Aix*Marseille université
Socialement engagée

Année 2021-22

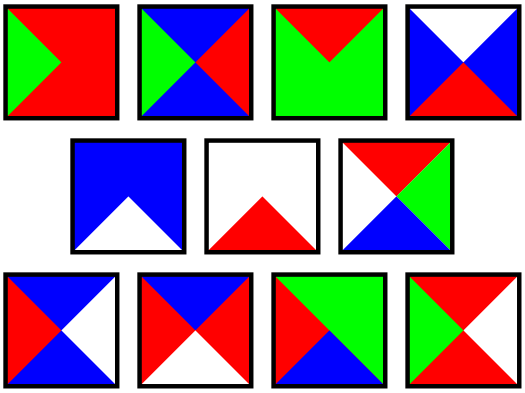
Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 6 (informatique)
Code UE : SIN3U02L

Tuiles de Wang

1 Tuiles de Wang

Ce sujet est construit sur l'idée de réaliser des pavages du plan, à l'aide d'objets simples nommés des tuiles de Wang. Il s'agit de tuiles carrées divisées en quatre triangles. Chaque triangle est associé à un côté du carré, et coloré d'une couleur particulière. Le dessin suivant permet de rapidement se faire une idée de la forme d'une tuile, ainsi que du fait que les couleurs des triangles ne sont pas forcément toutes distinctes.



Nous vous conseillons de bien lire les consignes à respecter avant de vous attaquer au TP.

2 Consignes à suivre

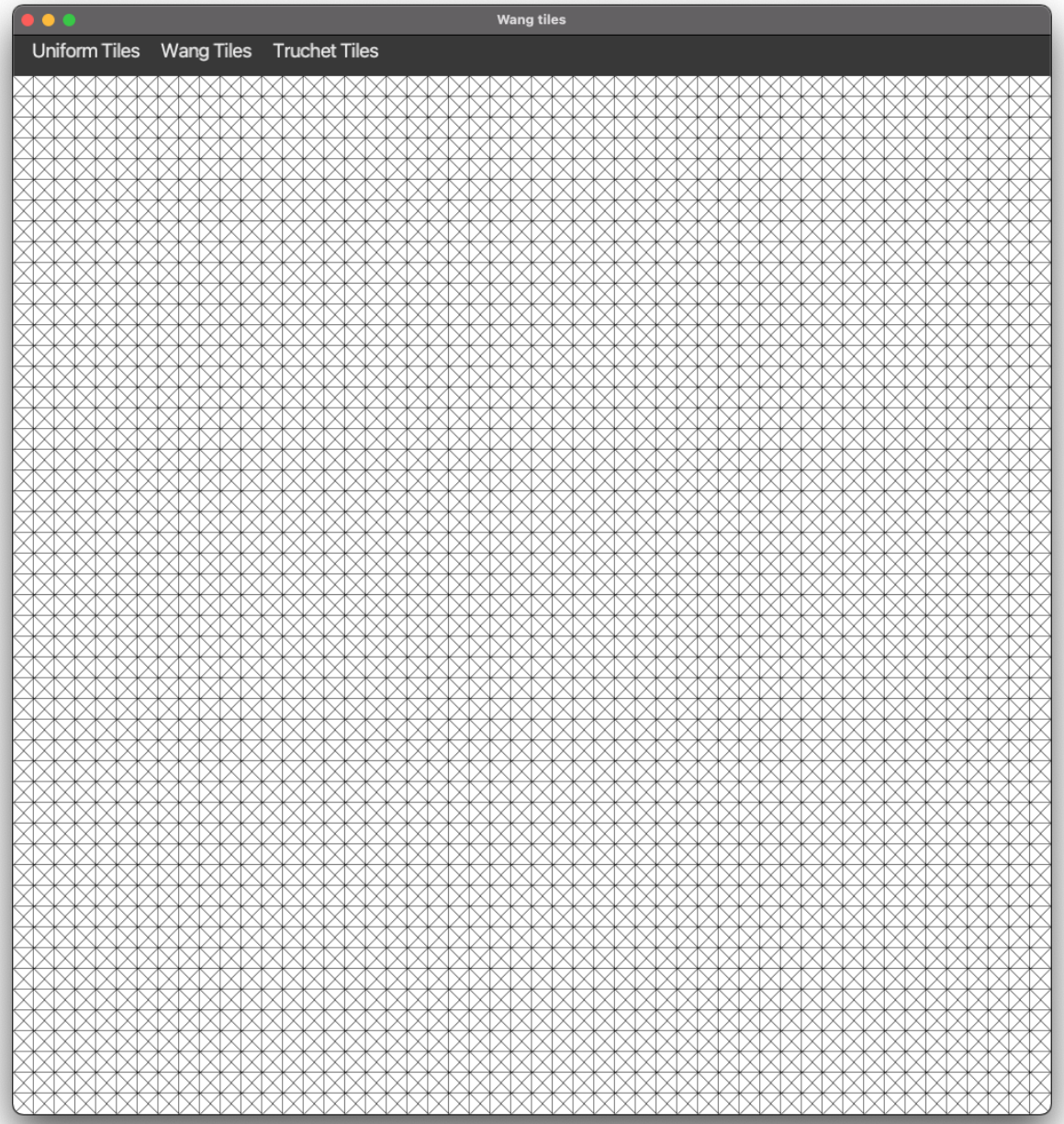
2.1 TP noté

Ce TP est à faire en 8h de séances de TP comprise entre le 18 octobre et le 7 novembre. Vous avez le droit de faire le projet seul ou en binôme. Vous avez aussi le droit de travailler en dehors des heures de TP. Il faudra que le dépôt *git* soit à jour pour le 7 novembre à 23h. Toute mise à jour après cette date sera impossible. Le TP sera évalué et comptera pour 20% de la note finale.

2.2 Consignes pour démarrer le TP

Comme pour les TP précédents, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du premier TP.

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le code cliquer deux fois sur `wang-tile` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant.



Pour exécuter les tests, il faut passer par l'onglet gradle à droite et cliquer deux fois sur '`wang-tile` -> `Tasks` -> `verification` -> `test`.

Lien vers le projet gitlab à forker pour le TP : lien

2.2.1 Respect de la propriété intellectuelle

Comme pour tout TP noté, nous vous demandons de ne pas partager votre programme, complet ou partiel, avec des membres d'autres équipes que la votre. Le non-respect de cette consigne vous expose à recevoir une **note nulle**. Tout emprunt que vous effectuez doit être proprement documenté en indiquant quelle partie de votre programme est concerné et de quelle source elle provient (nom d'un autre étudiant, site internet, *etc.*).

2.2.2 Critères d'évaluation

Ce TP sera évalué et comptera pour 20% de la note finale de l'unité d'enseignement de programmation 2. Vous serez évalué sur :

— **La propreté du code :** comme indiqué dans le cours, il est important de programmer proprement. Des répétitions de code trop visibles, des noms mal choisis ou des fonctions ayant beaucoup de lignes de code (plus de dix) vous pénaliseront. Le sujet vous donne les méthodes que vous devez absolument écrire mais il est tout à fait autoriser d'écrire des méthodes supplémentaires, de créer des constantes, ... pour augmenter la lisibilité du code. On rappelle que vous devez écrire le code en anglais.

— **La correction du code :** on s'attend à ce que votre code soit correct, c'est-à-dire respecte les spécifications dans le sujet. Vous avez tout intérêt à tester votre code pour vérifier son comportement. Les classes de tests ne sont demandées que pour la tâche 7 mais nous vous conseillons de les écrire en même temps que vous écrivez vos classes.

— **Modificateurs d'accès et attributs `final` :** dans ce sujet, on ne vous donnera pas toujours les modificateurs d'accès pour les différents éléments du code. Ce sera donc à vous de choisir l'accessibilité de certains éléments entre : `private`, `public`, `protected` et *default* (pas de mot-clé). Vous aurez aussi à choisir si vous souhaitez utiliser le mot-clé `final` sur les attributs des classes. Vous serez évalué sur ces choix.

— **Les commit/push effectués :** il vous faudra travailler en continu avec `git` et faire des *push/commit* le plus régulièrement possible. Un projet ayant très peu de *push/commit* effectués juste avant la date limite sera considéré comme suspicieux et noté en conséquence. Un minimum accepté pour ce TP sera d'au moins **2 pushs sur deux jours différents** et d'au moins **10 commits** au total. Si ces minimums ne sont pas respectés, le TP recevra une note de zéro. Vous devez faire un commit par méthode que vous codez et un push après chaque tâche. Si vous êtes en binôme, il faudra qu'il y ait des **push/commit** avec les comptes **etulab** des deux membres.

2.3 Première modification de votre dépôt

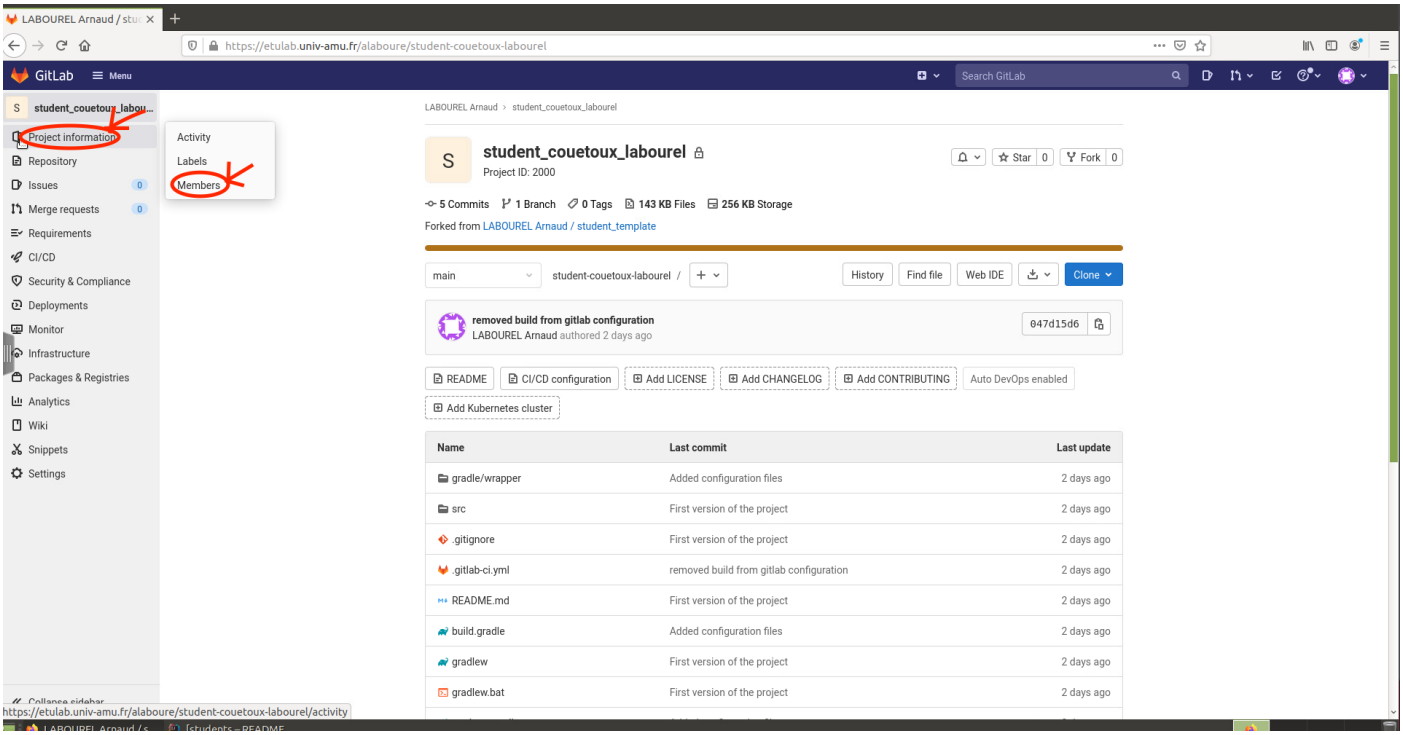
Modifier le fichier `README.md` à la racine du projet. Ce fichier devra contenir votre nom de famille et votre prénom ainsi que le nom de famille et le prénom de votre éventuel co-équipier.

