Solution de l'exercice 23

1. Définir le destructeur de la classe `Carte` ainsi que le constructeur de recopie et l'opérateur d'affectation n'est pas nécessaire car la classe n'a que des attributs primitifs : les opérations générées par défaut par le compilateur conviennent. On peut (mais ce n'est pas obligatoire, seulement conseillé) le faire apparaître explicitement en utilisant le mot clé **default**.

```
namespace Set {
 //...
 class Carte {
 public:
  Carte(Couleur c, Nombre v, Forme f, Remplissage r) :couleur(c), nombre(v),
      forme(f), remplissage(r) {}
  Couleur getCouleur() const { return couleur; }
  Nombre getNombre() const { return nombre; }
  Forme getForme() const { return forme; }
  Remplissage getRemplissage() const { return remplissage; }
  ~Carte() = default; // optionnel
  Carte(const Carte& c) = default; // optionnel
  Carte& operator=(const Carte& c) = default; // optionnel
 private:
  Couleur couleur;
  Nombre nombre;
  Forme forme;
  Remplissage remplissage;
 };

 ostream& operator<<(ostream& f, const Carte& c);
 //...
}
```

set.h

```
namespace Set {
 //...
 ostream& operator<<(ostream& f, const Carte& c) {
  f << "(" << c.getNombre() << "," << c.getCouleur()<<"," <<c.getForme()<<","
      << c.getRemplissage() << ")";
  return f;
 }
 //...
}
```

set.cpp

2. La classe `Carte` ne disposant pas de constructeur sans argument, il n'est pas possible de définir un tableau d'objets `Carte` ou d'utiliser l'opérateur **new**`[]` pour allouer dynamiquement un tableau d'objets `Carte` sans fournir d'initialisateur. En effet, ces objets `Carte` doivent être initialisés en faisant appel à l'unique constructeur disponible.

   Remarque : Notons cependant qu'à un stade avancé de maîtrise du C++, cette opération resterait néanmoins possible en utilisant l'autre opérateur **new**`[]()` de bas niveau, ou encore, un objet `allocator` qui permet la séparation de l'allocation mémoire de la construction d'un objet (et la destruction de la désallocation).

   Notons aussi qu'une telle opération est possible en fournissant des initialisateurs. Cependant, lorsque le nombre d'initialisateurs requis n'est pas connu, cette opération n'est pas possible.
   Il est tout à fait possible de créer des tableaux de pointeurs d'objets `Carte`. En effet, un pointeur n'a pas besoin d'être *initialisé* :

```
Carte* tab[10]; // tableau de 10 pointeurs
Carte** tabdyn=new Carte*[10]; /* tableau alloué dynamiquement de 10 pointeurs
    */
```

La cellule d'un tel tableau peut évidemment contenir l'adresse d'un objet `Carte` qui aura été créé (dynamiquement ou non) par la suite.

3. L'association qui lie les classes `Carte` et `Jeu` est une composition de `Jeu` vers `Carte` : la classe `Carte` est composée par la classe `Jeu` ; la classe `Jeu` compose la classe `Carte`. En effet, c'est l'objet `Jeu` qui va gérer le cycle de vie des objets `Carte`. Cette composition à une multiplicité de 81 du côté de `Carte`.
4. Une fois la carte créée (dynamiquement), il ne sera plus possible de la modifier par erreur.
5. Définir le destructeur de la classe `Jeu` est nécessaire car un objet `Jeu` doit gérer le cycle de vie des objets `Carte` qu'il compose. Ainsi, avant la libération mémoire d'un objet `Jeu`, il faut désallouer tous les objets `Carte` pointés par les pointeurs contenus dans le tableau `cartes`. On remarquera que la tableau `cartes` lui-même est un tableau automatique : il n'y a pas à libérer ce tableau (cela est fait automatiquement).
   Pour interdire la duplication on peut :
   — Soit mettre le constructeur de recopie et l'opérateur d'affectation dans la partie privée de la classe. Le compilateur générera une erreur d'accès dans le cas où l'on tente une utilisation de ces opérations en dehors des méthodes de la classe `Jeu`. Ces opérations peuvent ne pas être définies. L'éditeur de liens générera une erreur si on tente d'utiliser ces opérations dans les méthodes de la classe `Jeu`. C'est l'unique méthode possible en C++98.
   — Soit utiliser le mot clé **`delete`** après la déclaration du constructeur de recopie et de l'opérateur d'affectation. Le compilateur générera une erreur dans le cas où l'on tente une utilisation de ces opérations quel que soit le contexte. C'est la méthode conseillée depuis C++11.

```
namespace Set {
 //...
 class Jeu {
 public:
  Jeu();
  ~Jeu();
  Jeu(const Jeu& j) = delete;
  Jeu& operator=(const Jeu& j) = delete;
  size_t getNbCartes() const { return 81; }
  const Carte& getCarte(size_t i) const { if (i >= 81) throw SetException("
      Carte invalide");  return *cartes[i]; }
 private:
  const Carte* cartes[81];
 };
 //...
}
```

