jour si vous modifiez la signature et/ ou le corps d'une méthode ou d'une classe.

3.5.1 Documenter le code C# (XML)

Veuillez inclure les commentaires XML standards de C#. Chaque méthode doit au moins inclure un `<summary>` et tous les paramètres (`<param name="something">`) ainsi qu'un `<returns>` si la méthode n'est `void`.

Formatage des tags XML

1. Les tags avec une seule ligne doivent commencer et se terminer sur la même ligne.

2. Les tags sur multiples lignes doivent avoir une ligne vide après l'ouverture et avant la fermeture. Et ne doivent comporter une indentation.

3. Les tags imbriqués doivent avoir une indentation de 4 espaces.

Exemple :

```
/// <summary>This has only one line</summary>
/// <remarks>
/// This has multiple
/// lines
/// </remarks>
/// <param name="n">
///     <ul>
///         <li>This is nested</li>
///     </ul>
/// </param>
```

Exemple complet

```
/// <summary>Class level summary documentation goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface {
    /// <summary>Store for the name property.</summary>
    private string _name = null;

    /// <summary>The class constructor.</summary>
    public TestClass() {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>Name property.</summary>
    /// <value>A value tag is used to describe the property value.</value>
    public string Name {
        get {
            if (_name == null) {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>Description for SomeMethod.</summary>
    /// <param name="s">Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s) {
    }

    /// <summary>Some other method.</summary>
    /// <returns>Return results are described through the returns tag.</returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific method.
    /// </seealso>
    public int SomeOtherMethod() {
        return 0;
    }

    /// <summary>The entry point for the application.</summary>
    /// <param name="args">A list of command line arguments.</param>
    static int Main(System.String[] args) {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>Documentation that describes the interface goes here.</summary>
/// <remarks>Details about the interface go here.</remarks>
interface TestInterface {
    /// <summary>Documentation that describes the method goes here.</summary>
    /// <param name="n">Parameter n requires an integer argument.</param>
    /// <returns>The method returns an integer.</returns>
    int InterfaceMethod(int n);
}
```

3.5.2 Documenter le code Python avec des docstring Sphinx

Veuillez inclure des docstring du format Sphinx sur Python. Chaque méthode doit au moins inclure une courte description sur ce qu'elle fait et tous les paramètres (`:param something: it's something`) ainsi qu'un `:returns: Blablabla` si la méthode retourne quelque chose (plus d'informations).

Formatage de la docstring

Veuillez noter que la description doit être sur la même ligne que sur l'ouverture de la docstring. Par exemple :

```
"""This is a good way to do.
I'm happy.
"""
```

Et ne pas faire :

```
"""
This is a good way to do.
I'm happy.
"""
```

Exemple complet

```
"""
.. module:: useful_1
   :platform: Unix, Windows
   :synopsis: A useful module indeed.

.. moduleauthor:: Andrew Carter <andrew@invalid.com>


"""

def public_fn_with_googley_docstring(name, state=None):
    """This function does something.

    Args:
       name (str):  The name to use.

    Kwargs:
       state (bool): Current state to be in.

    Returns:
       int.  The return code::

          0 -- Success!
          1 -- No good.
          2 -- Try again.

    Raises:
       AttributeError, KeyError

    A really great idea.  A way you might use me is

    >>> print public_fn_with_googley_docstring(name='foo', state=None)
    0

    BTW, this always returns 0.  **NEVER** use with :class:`MyPublicClass`.

    """
    return 0

def public_fn_with_sphinxy_docstring(name, state=None):
    """This function does something.

    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    return 0

def public_fn_without_docstring():
    return True

def _private_fn_with_docstring(foo, bar='baz', foobarbas=None):
    """I have a docstring, but won't be imported if you just use "members".
    """
    return None


class MyPublicClass(object):
    """We use this as a public class example class.
```

3.6 Documenter le projet et fonctionnalités

Le projet est documenté en utilisant `Sphinx`. Vous pouvez retrouver les syntaxes sur la documentation du projet sphinx.

Veillez à ce que :

- Chaque fonctionnalité ajoutée au projet soit documentée au sein de la section des guides.
- L'architecture doit être documentée, et donc si un changement d'architecture a lieu, veuillez mettre à jour la section architecture.
- Si la ou les méthodes de déploiement changent, veillez à mettre à jour la section *Déploiement*.

CHAPTER 4

Guides de développement

Cette section regroupe nos différents guides pour vous guider au travers des différents mécanismes mis en place par le projet et vous assister dans le développement de nouvelles fonctionnalités.

4.1 Scènes de Visual Novel

4.1.1 Ajouter un script

Dans le dossier Assets/YarnScripts, créez un nouveau script Yarn ayant pour extension de fichier `.yarn.txt` (**important**).

4.1.2 Modifier un script (Yarn)

Les scripts sont écrits dans le langage Yarn (documentation).

Afin d'éditer les nodes et scripts, utilisez l'éditeur **Merino** qui est inclut avec le projet dans le menu *Window* : `Window > Merino (Yarn Editor)`.