2.4 Ajout chargé de TP en tant que membre

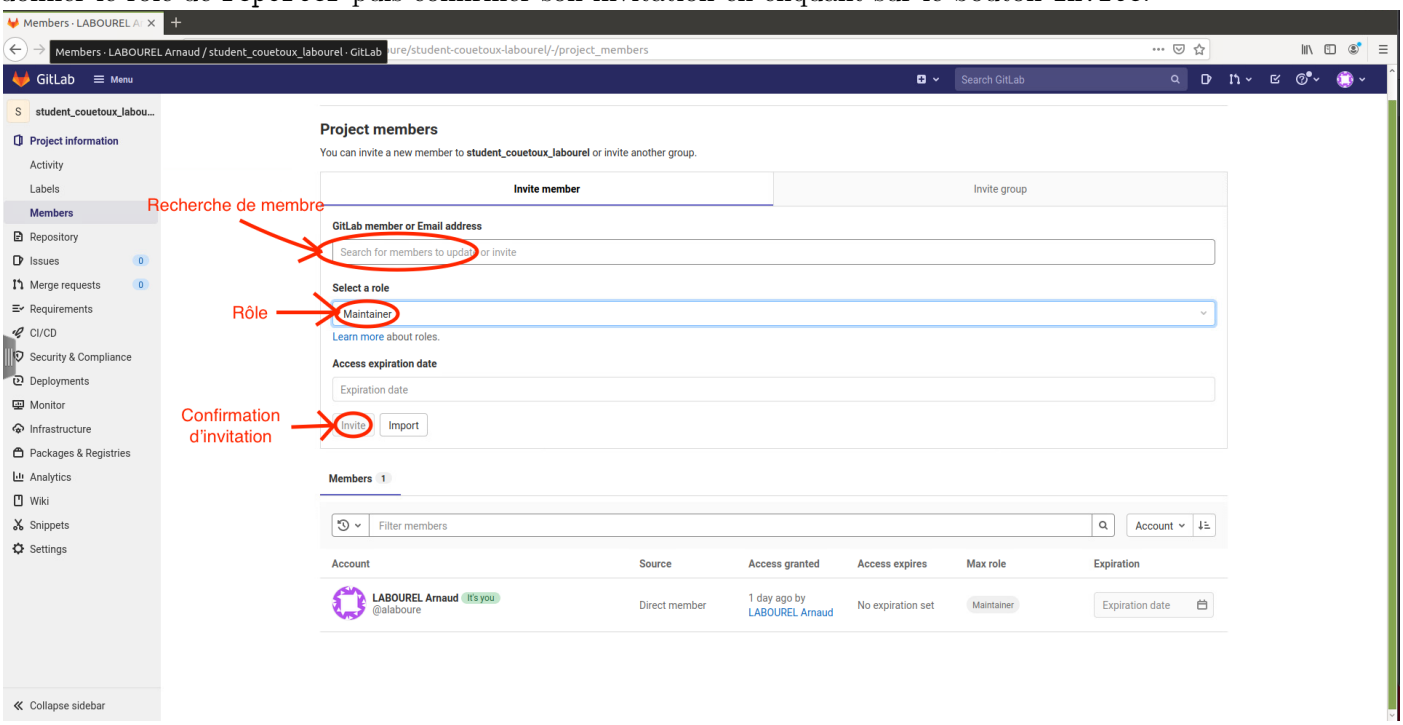
Afin que votre chargé de TP puisse vous évaluer, il vous faudra le rajouter en tant que membre. Pour rappel, les chargés de TP sont les suivants :

— TP 1 : Arnaud Labourel
— TP 2 : Pacôme Perrotin
— TP 3 : Nathanaël Eon
— TP 4 : Basile Couëtoux

Une fois le projet créé, vous pouvez rajouter le chargé de votre groupe de TP en cliquant sur `project information` dans le menu de gauche puis `members`.



Ensuite vous pouvez rechercher votre chargé de TP dans la barre dédiée. Un fois celui-ci trouvé vous pouvez lui donner le rôle de `reporter` puis confirmer son invitation en cliquant sur le bouton `invite`.



2.5 Package

Toutes les classes sauf `RandomUtil` devront être dans le package `model`.

3 Interfaces et classes déjà fournies

Dans ce projet, vous allez essentiellement travailler sur la partie modèle qui va permettre d'afficher les tuiles. On vous a fourni 5 interfaces (`Side`, `Tile`, `Square`, `Grid` et `TileGenerator`) ainsi que 5 classes implémentant ces interfaces (`EmptySide`, `EmptyTile`, `EmptySquare`, `EmptyGrid` et `EmptyTileGenerator`)

Vous trouverez dans cette partie les explication sur les interfaces et les classes qu'on vous fournit.

3.1 Définition des côtés d'une tuile

3.1.1 Interface `Side`

L'interface `Side` définit les fonctionnalités attendues d'un côté d'une tuile. Les méthodes sont les suivantes :

— `Color color()` : retourne la couleur (classe `javafx.scene.paint.Color`) du côté. C'est cette couleur qui sera utilisée pour l'affichage.

3.1.2 Classe `EmptySide`

La classe `EmptySide` correspond à un côté d'une tuile vide. Elle implémente l'interface `Side` et contient les éléments suivants :

— `Color color()` : retourne la couleur blanche (`Color.WHITE`).

— une constante `public static final Side EMPTY_SIDE` qui contient une instance de la classe

3.2 Définition d'une tuile

3.2.1 Enum `CardinalDirection`

Cette énumération définit 4 valeurs possibles correspondant aux 4 directions cardinales : `NORTH`, `EAST`, `SOUTH` et `WEST`. Elle définit aussi :

— un attribut `public final int deltaRow` correspondant à la valeur à ajouter à l'indice de ligne d'une case pour obtenir l'indice de ligne de la case voisine dans la direction considérée.

— un attribut `public final int deltaColumn` correspondant à la valeur à ajouter à l'indice de colonne d'une case pour obtenir l'indice de colonne de la case voisine dans la direction `this`.

— une méthode `public CardinalDirection oppositeDirection()` donnant la direction opposée à la direction `this`.

— une méthode `public int ordinal()` qui renvoie le numéro associé à la direction (`NORTH` $\rightarrow$ 0, `EAST` $\rightarrow$ 1, `SOUTH` $\rightarrow$ 2 et `WEST` $\rightarrow$ 3).

— une méthode `public static CardinalDirection[] values()` qui renvoie un tableau contenant les quatres directions cardinales dans les cases correspondant à leur numéro associé.

— une constante `public static final int NUMBER_OF_DIRECTIONS` qui est égale au nombre de directions possible (donc égal à 4).

3.2.2 Interface `Tile`

L'interface `Tile` définit les fonctionnalités attendues d'une tuile. La seule fonctionnalité d'une tuile est de pouvoir accéder à chacun des cotés associés à une direction. Cette interface définit donc la méthode suivante :

— `Side side(CardinalDirection direction)` qui renvoie le côté de la tuile dans la direction souhaitée.

3.2.3 Classe `EmptyTile`

La classe `EmptyTile` correspond à une tuile vide qui a donc ses quatre côtés égaux à un côté vide. Elle implémente l'interface `Tile` et contient les éléments suivants :

— `Side side(CardinalDirection direction)` renvoie une instance de `EmptySide`.

— une constante `public static Tile EMPTY_TILE` qui contient une instance de la classe.

3.3 Définition d'une case de la grille

L'interface `Square` correspond à une case de grille pouvant contenir une tuile. Cette interface définit donc les méthodes suivantes :

3.3.1 Interface `Square`

L'interface `Square` correspond à une case. Elle définit les méthodes suivantes :
— la méthode `public void put(Tile tile)` qui permet de récupérer la tuile mise dans la case.
— la méthode `Tile getTile()` qui permet de mettre une tuile dans la case.

3.3.2 Classe `EmptySquare`

La classe `EmptySquare` correspond à une case contenant une tuile vide. Elle implémente l'interface `Square` et contient les éléments suivants :
— la méthode `public void put(Tile tile)` qui ne fait rien.
— la méthode `Tile getTile()` qui renvoie une tuile vide.
— une constante `public static Square EMPTY_SQUARE` qui contient une instance de la classe.

3.4 Définition d'une grille

3.4.1 Interface `Grid`

L'interface `Grid` correspond à grille de cases. Cette interface définit donc les méthodes suivantes :
— une méthode `Square getSquare(int rowIndex, int columnIndex)` qui renvoie la case à ligne et colonne passées en arguments.
— une méthode `int getNumberOfRows()` qui renvoie le nombre de ligne de la grille.
— une méthode `int getNumberOfColumns()` qui renvoie le nombre de colonnes de la grille.
— une méthode `void fill(TileGenerator tileGenerator)` qui remplit les cases de la grille avec les tuiles générées par le générateur passé en argument.

3.4.2 Classe `EmptyGrid`

La classe `EmptyGrid` correspond à une grille ne contenant que des cases vides. Elle implémente l'interface `Grid` et contient les éléments suivants :
— un constructeur `public EmptyGrid(int numberOfRows, int numberOfColumns)` qui construit une grille vide avec les nombres de lignes et de colonnes donnés en arguments.
— des attributs `private int numberOfRows` et `private int numberOfColumns` qui correspondent respectivement au nombre de lignes de la grille et au nombre de colonnes de la grille.
— une méthode `Square getSquare(int rowIndex, int columnIndex)` qui renvoie une case vide.
— une méthode `int getNumberOfRows()` qui renvoie le nombre de lignes de la grille.
— une méthode `int getNumberOfColumns()` qui renvoie le nombre de colonnes de la grille.
— une méthode `void fill(TileGenerator tileGenerator)` qui ne fait rien.

3.5 Définition d'un générateur de tuiles

3.5.1 Interface `TileGenerator`

Un générateur de tuile permet de générer des tuiles. Elle définit la méthode suivante :
— une méthode `Tile nextTile(Square square)` qui renvoie une tuile pouvant être placée dans la case passée en argument.