set.h

```
namespace Set {
 //...
 Jeu::Jeu() {
  size_t i = 0;
  for (auto c : Couleurs)
   for (auto v : Nombres)
    for (auto f : Formes)
     for (auto r : Remplissages) cartes[i++] = new Carte(c, v, f, r);
 }

 Jeu::~Jeu() {
  for (size_t i = 0; i < getNbCartes(); i++) delete cartes[i];
 }
 //...
}
```

set.cpp

6. L'association qui lie les classes `Carte` et `Pioche` est une agrégation de `Pioche` vers `Carte` : la classe `Pioche` agrège la classe `Carte` avec une valeur de multiplicité * du côté `Carte` et a priori une valeur de

multiplicité de 1 du côté `Pioche` (si on considère qu'une carte ne peut être que dans une seule pioche à la fois). En effet, bien qu'un objet `Pioche` est composé d'objets `Carte`, il n'est pas responsable de leur cycle de vie.

7. Puisque l'unique constructeur de `Pioche` prend une seule valeur de type **`const`** `jeu&`, il est intéressant d'utiliser **`explicit`** devant le constructeur de la classe pour éviter les coversions accidentelles de `Jeu` en `Pioche`. Le destructeur de la classe `Pioche` est nécessaire afin de libérer le tableau de pointeurs qui a été alloué dynamiquement. On remarquera que les objets `Carte` pointés ne sont pas libérés.

```
namespace Set {
 //...
 class Pioche {
  // désigne un paquet de cartes on l'on ne peut que piocher : prendre une
     carte au hasard
 public:
  explicit Pioche(const Jeu& j); // construction d'une pioche à partir du jeu
  bool estVide() const { return nb == 0; }
  size_t getNbCartes() const { return nb; }
  const Carte& piocher();
  ~Pioche();
  Pioche(const Pioche& p) = delete;
  Pioche& operator=(const Pioche& p) = delete;
 private:
  const Carte** cartes = nullptr;
  size_t nb = 0;
 };
 //...
}
```

set.h

```
namespace Set {
 //...
 Pioche::Pioche(const Jeu& j) :cartes(new const Carte*[j.getNbCartes()]), nb(j
    .getNbCartes()) {
  for (size_t i = 0; i < nb; i++) cartes[i] = &j.getCarte(i);
 }

 const Carte& Pioche::piocher() {
  if (nb == 0) throw SetException("Pioche vide");
  size_t x = rand() % nb; // on tire une position entre 0 et nb-1
  const Carte* c = cartes[x]; // on retient l'adresse
  for (size_t i = x + 1; i < nb; i++) cartes[i - 1] = cartes[i]; // on décale
     toutes les cartes aux rangs suivants
  nb--;
  return *c;
 }

 Pioche::~Pioche() {
  delete[] cartes;
 }
 //...
}
```

set.cpp

8. L'association qui lie les classes `Carte` et `Plateau` est une agrégation de `Plateau` vers `Carte` : la classe `Plateau` agrège la classe `Carte` avec une valeur de multiplicité 12..* du côté `Carte` et a priori une valeur de multiplicité de 1 du côté `Plateau` (si on considère qu'une carte ne peut être que dans une seule pioche à la fois). En effet, bien qu'un objet `Plateau` est composé d'objets `Carte`, il n'est pas responsable de leur cycle de vie.
9. Le destructeur de la classe `Plateau` est nécessaire pour désallouer le tableau `cartes` alloué dynamiquement.

```
namespace Set {
```

```
 //...
 class Plateau {
 public:
  Plateau() = default;
  ~Plateau() { delete[] cartes; }
  size_t getNbCartes() const { return nb; }
  void ajouter(const Carte& c);
  void retirer(const Carte& c);
  void print(ostream& f = cout) const;
  Plateau(const Plateau& p);
  Plateau& operator=(const Plateau& p);
 private:
  const Carte** cartes = nullptr;
  size_t nbMax = 0;
  size_t nb = 0;
 };

 ostream& operator<<(ostream& f, const Plateau& m);
 //...
}
```