Les dialogues

Un dialogue se compose ainsi :

```
Eve: Hey, I'm a character speaking in a dialogue!
Bob: Wow! Me too!
I'm a narrative guy I guess...
```

Les nœux (nodes)

- Le node de base : **Start**

```
Eve: Hi!
[[ NodeInTheWoods ]]
Eve: I'm back.
```

Ce node instruit Eve a dire bonjour et à aller dans les bois, donc dans le node `NodeInTheWoods`, **Eve ne reviendra jamais des bois**, car c'est un `GOTO`.

- Le node dans les bois : **NodeInTheWoods**

```
Eve: the woods are so scary...
```

Les choix de nœux

Si nous souhaitons laisser le choix à l'utilisateur entre aller dans le nœu de la maison hantée ou dans le nœu de la plage on fera :

```
Eve: where should I go...
[[ The haunted house | HauntedHouse ]]
[[ The beach | Beach ]]
```

Les choix simples

Si vous souhaitez faire en sorte que l'utilisateur puisse choisir pour executer un dialogue puis retourner à l'exécution normale, faites :

```
Mae: What did you say to her?
-> Nothing.
    Mae: Oh, man. Maybe you should have.
-> That she was an idiot.
    Mae: Hah! I bet that pissed her off.
Mae: Anyway, I'd better get going.
```

Les choix conditionnels

Si vous souhaitez afficher des actions seulement dans certains cas, vous avez la possibilité de faire cela :

```
<< set $money = 2 >>
Bob: What would you like?
-> A burger. << if $money >= 5 >>
    Bob: Nice. Enjoy!
-> A soda. << if $money >= 2 >>
    Bob: Yum!
-> Nothing.
Bob: Thanks for coming!
```

Les déclarations de variables et de conditions

Vous pouvez déclarer des variables avec des expressions et des conditions avec les mots `if`, `elseif` et `else`.

```
<< set $number_of_hostages = 12 >>
<< set $hostages_saved = 1 >>
<< set $hostages_saved = $number_of_hostages / 2 >>

<< if $hostages_saved == $number_of_hostages and $time_remaining > 0 >>
    You win the game!
<< elseif $hostages_saved < $number_of_hostages and $time_remaining > 0 >>
    You need to rescue more hostages!
<< elseif $bomb_has_exploded == 0 >>
    You failed to rescue the hostages before time ran out!
<< else >>
    You failed everything.
<< endif >>
```

4.2 Gestion des décisions

4.2.1 Répertoriage des choix possibles

Afin de permettre une consistance simple et efficace des choix possibles entre les scripts Yarn et les bases de données côté serveur, nous utilisons un mécanisme de répertoriage de ces derniers.

Méta-données d'un chapitre

Dans un fichier, `Assets/ChapterData/<chapter_id>/questions.json`, vous retrouverez et respecterez la structure suivante :

Listing 1: `Assets/ChapterData/<chapter_id>/questions.json`

```
{
    "<question_id>": {
        "<answer_id>": {
            "description": "something"
        }
    }
}
```

4.2.2 Sauvegarde et propagation d'une décision

Lorsque le joueur prend une décision, elle doit être annoncée en accordance avec les *méta-données d'un chapitre* par le biais de l'instruction Yarn suivante :

```
<< register_choice Globals <chapter_id> <question_id> <answer_id> >>
```

Le jeu se chargera ensuite de la serialisation et de la communication avec le serveur distant (en tâche asynchrone).

Note: Veillez à ce que le *prefab* `Globals` soit disponible dans votre scène.

4.3 Charger des scènes

Pour charger des scènes, vous devez utiliser `Managers.InternalScenesManager.LoadScene(name: string)` afin de correctement prendre en charge les longues étapes de chargement, et non pas geler l'écran de l'utilisateur. Voici la définition :

class Managers.**InternalScenesManager**

MainMenu = "MainMenuScene";
Le nom de la scène du menu principal.

Office = "OfficeScene";
Le nom de la scène du bureau de l'urbaniste.

VisualNovel = "VisualNovelScene";
Le nom de la scène du bureau de visual novel.

LoadScene(*name: string*)

Parameters **name** – Le nom de la scène à charger.

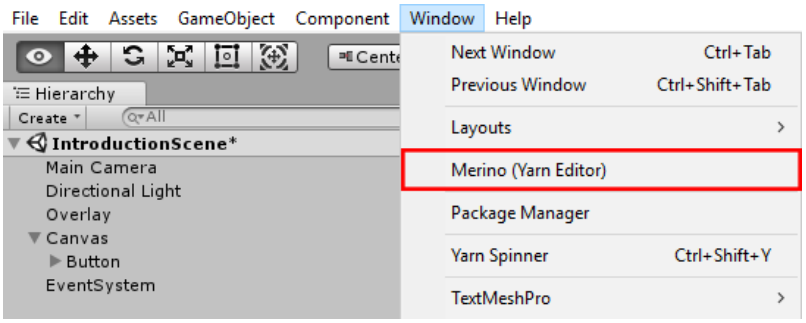


Fig. 1: Capture d'écran du menu Unity

CHAPTER 5

Déploiement

CHAPTER 6

Ressources

Cette page contient les liens vers les différentes ressources du projet disponibles sur Google Drive.

1. Le cahier des charges fonctionnel ;
2. Tableau de bord des tâches du projet.

Index

L

M