3.5.2 Classe `EmptyTileGenerator`

La classe `EmptyTileGenerator` correspond à une générateur ne générant que des tuiles vides. Elle implémente l'interface `TileGenerator` et contient les éléments suivants :

— un constructeur `public EmptyTileGenerator()`.
— une méthode `Tile nextTile(Square square)` qui renvoie une tuile vide.

4 Tâche 1 : classes non-vides et générateurs de tuiles colorée uniformément

Dans cette première tâche, vous aurez à implémenter des nouvelles classes implémentant les 5 interfaces `Side`, `Tile`, `Square`, `Grid` et `TileGenerator` afin d'afficher une grille contenant des cases de la même couleur. Afin de réaliser cette tâche, vous devez implémenter les classes suivantes qui sont décrites ci-après :

— `ColoredSide`
— `UniformTile`
— `ArraySquare`
— `UniformTileGenerator`

Vous devrez aussi implémenter les classes de tests correspondantes (dans le répertoire `src/test/java/model/`) qui devront vérifier le comportement de ces quatre classes :

— `ColoredSideTest`
— `UniformTileTest`
— `ArraySquareTest`
— `UniformTileGeneratorTest`

4.1 Classe `ColoredSide`

La classe `ColoredSide` correspond à un côté d'une tuile ayant une couleur. Elle implémente l'interface `Side` et contient les éléments suivants :

— un attribut `private final Color color` qui contient la couleur du côté.
— un constructeur `public ColoredSide(Color color)` qui construit un côté avec la couleur donné en argument.
— `Color color()` qui renvoie la couleur du côté.

4.2 Classe `UniformTile`

La classe `UniformTile` correspond à une tuile dont les quatre côtés sont identiques. Elle implémente l'interface `Tile` et contient les éléments suivants :

— un attribut `private final Side side` qui contient le côté unique de la tuile.
— un constructeur `public UniformTile(Side side)` qui construit une tuile avec le côté passé en argument.
— une méthode `Side side(CardinalDirection direction)` qui renvoie le côté unique de la tuile pour chacune des quatre directions.

4.3 Classe `ArraySquare`

La classe `ArraySquare` correspond à une case d'une grille. Elle implémente l'interface `Square` et contient les éléments suivants (le nom de cette classe sera expliqué dans la suite de cette planche) :

— un attribut `private Tile tile` qui contient la tuile de la case.
— un constructeur `public ArraySquare()` qui construit une case contenant une tuile vide.
— la méthode `public void put(Tile tile)` qui met la tuile passée en argument dans la case.
— la méthode `Tile getTile()` qui renvoie la tuile contenu dans la case.

4.4 Classe `ArrayGrid`

La classe `ArrayGrid` correspond à une grille de cases pouvant contenir des tuiles. Elle implémente l'interface `Grid` et contient les éléments suivants :

- un attribut `private final Square[][] squares` qui correspond à la matrice des cases de la grille.
- des attributs `private int numberOfRows` et `private int numberOfColumns` qui correspondent respectivement au nombre de lignes de la grille et au nombre de colonnes de la grille.
- un constructeur `public ArrayGrid(int numberOfRows, int numberOfColumns)` qui construit une grille avec les nombres de lignes et de colonnes donnés en arguments. Ce constructeur initialise la matrice `squares` aux bonnes dimensions et remplie les cases de la matrice d'instances de `ArraySquare`.
- une méthode `Square getSquare(int rowIndex, int columnIndex)` qui renvoie la case à la ligne et colonne passées en arguments.
- une méthode `int getNumberOfRows()` qui renvoie le nombre de lignes de la grille.
- une méthode `int getNumberOfColumns()` qui renvoie le nombre de colonnes de la grille.
- une méthode `void fill(TileGenerator tileGenerator)` qui pour chaque case de la grille génère une tuile (via le générateur passé en argument) en donnant la case en argument puis met cette tuile dans la case.

4.5 Classe `UniformTileGenerator`

La classe `UniformTileGenerator` correspond à un générateur ne générant que des tuiles de la même couleur. Elle implémente l'interface `TileGenerator` et contient les éléments suivants :

- un attribut `private final Tile tile` qui correspond à la tuile que doit générer le générateur. Cette tuile est donc une instance d'`UniformTile` avec comme côté une instance de `ColoredSide` ayant la bonne couleur.
- un constructeur `public UniformTileGenerator(Color color)` qui initialise l'attribut `tile` pour qu'il corresponde à une tuile de la couleur demandée.
- une méthode `Tile nextTile(Square square)` qui renvoie l'attribut `tile`.

4.6 Tests unitaires de vos classes

Comme indiqué précédemment, vous devez écrire les classes de tests suivantes à placer dans `src/test/java/model/` :

- `ColoredSideTest`
- `UniformTileTest`
- `ArraySquareTest`
- `UniformTileGeneratorTest`

Pour tester des classes, il est judicieux de redéfinir dans les classes testées les méthodes `boolean equals(Object o)` et `String toString()` qui sont utilisées respectivement pour les appels de la forme `assertThat(o1).isEqual(o2)` et l'affichage des résultats des tests. Nous vous conseillons de les générer avec `IntelliJ` (clic droit `generate`) ou bien de suivre le format ci-dessous.

```
package model;

import java.util.Objects;

public class MyClass {
  int primitiveAttribute;
  String nonPrimitiveAttribute;

  public MyClass(int primitiveAttribute, String nonPrimitiveAttribute) {
```

```
    this.primitiveAttribute = primitiveAttribute;
    this.nonPrimitiveAttribute = nonPrimitiveAttribute;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    MyClass myClass = (MyClass) o;
    return primitiveAttribute == myClass.primitiveAttribute
      && Objects.equals(nonPrimitiveAttribute, myClass.nonPrimitiveAttribute);
  }


  @Override
  public String toString() {
    return "MyClass{" +
            "primitiveAttribute=" + primitiveAttribute +
            ", nonPrimitiveAttribute='" + nonPrimitiveAttribute + '\'' +
            '}';
  }
}
```

4.7 Test visuel de vos classes

Une fois que vous avez codé les classes demandés et les classes de tests correspondantes, il va falloir modifier deux autres classes afin de pouvoir tester visuellement votre code.

La première classe à modifier est la classe `GridCanvas` dans le répertoire `view`. Il vous faut changer dans le constructeur de la classe l'appel au constructeur `new EmptyGrid(numberOfRows, numberOfColumns)` par `new ArrayGrid(numberOfRows, numberOfColumns)`. Une fois cela fait vous pouvez supprimer le commentaire `TODO`.

La deuxième classe à modifier est la classe `GridController`. Vous devez tout d'abord changer les couleurs définies dans la classe. Les couleurs par défaut sont les suivantes :

```
public static final Color COLOR_FIRST_NAME_ONE = Color.RED;
public static final Color COLOR_LAST_NAME_ONE = Color.BLUE;
public static final Color COLOR_FIRST_NAME_TWO = Color.GREEN;
public static final Color COLOR_LAST_NAME_TWO = Color.YELLOW;
```

La règle est d'utiliser des couleurs qui correspondent à vos prénoms et noms.

- la couleur `COLOR_FIRST_NAME_ONE` doit correspondre à la première lettre du prénom du premier membre du projet.
- la couleur `COLOR_LAST_NAME_ONE` doit correspondre à la première lettre du nom de famille du premier membre du projet.
- la couleur `COLOR_FIRST_NAME_TWO` doit correspondre à la première lettre du prénom du deuxième membre du projet ou bien la deuxième lettre du prénom de l'unique membre du projet si celui-ci ne comporte qu'un membre.
- la couleur `COLOR_LAST_NAME_TWO` doit correspondre à la première lettre du nom de famille du deuxième membre du projet ou bien la deuxième lettre du nom de famille de l'unique membre du projet si celui-ci ne comporte qu'un membre.

La table ci-dessous vous donne la correspondance entre lettre et couleur. Il faut prendre la couleur dans la première colonne pour la première occurrence de la lettre et les colonnes suivantes si la lettre est déjà apparue.

Lettre	Couleur 1	Couleur 2	Couleur 3	Couleur 4
A	ALICEBLUE	ANTIQUEWHITE	AQUA	AQUAMARINE
B	BLUE	BISQUE	BLACK	BEIGE
C	CADETBLUE	CHARTREUSE	CHOCOLATE	CORNFLOWERBLUE
D	DARKBLUE	DARKGRAY	DARKKHAKI	DARKORANGE
E	DIMGREY	DODGERBLUE	DEEPPINK	DARKTURQUOISE
F	FIREBRICK	FLORALWHITE	FORESTGREEN	FUCHSIA
G	GREEN	GHOSTWHITE	GOLD	GOLDENROD
H	GRAY	HOTPINK	GAINSBORO	HONEYDEW
I	INDIANRED	INDIGO	IVORY	GREENYELLOW
J	BROWN	BLUEVIOLET	BLUE	BURLYWOOD
K	MOCCASIN	MISTYROSE	MINTCREAM	MIDNIGHTBLUE
L	LAWNGREEN	LAVENDERBLUSH	LEMONCHIFFON	LAVENDER
M	MAGENTA	MAROON	MEDIUMAQUAMARINE	MIDNIGHTBLUE
N	NAVAJOWHITE	NAVY	TURQUOISE	TRANSPARENT
O	OLDLACE	OLIVE	OLIVEDRAB	ORANGE
P	PAPAYAWHIP	PEACHPUFF	PERU	PINK
Q	LIGHTGRAY	LIGHTGREEN	LIGHTPINK	LIGHTSKYBLUE
R	RED	ROSYBROWN	ROYALBLUE	LIGHTSALMON
S	SADDLEBROWN	SALMON	SANDYBROWN	SEAGREEN
T	TAN	TEAL	THISTLE	TOMATO
U	LIGHTBLUE	LIGHTCORAL	LIGHTCYAN	LIGHTCORAL
V	MEDIUMBLUE	MEDIUMSLATEBLUE	MEDIUMSPRINGGREEN	MEDIUMSEAGREEN
W	WHEAT	WHITESMOKE	LIGHTSLATEGRAY	SLATEBLUE
X	MEDIUMORCHID	MEDIUMPURPLE	MEDIUMTURQUOISE	MEDIUMVIOLETRED
Y	YELLOW	YELLOWGREEN	LIGHTPINK	LIGHTGREY
Z	PALEGOLDENROD	PALEGREEN	PALETURQUOISE	PALEVIOLETRED