set.h

```
namespace Set {
 //...
 void Plateau::ajouter(const Carte& c) {
  if (nb == nbMax) {
   const Carte** newtab = new const Carte*[(nbMax + 1) * 2];
   for (size_t i = 0; i < nb; i++) newtab[i] = cartes[i];
   auto old = cartes;
   cartes = newtab;
   nbMax = (nbMax + 1) * 2;
   delete[] old;
  }
  cartes[nb++] = &c;
 }

 void Plateau::retirer(const Carte& c) {
  size_t i = 0;
  while (i < nb && cartes[i] != &c) i++;
  if (i == nb) throw SetException("Carte inexistante");
  i++;
  while (i < nb) {
   cartes[i - 1] = cartes[i]; i++;
  }
  nb--;
 }

 void Plateau::print(ostream& f) const {
  for (size_t i = 0; i < nb; i++) {
   if (i % 4 == 0) f << "\n";
   f << *cartes[i] << " ";
  }
  f << "\n";
 }

 ostream& operator<<(ostream& f, const Plateau& m) {
  m.print(f);
  return f;
 }

 Plateau::Plateau(const Plateau& p):cartes(new const Carte*[p.nb]),nb(p.nb),
```

```
    nbMax(p.nb) {
  for (size_t i = 0; i < nb; i++) cartes[i] = p.cartes[i];
 }

 Plateau& Plateau::operator=(const Plateau& p) {
  if (this != &p) {
   if (p.nb > nbMax) {
    const Carte** newtab = new const Carte*[p.nb];
    for (size_t i = 0; i < nb; i++) newtab[i] = p.cartes[i];
    auto old = cartes;
    cartes = newtab;
    nb=nbMax = p.nb;
    delete[] old;
   }
   else {
    for (size_t i = 0; i < nb; i++) cartes[i] = p.cartes[i];
    nb = p.nb;
   }
  }
  return *this;
 }
 //...
}
```

set.cpp

10. L'association qui lie les classes `Carte` et `Combinaison` est une agrégation de `Combinaison` vers `Carte` : la classe `Combinaison` agrège la classe `Carte` avec une valeur de multiplicité 3 du côté `Carte` et a priori une valeur de multiplicité de * du côté `Combinaison` (si on considère qu'une carte ne peut être impliquée dans plusieurs combinaisons). En effet, bien qu'un objet `Combinaison` est composé d'objets `Carte`, il n'est pas responsable de leur cycle de vie.
11. Les définitions du destructeur, le constructeur de recopie et l'opérateur d'affectation de la classe `Combinaison` ne sont pas nécessaires puisque les opérations générées par défaut par le compilateur conviennent.

```
namespace Set {
 //...
 class Combinaison {
 public:
  Combinaison(const Carte& u, const Carte& d, const Carte& t) :c1(&u), c2(&d),
      c3(&t) {}
  bool estUnSET() const;
  const Carte& getCarte1() const { return *c1; }
  const Carte& getCarte2() const { return *c1; }
  const Carte& getCarte3() const { return *c1; }
  ~Combinaison() = default;
  Combinaison(const Combinaison& c) = default;
  Combinaison& operator=(const Combinaison& c) = default;
 private:
  const Carte* c1;
  const Carte* c2;
  const Carte* c3;
 };

 ostream& operator<<(ostream& f, const Combinaison& c);

 //...
}
```

set.h

```
namespace Set {
 //...
```

```
 bool Combinaison::estUnSET() const {
  bool c = (c1->getCouleur() == c2->getCouleur() && c1->getCouleur() == c3->
      getCouleur()) || (c1->getCouleur() != c2->getCouleur() && c1->getCouleur
      () != c3->getCouleur() && c2->getCouleur() != c3->getCouleur());
  bool n = (c1->getNombre() == c2->getNombre() && c1->getNombre() == c3->
      getNombre()) || (c1->getNombre() != c2->getNombre() && c1->getNombre() !=
       c3->getNombre() && c2->getNombre() != c3->getNombre());
  bool f = (c1->getForme() == c2->getForme() && c1->getForme() == c3->getForme
      ()) || (c1->getForme() != c2->getForme() && c1->getForme() != c3->
      getForme() && c2->getForme() != c3->getForme());
  bool r = (c1->getRemplissage() == c2->getRemplissage() && c1->getRemplissage
      () == c3->getRemplissage()) || (c1->getRemplissage() != c2->
      getRemplissage() && c1->getRemplissage() != c3->getRemplissage() && c2->
      getRemplissage() != c3->getRemplissage());
  return c && n && f && r;
 }

 ostream& operator<<(ostream& f, const Combinaison& c) {
  f << "[ " << c.getCarte1() << " ; " << c.getCarte2() << " ; " << c.getCarte2
      () << " ]";
  return f;
 }
 //...
}
```