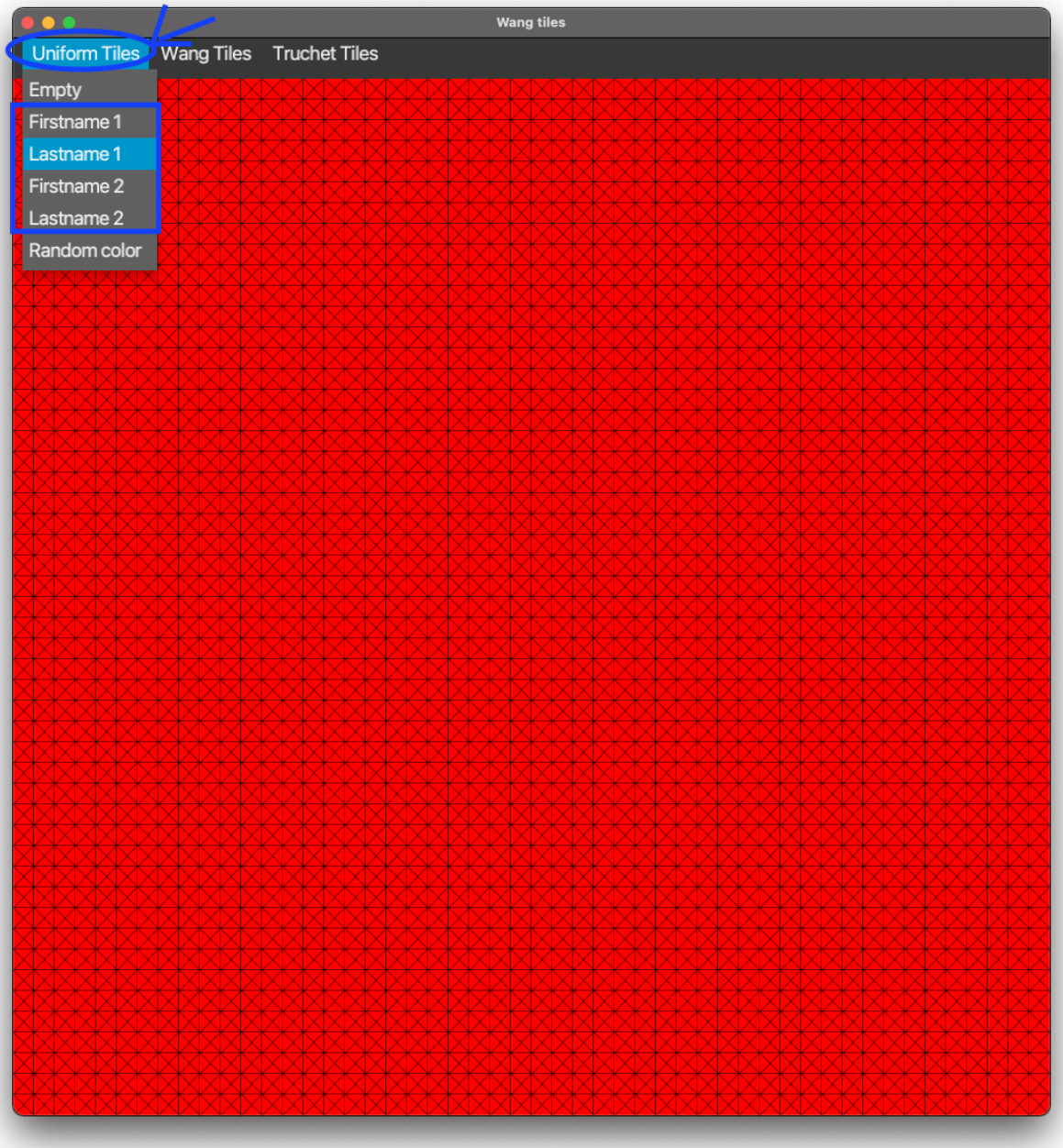
Les couleurs de base correspondent par exemple à une équipe composée de **R**obert **B**artolini et de **G**érard **Y**amazaki. Les couleurs suivantes pourraient correspondre **G**érard **G**ameur et **G**érard **M**anvussa :

```
public static final Color COLOR_FIRST_NAME_ONE = Color.GREEN;
public static final Color COLOR_LAST_NAME_ONE = Color.GHOSTWHITE;
public static final Color COLOR_FIRST_NAME_TWO = Color.GOLD;
public static final Color COLOR_LAST_NAME_TWO = Color.MAGENTA;
```

La deuxième modification à faire est de décommenter le code des méthodes suivantes (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait) :

- `public void updateFirstNameOneUniformTile()`
- `public void updateFirstNameTwoUniformTile()`
- `public void updateLastNameOneUniformTile()`
- `public void updateLastNameTwoUniformTile()`

Une fois que vous avez fait toutes ces modifications, vous pouvez tester votre code en relançant l'application et en utilisant le menu `Uniform tiles` qui va vous permettre de générer des tuiles des 4 couleurs que vous avez définies (cf l'image ci-dessous).



5 Tâche 2 : générateurs de tuiles uniformes aléatoires

Pour cette tâche, vous allez devoir générer des tuiles uniformes de manière aléatoire.

5.1 Classe utilitaire `RandomUtil`

Pour se simplifier la vie, on va tout d'abord définir une classe utilitaire `RandomUtil` que vous allez placer dans le répertoire `util` et qui va nous permettre de tirer un élément au hasard dans un tableau ou une liste. Cette classes contiendra les élément suivants :

— un constructeur `private RandomUtil()` qui ne fait rien et ne sera jamais appelé.

— une méthode `public static <T> T randomElement(List<T> elements, Random random)` qui tire au hasard un élément de la liste à l'aide du générateur aléatoire `random`.
— une méthode `public static <T> T randomElement(T[] elements, Random random)` qui tire au hasard un élément du tableau à l'aide du générateur aléatoire `random` (cette méthode sera utile par la suite).

Pour cette classe, on utilisera la méthode `int nextInt(int bound)` de la classe `Random` qui renvoie un entier au hasard compris entre `0` et `bound - 1` inclus.

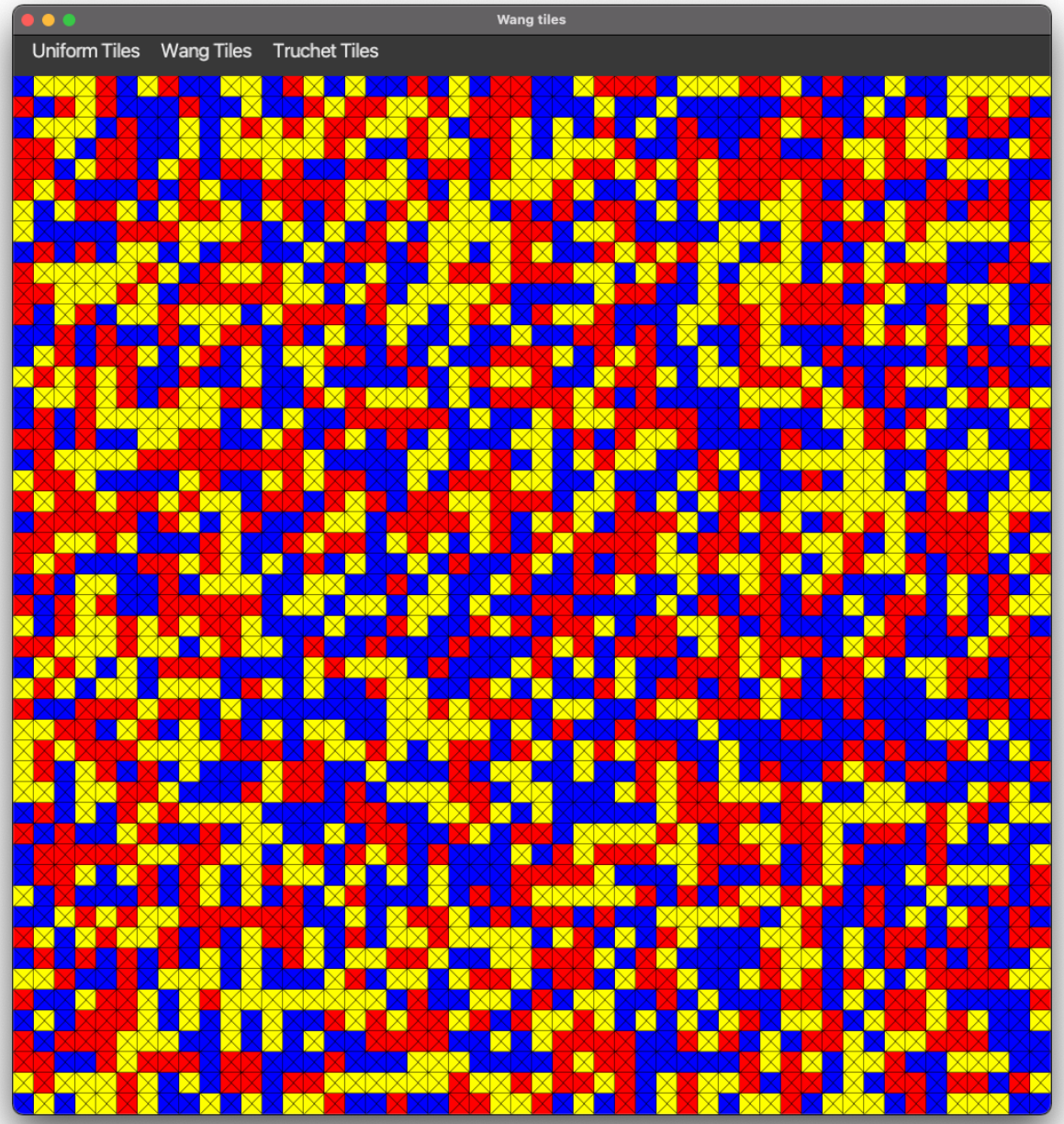
5.2 Classe `RandomUniformTileGenerator`

La classe `RandomUniformTileGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles uniformes d'une couleur choisi au hasard parmis une liste de couleur donnée en argument lors de la construction d'une instance.

— un attribut `private final List<Tile> tiles` qui correspond à la liste de tuiles que peut générer le générateur. Ces tuiles sont des instances d'`UniformTile` avec chacune comme côté une instance de `ColoredSide` ayant une des couleurs possibles.
— un attribut `private final Random randomGenerator` qui contient le générateur aléatoire utilisé pour tirer au hasard une tuile.
— un constructeur `public RandomUniformTileGenerator(List<Color> colors, Random randomGenerator)` qui initialise :
  — l'attribut `tiles` pour qu'il corresponde à la liste des tuiles des couleurs demandées.
  — l'attribut `randomGenerator` avec le `randomGenerator` passé en argument.
— une méthode `Tile nextTile(Square square)` qui renvoie une tuile tirée au hasard dans la liste `tiles`.

5.3 Test du code

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateRandomColorUniformTile()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Uniform Tiles` → `Random color`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :



6 Tâche 3 : tuiles de Wang aléatoires

Pour cette tâche, vous allez devoir générer des tuiles de Wang (tuiles pouvant avoir des couleurs différentes pour chaque côté) de manière aléatoire.

6.1 Classe `WangTile`

La classe `WangTile` correspond à une tuile dont chacun des côtés est associé à une couleur. Elle implémente l'interface `Tile` et contient les éléments suivants :

— un attribut `private final Side[] sides` qui est un tableau de taille `CardinalDirection.NUMBER_OF_DIRECTIONS` et qui va contenir dans la case d'indice `i` le côté

associé à la direction de numéro `i` (donné par la méthode `ordinal()` de `CardinalDirection`).
- un constructeur `public WangTile(Side[] sides)` qui construit une tuile avec les côtés passés en argument.
- une méthode `Side side(CardinalDirection direction)` qui renvoie le côté de la tuile pour la direction demandée.

6.2 Classe `WangTileTest`

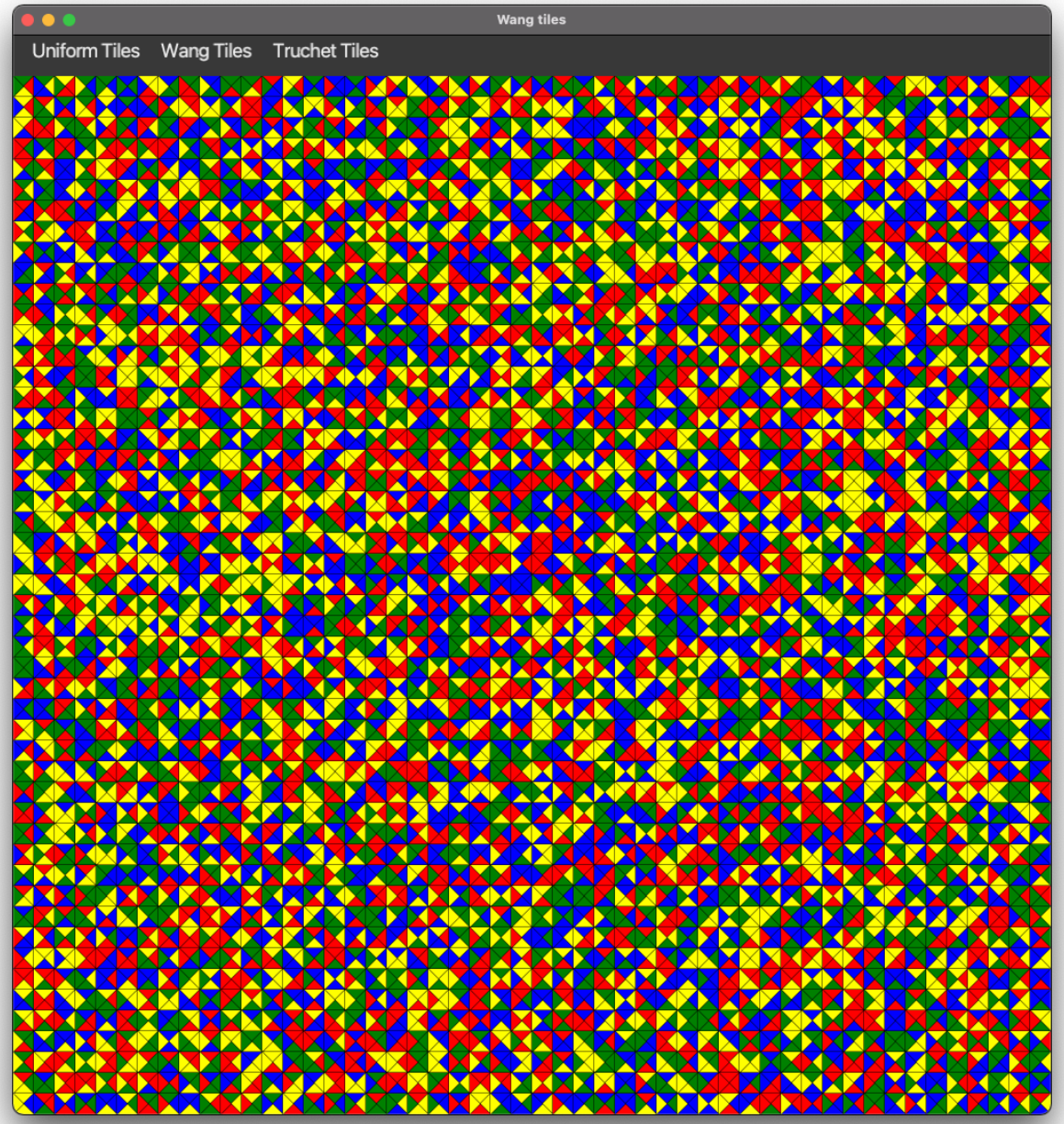
Il vous faut écrire une classe de test testant que le comportement de la classe `WangTile` est bien celui décrit ci-dessus.

6.3 Classe `RandomWangTileGenerator`

La classe `RandomWangTileGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles de Wang dont les couleurs des côtés sont choisies au hasard parmi les couleurs possibles.
- un attribut `private final List<Side> availableSides` qui correspond aux côtés des couleurs possibles.
- un attribut `private final Random randomGenerator` qui contient le générateur aléatoire utilisé pour tirer au hasard une tuile.
- un constructeur `public RandomWangTileGenerator(List<Color> colors, Random randomGenerator)` qui initialise :
  - l'attribut `availableSides` pour qu'il corresponde à une liste de côtés contenant un côté pour chaque couleur de la liste passé en argument.
  - l'attribut `randomGenerator` avec le `randomGenerator` passé en argument.
- une méthode `Tile nextTile(Square square)` qui renvoie une instance de `WangTile` pour laquelle on a tiré pour chaque direction un côté au hasard parmi `availableSides`.

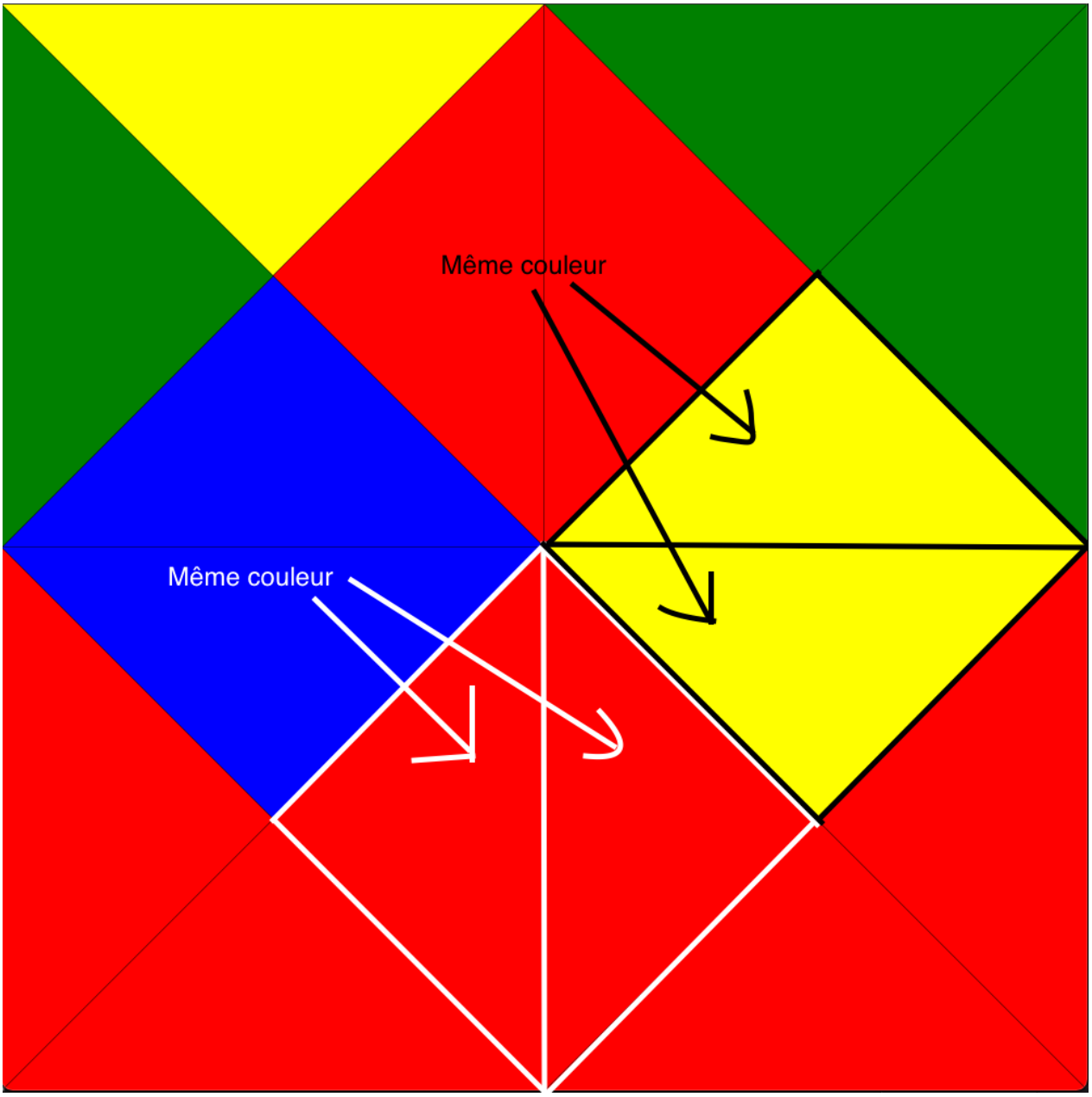
6.4 Test du code

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateRandomWangTile()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Wang tiles` → `Random`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :



7 Tâche 4 : contraintes sur le placement

Pour cette tâche vous allez devoir modifier des interfaces et des classes qui existent afin de pouvoir gérer facilement les contraintes de placements que les tuiles doivent respecter. En effet, le concept de ces tuiles est qu'on ne peut placer une tuile que si ces côtés correspondent aux côtés des tuiles voisines déjà posées. Par exemple, lorsqu'on pose la tuile en bas à droite dans le dessin suivant il faut que le côté nord de la tuile soit de la même couleur que le côté sud de la tuile voisine au nord et que le côté ouest de la tuile soit de la même couleur que le côté est de la tuile voisine à l'ouest.



Il va falloir donc modifier les interfaces et classes suivantes :
— interface `Side`
— classe `EmptySide`
— Classe `ColoredSide`
— Interface `Square`
— Classe `EmptySquare`
— Classe `ArraySquare`
— Classe `ArrayGrid`
Il faut aussi que vous créiez une classe `RandomConstrainedWangTileGenerator`.

7.1 Interface `Side`

Il faut rajouter les méthodes suivantes à l'interface `Side` :
— `List<Side> compatibleSides(List<Side> sides)` : filtre la liste `sides` donnés en argument pour ne

garder que les côtés compatibles.
— `boolean accept(Side side)` : retourne un booléen indiquant si le côté accepte le côté `side` passé en argument comme côté compatible.

7.2 Classe `EmptySide`

Il faut rajouter les méthodes suivantes à la classe `EmptySide` :
— `List<Side> compatibleSides(List<Side> sides)` : considère que tous les côtés sont compatibles et renvoie donc la liste `sides` passé en argument.
— `boolean accept(Side side)` : accepte tous les côtés et renvoie donc toujours `true`.

7.3 Classe `ColoredSide`

Il faut rajouter les méthodes suivantes à la classe `ColoredSide` :
— `boolean accept(Side side)` : accepte le côté `side` et donc renvoie `true` si celui-ci a la même couleur que `this`. Renvoie `false` autres cas.
— `List<Side> compatibleSides(List<Side> sides)` : filtre la liste donné en argument pour ne garder que les côtés acceptés.