set.cpp

12. La classe `Controleur` compose à la fois la classe `Jeu` et la classe `Pioche` avec une multiplicité de 1 à chaque fois.
13. Il est nécessaire de définir le destructeur de la classe `Controleur` pour dasllouer l'objet `Pioche`.

```
namespace Set {
 //...
 class Controleur {
 public:
  Controleur();
  void distribuer();
  ~Controleur() { delete pioche; }
  const Plateau& getPlateau() const { return plateau; }
 private:
  Jeu jeu;
  Pioche* pioche=nullptr;
  Plateau plateau;
 };
 //...
}
```

set.h

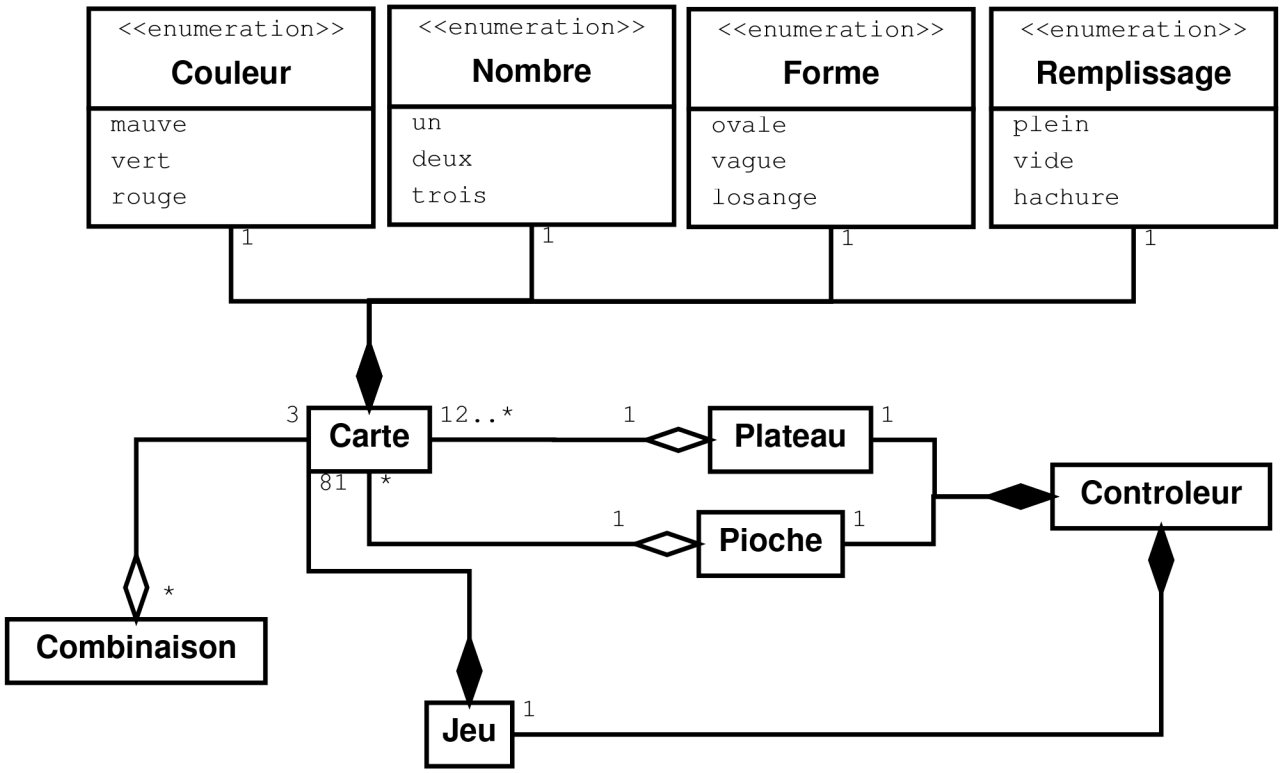
```
namespace Set {
 //...
 Controleur::Controleur() {
  pioche = new Pioche(jeu);
 }

 void Controleur::distribuer() {
  if (plateau.getNbCartes() < 12)
   while (!pioche->estVide() && plateau.getNbCartes() < 12) plateau.ajouter(
       pioche->piocher());
  else
   if (!pioche->estVide()) plateau.ajouter(pioche->piocher());
 }
```

```
  //...
}
```

set.cpp

14. Voici un diagramme UML représentant les associations entre classes. A titre d'exemple, on y fait aussi apparaître les énumérations afin de montrer comment elles peuvent être représentées.