7.4 Interface `Square`

Il faut rajouter les méthodes suivantes à l'interface `Square` :
— `void setNeighbor(Square neighbor, CardinalDirection direction)` qui permet de fixer la case voisine de la case dans la direction souhaitée.
— `Square getNeighbor(CardinalDirection direction)` qui permet de récupérer la case voisine le voisin de la case dans la direction souhaitée.
— `List<Side> compatibleSides(List<Side> sides, CardinalDirection direction)` qui filtre la liste des côtés pour ne garder que les côtés qui sont acceptés par le côté incident de la tuile de la case voisine dans la direction. Si par exemple la direction donnée correspond au nord alors la liste est filtrée pour ne garder que les côtés acceptés par le côté sud de la tuile de la case voisine au nord.
— `List<Tile> compatibleTiles(List<Tile> tiles)` qui filtre la liste des tuiles pour ne garder que les tuiles dont les quatre côtés sont compatibles avec les tuiles des cases voisines.

7.5 Classe `EmptySquare`

Il faut rajouter les méthodes suivantes à la classe `EmptySquare` :
— `void setNeighbor(Square neighbor, CardinalDirection direction)` qui ne fait rien.
— `Square getNeighbor(CardinalDirection direction)` qui renvoie une instance d'`EmptySquare`.
— `List<Side> compatibleSides(List<Side> sides, CardinalDirection direction)` qui renvoie la liste donnée en argument.
— `List<Tile> compatibleTiles(List<Tile> tiles)` qui renvoie la liste donnée en argument.

7.6 Classe `ArraySquare`

Il faut rajouter les éléments à la classe `ArraySquare` :
— un attribut `private final Square[] neighbors` de longueur `CardinalDirection.NUMBER_OF_DIRECTIONS` contenant la liste des voisins de la case.
— changer le constructeur pour qu'il initialise les `neighbors` avec des instances d'`EmptySquare`.
— la méthode `void setNeighbor(Square neighbor, CardinalDirection direction)` qui permet de fixer la case voisine de la case dans la direction souhaitée.
— la méthode `Square getNeighbor(CardinalDirection direction)` qui permet de récupérer la case voisine le voisin de la case dans la direction souhaitée.

— la méthode `List<Side> compatibleSides(List<Side> sides, CardinalDirection direction)` qui filtre la liste des côtés pour ne garder que les côtés qui sont acceptés par le côté incident de la tuile de la case voisine dans la direction. Si par exemple la direction donnée correspond au nord alors la liste est filtrée pour ne garder que les côtés acceptés par le côté sud de la tuile de la case voisine au nord.
— la méthode `List<Tile> compatibleTiles(List<Tile> tiles)` qui filtre la liste des tuiles pour ne garder que les tuiles dont les quatre côtés sont compatibles avec les tuiles des cases voisines.

7.7 Classe `ArrayGrid`

Il faut changer le constructeur d'`ArrayGrid` pour initialiser les voisins des cases aux cases voisines.

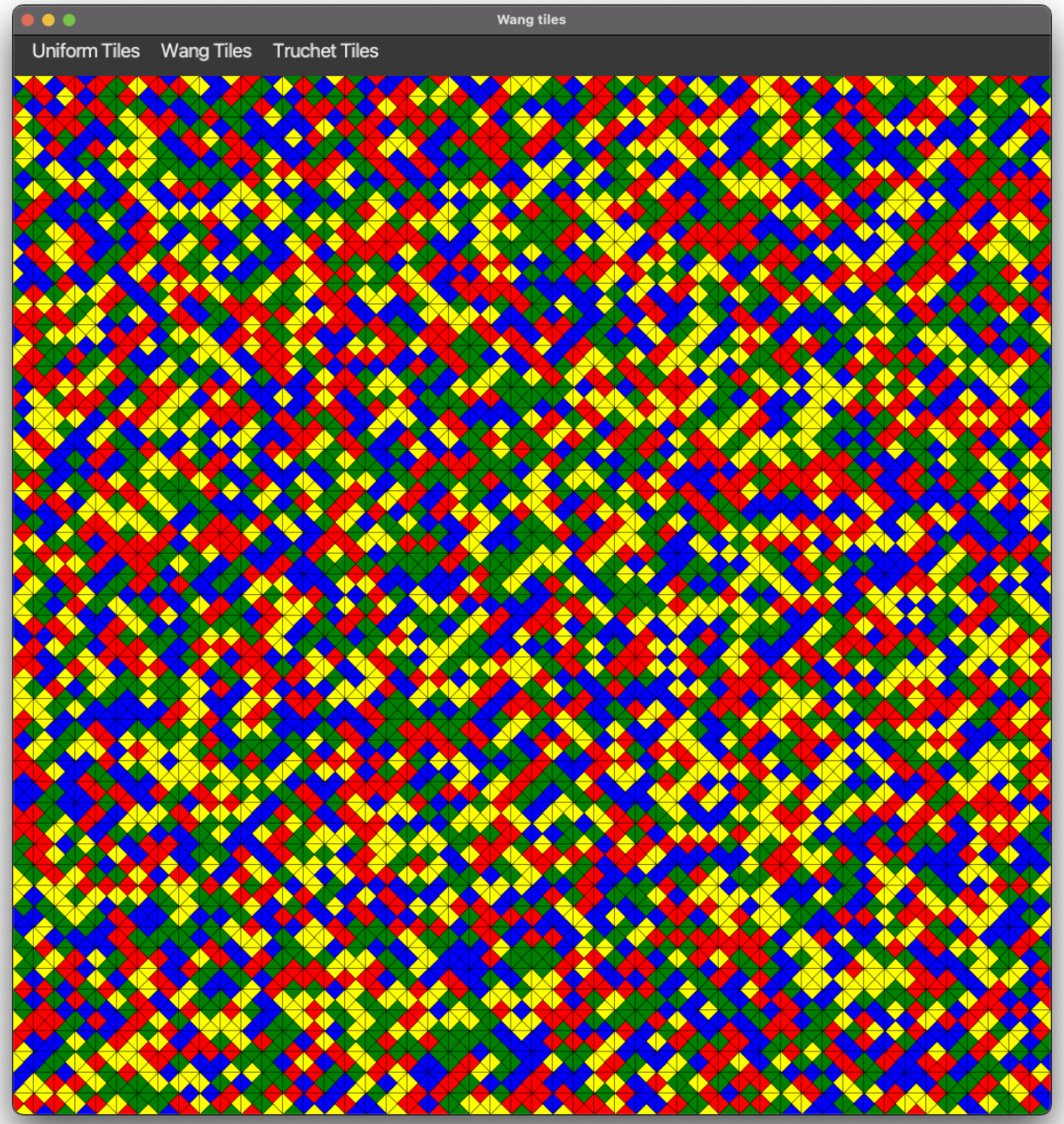
7.8 Classe `RandomConstrainedWangTileGenerator`

La classe `RandomConstrainedWangTileGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles de Wang dont les couleurs des côtés sont compatibles avec les tuiles voisines déjà posées et qui sont choisis au hasard parmi les couleurs possibles lorsqu'il n'y a pas de contraintes sur le côté.

— un attribut `private final List<Side> availableSides` qui correspond aux côtés des couleurs possibles.
— un attribut `private final Random randomGenerator` qui contient le générateur aléatoire utilisé pour tirer au hasard une tuile.
— un constructeur `public RandomConstrainedWangTileGenerator(List<Color> colors, Random randomGenerator)` qui initialise :
  — l'attribut `availableSides` pour qu'il corresponde à une liste de côtés contenant un côté pour chaque couleur de la liste passé en argument.
  — l'attribut `randomGenerator` avec le `randomGenerator` passé en argument.
— une méthode `Tile nextTile(Square square)` qui renvoie une instance de `WangTile` pour laquelle les côtés sont tirés au hasard parmi les côtés d'`availableSides` qui sont compatibles avec les contraintes de la case `square`.

7.9 Test du code

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateRandomConstrainedWangTile()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Wang Tiles` → `Random constrained`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :



8 Tâche 5 : `Iterable` et tuiles de Wang parmi un jeu de tuiles

Le but de cette tâche est de rendre itérable la grille et de remplir la grille à partir d'un ensemble de tuiles.

8.1 Interface `Grid`

Changer l'interface `Grid` pour qu'elle étende l'interface `Iterable<Square>`.

8.2 Classe `EmptyGrid`

Faites les modifications suivantes :

— Ajouter une méthode `public Iterator<Square> iterator()` qui renvoie une instance de `EmptySquareGridIterator`.

8.3 Classe `EmptySquareGridIterator`

Le rôle de `EmptySquareGridIterator` est de permettre de parcourir les cases de la grille. La grille étant vide, une instance de `EmptySquareGridIterator` ne parcours pas d'éléments. Elle devra implémenter `Iterator<Square>` de la manière suivante :

— un constructeur `EmptySquareGridIterator()`.
— une méthode `public Square next()` : lève une exception de type `NoSuchElementException`.
— `public boolean hasNext()` qui renvoie `false`.
— des attributs que vous devrez déterminer.

8.4 Classe `ArrayGrid`

Faites les modifications suivantes :

— Ajouter une méthode `public Iterator<Square> iterator()` qui renvoie une instance de `SquareGridIterator`.
— utiliser une boucle `for(Square square : this)` dans la méthode `fill`.

8.5 Classe `SquareGridIterator`

Le rôle de `SquareGridIterator` est de permettre de parcourir les cases de la grille. Une instance de cette classe devra stocker assez d'information pour se souvenir de la position de la prochaine case à parcourir. La classe devra implémenter l'interface `Iterator<Square>` de la manière suivante :

— un constructeur `SquareGridIterator(ArrayGrid grid)`.
— une méthode `public Square next()` : renvoie la prochaine case à parcourir. Les cases devront être parcourues lignes par ligne, par ordre croissant du numéro de colonne dans chaque ligne et par nombre croissant de numéro de ligne. La première case renvoyée sera donc celle de coordonnées `(0,0)` puis `(0,1)`, ..., `(0,numberOfRows-1)`, `(1,0)`, `(1,1)`, ..., jusqu'à la dernière case `(numberOfColumns-1,numberOfRows-1)`. S'il ne reste plus de cases à parcourir lors de l'appel, la méthode devra lever une exception de type `NoSuchElementException`.
— `public boolean hasNext()` qui renvoie `true` s'il reste des cases à parcourir et `false` sinon.
— des attributs que vous devrez déterminer.

8.6 Classe `RandomTileSetGenerator`

La classe `RandomTileSetGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles de Wang dont les côtés sont compatibles avec les tuiles voisines et qui sont choisies au hasard parmi un ensemble de tuiles donnés.

— un attribut `List<Tile> availableTiles` qui à l'ensemble des tuiles utilisables pour paver la grille.
— un constructeur `RandomTileSetGenerator(Iterable<Tile> tiles, Random randomGenerator)` qui initialise :
  — l'attribut `availableTiles` pour qu'il corresponde à une liste de tuiles correspondant aux tuiles contenus dans `tiles`.
  — l'attribut `randomGenerator` avec le `randomGenerator` passé en argument.
— une méthode `Tile nextTile(Square square)` qui renvoie une instance de `WangTile` appartenant à `availableTiles` qui est tirée au hasard parmi les tuiles compatibles avec la case `Square`.