Exercice 24 - Problèmes de conception

Dans l'application, les objets `Carte` sont gérés par un module appelé `Jeu` qui est responsable de leur création (et destruction.

1. Expliciter des intérêts de mettre en place le Design Pattern *Singleton* pour la classe `Jeu`. Transformer la classe `Jeu` en singleton. On étudiera les différentes possibilités.
2. Modifier les classes qui utilisent `Jeu` en profitant de l'accès central de l'instance `Jeu`. Proposer une nouvelle version du diagramme de classe pour tenir compte de cela.
3. Faire en sorte que seule l'instance de la classe `Jeu` puisse créer des objets `Cartes`.

La méthode `Jeu::getCarte()` permet d'accéder une carte à partir d'un numéro qui est arbitraire du point de vue de l'utilisateur et qui expose la structure de données utilisée.

4. Afin de pouvoir parcourir séquentiellement les cartes du jeu, appliquer le design pattern *Iterator* à cette classe en déduisant son implémentation du code suivant :

```
void afficherCartes(){
  for(Jeu::Iterator it= Jeu::getInstance().getIterator();!it.isDone();it.next
      ())
   std::cout<<it.currentItem()<<"\n";
}
```

   Mettre la méthode `Jeu::getCarte()` dans la partie privée en rendant son accès exlusif à la classe `Jeu::Iterator`. Modifier les éléments du code impacté par la modification de l'interface de la classe `Jeu`.

En fait, plusieurs types d'itérateur peuvent être proposé par une classe afin d'offrir des services particuliers.

5. Compléter la classe `Jeu` afin de pouvoir parcourir séquentiellement uniquement les cartes du jeu ayant une forme donnée :

```
void afficherCartes(Forme f){
  for(Jeu::FormeIterator it= Jeu::getInstance().getIterator(f);!it.isDone();it
      .next())
   std::cout<<it.currentItem()<<"\n";
}
```

6. Appliqer le design pattern iterator pour parcourir les cartes d'un plateau en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL) :

```
void afficherCartes(const Plateau& p){
 for(Plateau::const_iterator it=p.begin();it!=p.end();++it)
   std::cout<<*it<<"\n";
}
```

Remarque : Puisque les cartes (du jeu, on d'un plateau) sont non modifiables, on remarquera que tous les itérateurs implémentés dans cet exercice propose un accès uniquement en lecture. En pratique, dans une interface similaire au conteneurs standards du C++, on a aussi la classe `iterator` (en plus de `const_iterator`. L'opérateur d'indirection * renvoie alors une référence non-c**const** de l'objet désigné par l'itérateur, ce qui permet de modifier éventuellement cet objet. Quand les deux itérateurs (**const** et non-**const**) sont implémentés, les versions non-**const** de `begin()` et `end()` renvoie un `iterator`, alors que les versions **const** de `begin()` et `end()` renvoie un `const_iterator`. Afin de pouvoir obtenir un `const_iterator` à partir d'un objet non constant, on implémente aussi souvent les méthodes `cbegin()` et `cend()`, disponibles en version **const** uniquement (elles sont donc appelables à partir d'un objet **const** ou non-**const**) et qui renvoient un `const_iterator`.

7. **(question d'approfondissement à faire chez soi ou en TD s'il reste du temps)** Finaliser le jeu de manière à pouvoir jouer au Set !.

Solution de l'exercice 24

1. Ici l'intérêt est multiple. Tout d'abord au niveau mémoire, un objet `Jeu` est assez lourd. Il peut donc être utile de s'assurer de l'unicité de l'instance de cette classe d'autant qu'il n'y a pas vraiment d'intérêt d'avoir plusieurs instances de cette classe en mémoire. La mise en place du singleton permet aussi d'avoir un accès simplifié à l'instance `Jeu` depuis n'importe quel point du code.
   Pour implémenter le singleton, il y a plusieurs possibilités. Notons que le constructeur et le destructeur de la classe doivent être privés. Le constrcuetru de recopie et l'opérateur d'affectation restent désactivés. Tous les accès de création, duplication et destruction sont maintenant bloqués. C'est la méthode statique `getInstance()` qui permettra de créer l'unique instance et d'en fournir un accès.

```
namespace Set {
 //...
 class Jeu {
 public:
  static Jeu& getInstance();
  //...
 private:
  Jeu();
  ~Jeu();
  Jeu(const Jeu& j) = delete;
  Jeu& operator=(const Jeu& j) = delete;
  //...
 };
 //...
}
```

set.h

   Dans une première version, on se base sur le fait qu'un objet local **static** est créé lors de la première utilisation de la fonction dans lequel il se trouve (ici `getInstance()`). Ensuite, pendant toute la durée de la vie du programme, c'est le même objet qui est réutilisé (il n'est pas recréé). Ainsi, la méthode `getInstance()` renvoie toujours le même objet (le seul qui sera créé)... Notons qu'il n'est pas possible ici de libérer l'unique instance et d'en recréer une autre. Cependant, cette version a l'avantage de décharger le client de la classe de la responsabilité de la destruction de cette instance (il n'y a pas de la méthode `libererInstance()`).

```
namespace Set {
 //...
 class Jeu {
 public:
  static Jeu& getInstance() {
     static Jeu jeu; // unique instance
     return jeu; // c'est toujours le même objet retourné
  }
  //...
 };
 //...
}
```

set.h

   Dans une deuxième version plus souple au niveau gestion mémoire, on propose aussi une méthode `libererJeu()` qui permet de libérer l'unique instance (qui peut être ensuite recréée si nécessaire). L'adresse de cette unique instance est stockée dans l'attribut statique `instance`. Notons que cet attribut statique doit être défini et initialisé (ici à `nullptr`) dans le fichier source. Si ce pointeur est nul, la méthode `getInstance()` crée dynamiquement l'instance avant de la renvoyer. Sinon elle ne fait que renvoyer l'instance. La méthode `libererInstance()` désalloue l'objet et remet le pointeur `instance` à l avaleur `nullptr`.

```
namespace Set {
 //...
 class Jeu {
 public:
  static Jeu& getInstance() { if (instance == nullptr) instance = new Jeu;
     return *instance; }
  static void libererInstance() { delete instance; instance = nullptr; }
```

```
  //...
 private:
  static Jeu* instance;

 };
 //...
}
```

set.h

```
namespace Set {
 //...
  Jeu* Jeu::instance=nullptr;
 //...
}
```

set.cpp

Une troisième version possible consiste à encapsuler un pointeur d'objet `Jeu` dans un objet statique de type `Handler` dont le destructeur assure la libération ultime de l'objet `Jeu` pointé. On obtient ainsi à la fois la possibilité de créer et libérer l'objet `Jeu` quand l'utilisateur le souhaite tout en garantissant sa destruction correcte en fin d'exécution du programme. En effet, puisque l'objet `handler` est statique, sa libération mémoire (et l'appel de son destructeur) est assuré à la terminaison du programme.

```
namespace Set {
 //...
 class Jeu {
 public:
  static Jeu& getInstance() { if (handler.instance == nullptr) handler.
      instance = new Jeu; return *handler.instance; }
  static void libererInstance() { delete handler.instance; handler.instance =
      nullptr; }
  //...
 private:
  struct Handler {
   Jeu* instance;
   Handler() :instance(nullptr) {} // pointeur initialisé à la création de l'
       objet Handler
   ~Handler() { delete instance; } // l'instance est libérée au moment de la
       destruction (automatique) du Handler
  };
  static Handler handler;
  //...
 };

 //...
}
```

set.h

```
namespace Set {
 //...
 Jeu::Handler Jeu::handler = Handler();
 //...
}
```

set.cpp

2. Voici les modification à effectuer :
   — Le constructeur de `Pioche` est désormais sans argument :

```
namespace Set {
 //...
 class Pioche {
```

```
 public:
  Pioche();
  //...
 };

 //...
}
```

set.h

```
namespace Set {
 //...
 Pioche::Pioche() :
  cartes(new const Carte*[Jeu::getInstance().getNbCartes()]), nb(Jeu::
      getInstance().getNbCartes()) {
  for (size_t i = 0; i < nb; i++) cartes[i] = &Jeu::getInstance().getCarte(i
      );
 }
 //...
}
```