8.7 Test du code

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateRandomWangTileSet()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Wang tiles` → `Random tile set`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :

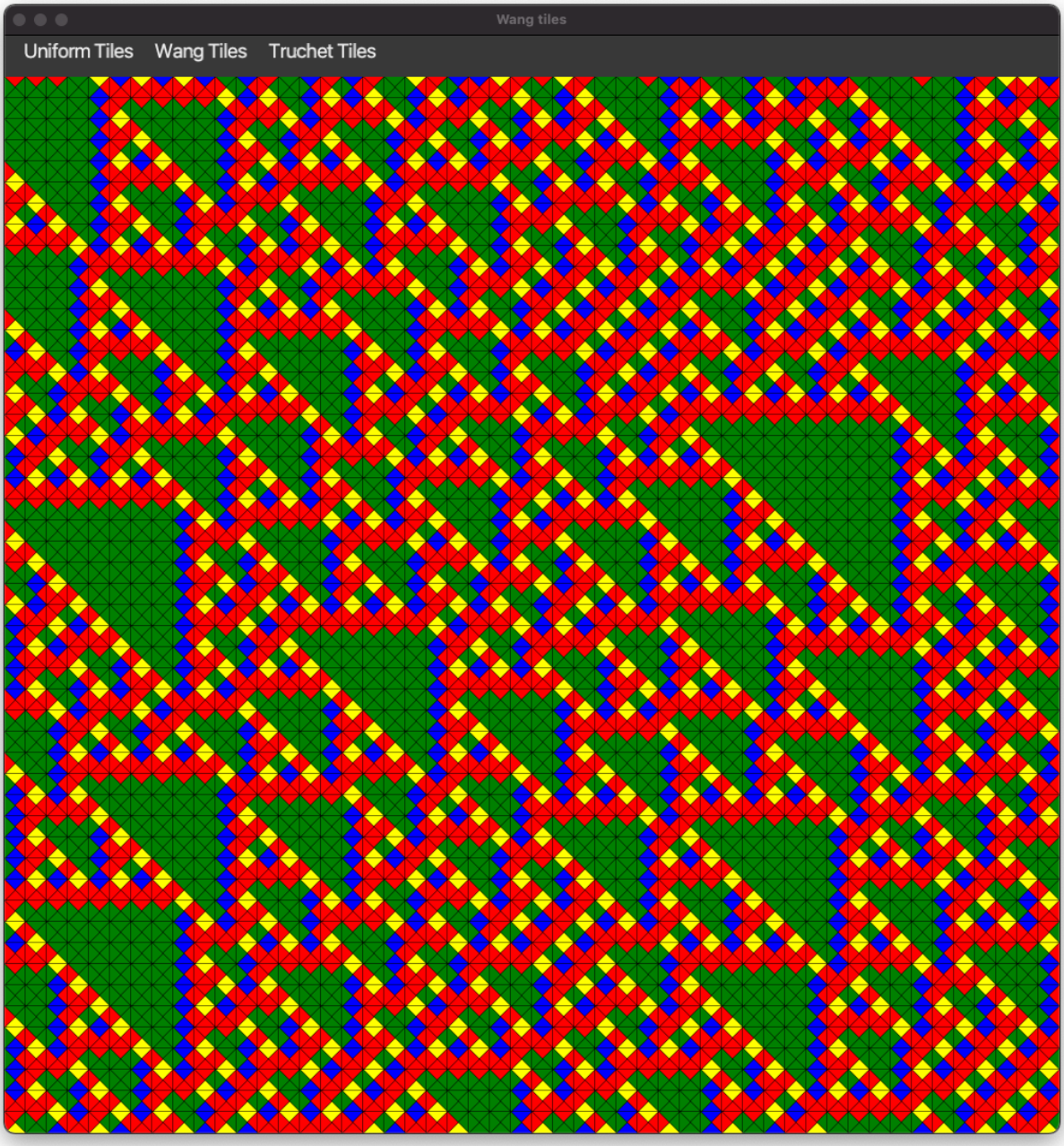


FIGURE 1 – Une tuile de Truchet

9 Tâche 6 : tuiles de Truchet et rotation

9.1 Classe `TruchetTile`

Le but de cette classe est de coder des tuiles de Truchet qui sont des tuiles ayant seulement deux couleurs (une pour les côtés nord et ouest et une autre pour les côtés sud et est). Voir figure ci-dessous pour un exemple d'une telle tuile. La classe devra contenir un constructeur `public TruchetTile(Side northWestColor, Side southEastColor)` qui construit une tuile avec les deux couleurs données. La classe `TruchetTile` devra implémenter l'interface `Tile`. C'est à vous de définir ses attributs et méthodes.

9.2 Enum `Rotation`

Définissez une `enum Rotation` avec les éléments suivants :

- un attribut `private final int numberOfQuarterTurns` qui donne le nombre de quarts de tour de la rotation.
- quatre valeurs `NO_TURN`, `QUARTER_TURN`, `HALF_TURN` et `THREE_QUARTER_TURN` qui correspondent respectivement à des rotations de `0`, `1`, `2` et `3` quarts de tour.
- un constructeur `Rotation(int numberOfQuarterTurns)` qui initialise `numberOfQuarterTurns`.
- une méthode `public CardinalDirection rotatedDirection(CardinalDirection direction)` qui renvoie la direction obtenue en appliquant la rotation dans le sens des aiguilles d'une montre. les exemples suivants vous donne les valeurs attendues que vous pourrez tester :
  - `Rotation.NO_TURN.rotatedDirection(CardinalDirection.NORTH)` → `CardinalDirection.NORTH`
  - `Rotation.QUARTER_TURN.rotatedDirection(CardinalDirection.EAST)` → `CardinalDirection.SOUTH`
  - `Rotation.HALF_TURN.rotatedDirection(CardinalDirection.SOUTH)` → `CardinalDirection.NORTH`
  - `Rotation.THREE_QUARTER_TURN.rotatedDirection(CardinalDirection.WEST)` → `CardinalDirection.SOUTH`

9.3 Classe `RotatedTile`

La classe `RotatedTile` correspond à une tuile à laquelle on a appliqué une rotation. Elle implémente l'interface `Tile` et contient les éléments suivants :

- un attribut `private final Tile tile` qui correspond à la tuile de départ.
- un attribut `private final Rotation rotation` qui correspond à la rotation appliquée à la tuile de départ.
- un constructeur `RotatedTile(Tile tile, Rotation rotation)` qui construit une tuile avec la tuile de départ et la rotation demandée.
- une méthode `Side side(CardinalDirection direction)` qui renvoie le côté de la tuile pour la direction demandée et qui correspond au côté de la tuile de départ dans la direction à laquelle on applique la `rotation`.

FIGURE 2 – Les rotations possibles d'une tuiles de Truchet

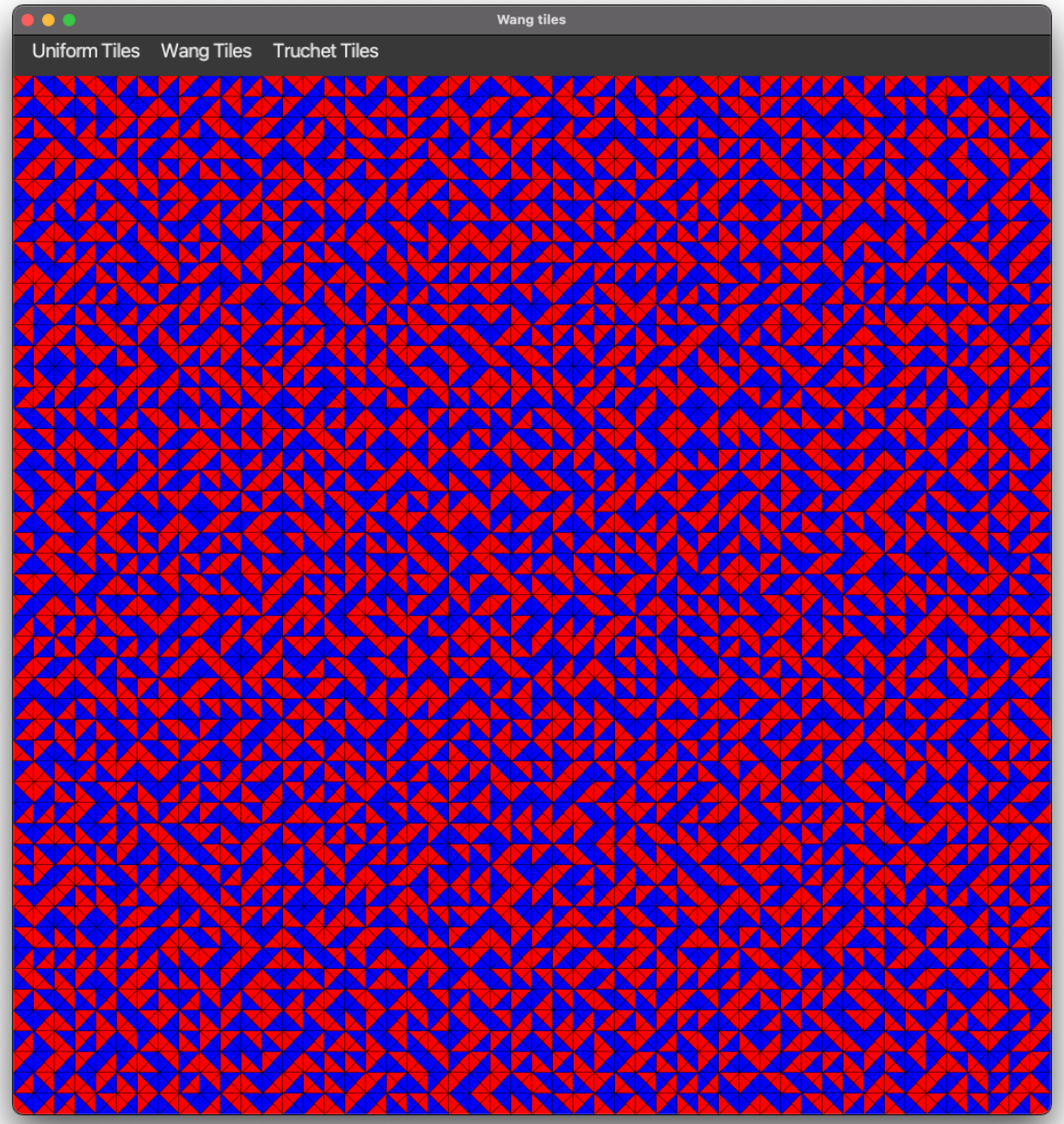
9.4 Classe `RandomRotatedTruchetTileGenerator`

La classe `RandomRotatedTruchetTileGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles de Truchet auxquelles on applique une rotation aléatoire.