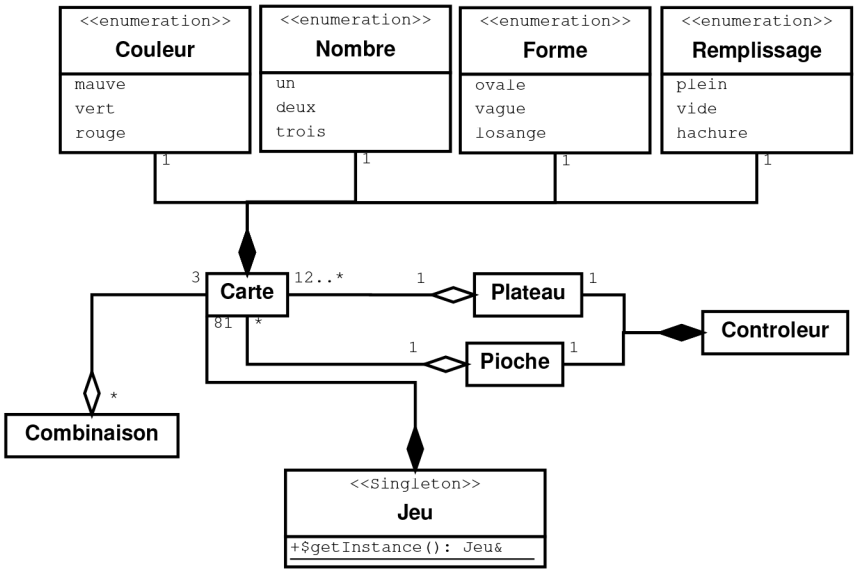
set.cpp

— La classe `Controleur` ne compose plus d'instance de `Jeu` : son attribut `jeu` est éliminé et son constructeur est modifié :

```
namespace Set {
 //...
 Controleur::Controleur() {
  pioche = new Pioche;
 }
 //...
}
```

set.cpp

Voici un nouveau diagramme de classes adapté aux changements :



Notons que les membres statiques sont marqués avec un `$` ou soulignés sur un diagramme UML.

3. Pour faire en sorte que seule l'instance de la classe `Jeu` puisse créer des objets `Cartes`, il sufit de mettre le constructeur de `Carte` dans la partie privée et de déclarer la classe `Jeu` comme amie de la classe :

```
namespace Set {
 //...
 class Carte {
 public:
```

```
  //...
 private:
  Carte(Couleur c, Nombre v, Forme f, Remplissage r) :couleur(c), nombre(v),
      forme(f), remplissage(r) {}
  friend class Jeu;
  //...
 };
 //...
}
```

set.h

Commencer par faire un rappel de l'intérêt du design pattern iterator. On étudie ce design pattern avec des interfaces différentes pour bien comprendre qu'il y a de multiples façons d'implémenter ce design pattern. Il ne faudrait pas focaliser uniquement sur la façon de faire classique de la STL, tant on recontre de multiples interfaces selon les bibliothèques.