- des attributs à définir par vous-même.
- un constructeur `RandomRotatedTruchetTileGenerator(Color color1, Color color2, Random randomGenerator)`.
- une méthode `Tile nextTile(Square square)` qui renvoie une instance de `RotatedTile` qui correspond à une tuile de Truchet obtenue à partir des couleurs `color1` et `color2` à laquelle on a appliqué une rotation aléatoire. La tuile devra donc être une des quatre tuiles ci-dessous :

9.5 Test du code `RandomRotatedTruchetTileGenerator`

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateRandomConstrainedWangTile()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Truchet tiles` → `Random`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :



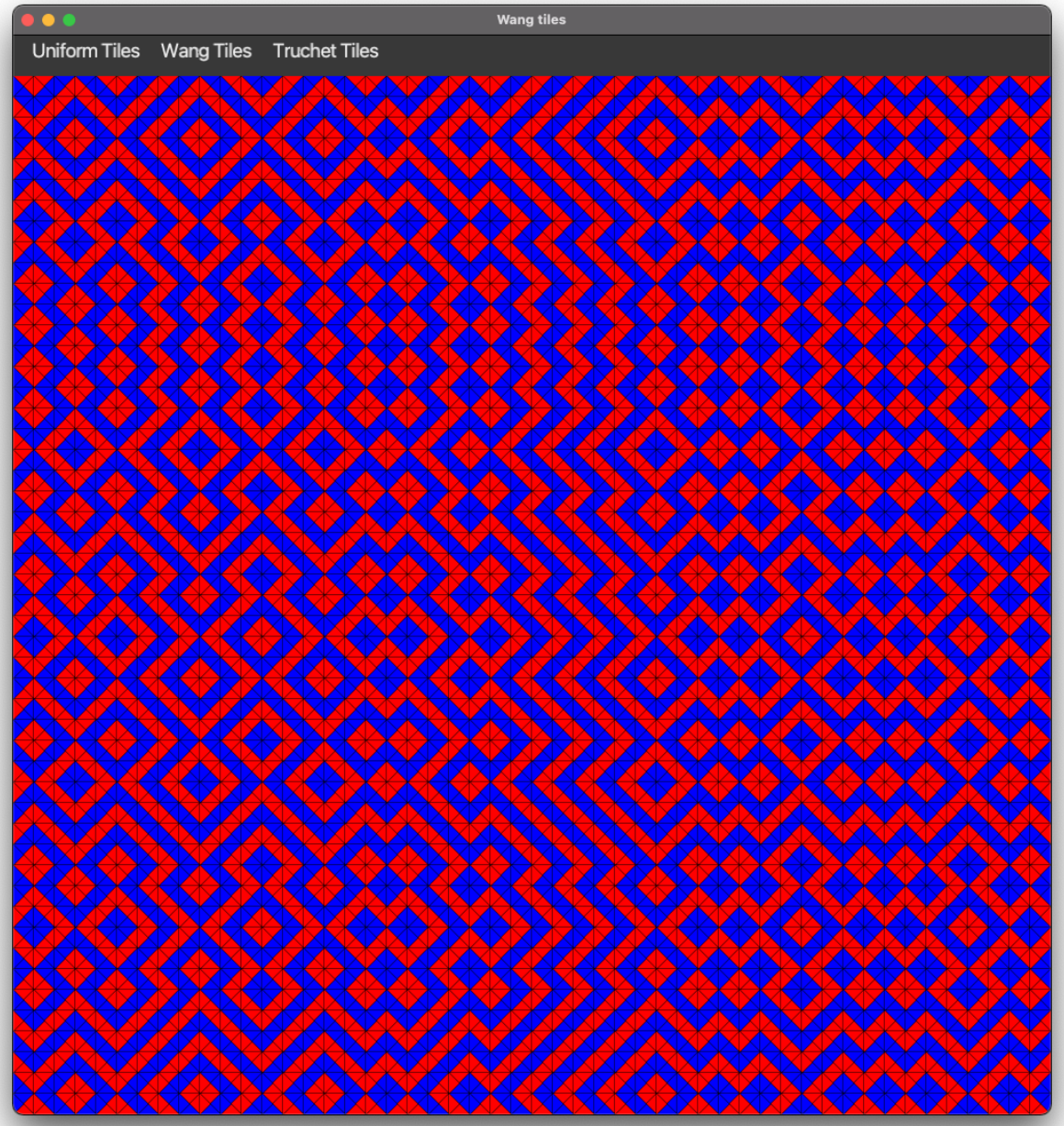
9.6 Classe `ConstrainedRotatedTruchetTileGenerator`

La classe `ConstrainedRotatedTruchetTileGenerator` correspond à un générateur (implémentant `TileGenerator`) générant des tuiles de Truchet auxquelles on a appliqué une rotation et dont les côtés sont compatibles avec les tuiles voisines.

- des attributs à définir par vous-même.
- un constructeur `ConstrainedRotatedTruchetTileGenerator(Color color1, Color color2, Random randomGenerator)`.
- une méthode `Tile nextTile(Square square)` qui renvoie une instance de `RotatedTile` qui correspond à une tuile de Truchet obtenue à partir des couleurs `color1` et `color2` à laquelle on a appliqué une rotation aléatoire et qui est compatible avec les tuiles voisines de la case `square`.

9.7 Test du code `ConstrainedRotatedTruchetTileGenerator`

Pour tester votre code, vous devez décommenter le code de la méthode `public void updateConstrainedTruchetTile()` dans la classe `GridController` (vous pouvez supprimer le commentaire `TODO` une fois que cela est fait). En passant par le menu `Truchet tiles` → `Constrained`, vous devriez obtenir un affichage ressemblant à l'affichage ci-dessous (les couleurs pouvant différer puis qu'elles dépendent de vos noms) :



10 Tâche 7 : fonctionnalités complémentaires

10.1 Classes de test

Pour cette sous-tâche, il vous faudra ajouter des tests pour toutes les classes que vous avez codés. Nous vous conseillons de les écrire en même temps que les classes dans le répertoire `main`.

10.2 Implémentation alternative des interfaces

Le but de cette tâche est de proposer des implémentations alternatives aux classes qu'on vous a demandées :

— Une classe `MapSquare` qui utilise un attribut de type Map<K,V> instancié à partir de la classe HashMap<K,V> pour stocker les voisins de la case. Les clés de la `Map` sont les directions et les valeurs sont les cases voisines.

— Une classe `ListGrid` qui utilisent un attribut de type List instancié à partir de la classe ArrayList. L'idée est de construire une liste de liste de cases pour stocker les cases de la grilles au lieu d'utiliser une matrice.

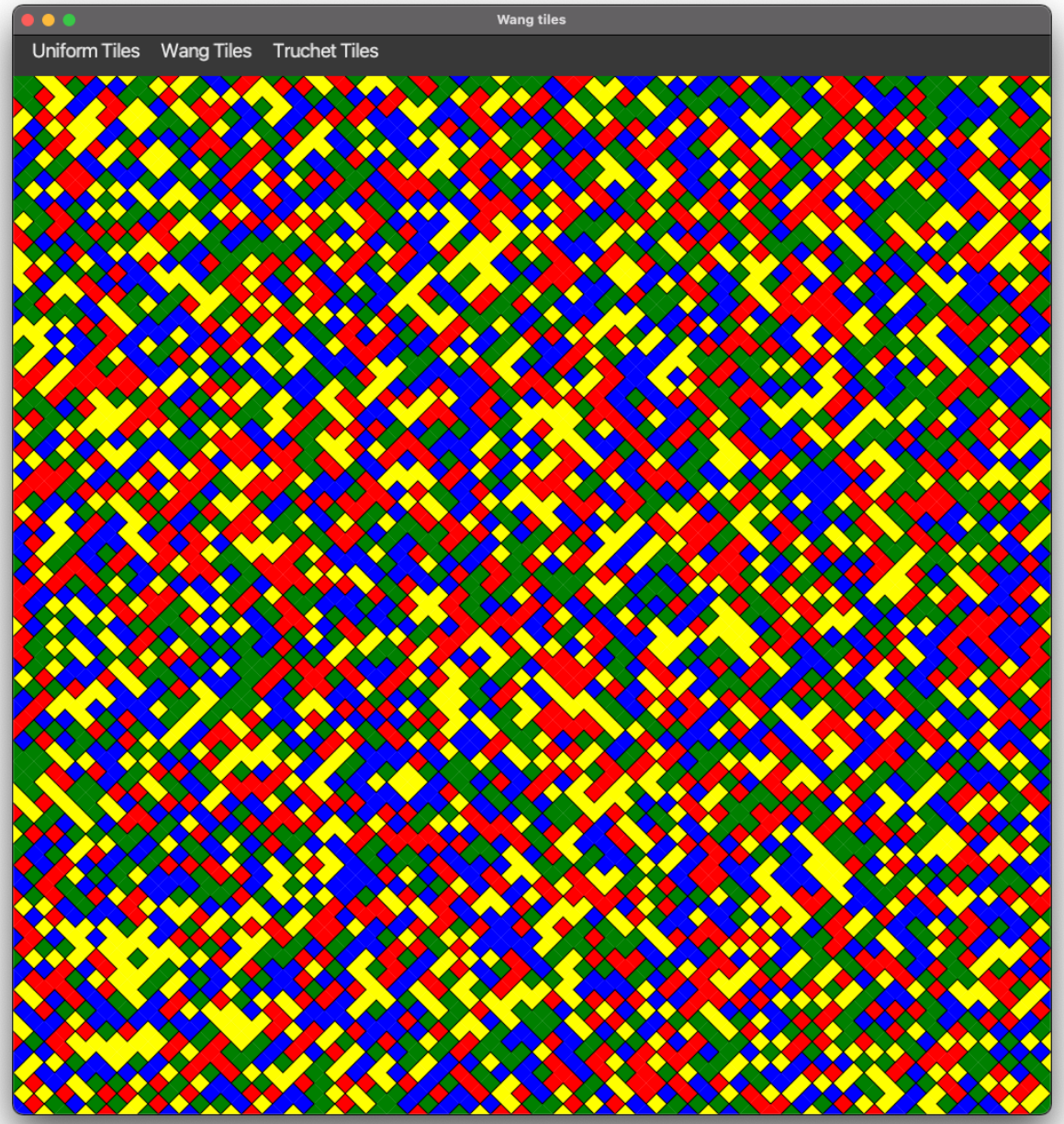
11 Tâche 8 : fonctionnalités optionelles

Ces tâches sont plus difficiles que les tâches précédentes et sont donc optionelles. Elle demandent de modifier l'affichage graphique (classe dans le répertoire `view`).

11.1 Changement du dessin

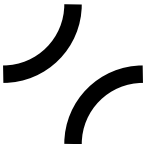
Changer la classe `GridCanvas` pour que les traits noirs séparant les triangles ne soit pas dessinés lorsque les deux triangles adjacents ont la même couleur.

Vous devrez obtenir un résultat similaire au résultat suivant pour l'affichage de tuiles de Wang avec contraintes :



11.2 Variantes tuiles de Truchet

Changer le code (notamment le code de la classe `GridCanvas`) pour définir une variante des tuiles de Truchet qui utilise des quarts de cercles (cf image ci-dessous).



Le but est d'obtenir des pavages ressemblant au pavage ci-dessous :