4. La première façon de faire est de copier l'interface proposée dans le poly dans la présentation de ce design pattern. Ce design pattern est appliqué à la classe `Jeu`.
   - Le code proposé dans l'exercice montre que la classe `Jeu` encapsule un type publique qui s'appelle `Iterator` (ce sera une classe définie à l'intérieur de la classe `Jeu`) et qu'elle propose une méthode `getIterator` permettant d'obtenir un itérateur pointant sur le premier élément de la séquence à parcourir.
   - La classe `Jeu::Iterator` doit proposer la méthode `currentItem()` renvoyant une référence sur l'objet `Carte` désigné par l'itérateur. Pour cela, l'attribut `i` le rang courant de l'objet `Carte` désigné par l'itérateur. On profite de l'accès central proposé par le singleton `Jeu` pour obtenir la carte concernée. On remarquera que seule la classe `Iterator` peut maintenant utiliser la méthode `getCarte()` qui est maintenant dans la partie privée de `Jeu`.
   - La classe `Jeu::Iterator` doit proposer la méthode `next` qui permet de faire en sorte que l'itérateur se déplace sur l'élément suivant dans la séquence.
   - La classe `Jeu::Iterator` devra propose la méthode `isDone()` qui renverra la valeur **`true`** si toutes les cartes ont été parcourues.
   - La classe `Jeu::Iterator` doit proposer un constructeur qui permet d'initialiser correctement l'attribut `i` . La méthode `Jeu::getIterator()` utilise ce constructeur. Remarquons que ce constructeur ne devrait être utilisé que par la méthode `Jeu::getIterator()`. C'est pourquoi, il a été placé dans la partie **`private`** de la classe `Iterator`. La classe `Jeu` a accès à ce constructeur en tant qu'amie de la classe.
   - On remarque que les méthodes `isDone()` et `currentItem()` ne modifient pas les attributs de la classe et devraient donc être **`const`**.
   - La méthode `Pioche::Pioche()` doit être modifier en conséquence car `Jeu::getCarte()` lui est maintenant inacessible.

```
namespace Set {
 //...
 class Jeu {
 public:
  //..
  class Iterator {
  public:
   void next() { if (isDone()) throw SetException("Iterateur en fin de
       sequence"); i++; }
   bool isDone() const { return i == Jeu::getInstance().getNbCartes();  }
   const Carte& currentItem() const {
     if (isDone()) throw SetException("Iterateur en fin de sequence");
     return Jeu::getInstance().getCarte(i);
   }
  private:
   size_t i = 0;
   friend class Jeu;
   Iterator() = default; // seule la classe Jeu peut construire un Iterator
  };
  Iterator getIterator() const { return Iterator(); }
 private:
  friend class Iterator;
```

```
  // getCarte() est maintenant réservé exclusivement à Iterator
  const Carte& getCarte(size_t i) const { if (i >= 81) throw SetException("
      Carte invalide");  return *cartes[i]; }
  //...
 };

 //...
}
```

set.h

```
namespace Set {
 //...
 Pioche::Pioche() :
  cartes(new const Carte*[Jeu::getInstance().getNbCartes()]), nb(Jeu::
      getInstance().getNbCartes()) {
  //for (size_t i = 0; i < nb; i++) cartes[i] = &Jeu::getInstance().getCarte(i
      );
  size_t i = 0;
  for (auto it = Jeu::getInstance().getIterator(); !it.isDone(); it.next(), i
      ++) cartes[i] = &it.currentItem();

 }
 //...
}
```

set.cpp

5. La classe `FormeIterator` est similaire à la classe `Iterator`. Les différences interviennent dans le constructeur et dans la méthode `next()` où il faut avancer dans la séquence jusqu'une carte avec la forme donnée soit trouvée.

```
namespace Set {
 //...
 class Jeu {
 public:
  //...
  class FormeIterator {
  public:
   void next() {
    if (isDone()) throw SetException("Iterateur en fin de sequence");
    i++;
    while (!isDone() && Jeu::getInstance().getCarte(i).getForme() != forme) i
        ++;

   }
   bool isDone() const { return i == Jeu::getInstance().getNbCartes(); }
   const Carte& currentItem() const { if (isDone()) throw SetException("
       Iterateur en fin de sequence"); return Jeu::getInstance().getCarte(i); }
  private:
   size_t i = 0;
   friend class Jeu;
   Forme forme;
   FormeIterator(Forme f) :forme(f) {
    while (!isDone() && (Jeu::getInstance().getCarte(i).getForme() != forme))
        {
     i++;
    }
   }
  };
  FormeIterator getIterator(Forme f) const { return FormeIterator(f); }
  friend class FormeIterator;
  //...
```

```
 };
 //...
}
```

set.h

6. Dans cette question, une deuxième implémantation mime la façon de faire de la STL. Le design pattern est appliqué à la classe `Plateau`.

```
namespace Set {
 //...
 class Plateau {
 public:
  //...
  class const_iterator {
  public:
   const_iterator & operator++() { current++;  return *this; }
   const Carte& operator*() const { return **current; }
   bool operator!=(const_iterator it) const { return current != it.current; }
  private:
   const_iterator(const Carte** c) :current(c) {}
   friend class Plateau;
   const Carte** current = nullptr;
  };
  const_iterator begin() const { return const_iterator(cartes); }
  const_iterator end() const { return const_iterator(cartes+nb); }
  //...
 };
 //...
}
```

set.h

— Cette fois-ci le types utilisé est `Plateau::const_iterator`.
— L'information permettant de savoir quelle est la carte pointée est stockée dans l'attribut `current`.
— Si l'interface diffère, le fonctionnement est assez similaire. L'opérateur `++` joue le rôle de la méthode `next()` alors que l'opérateur `*` joue celui de la méthode `currentItem()`.
— La méthode `begin()` joue le rôle de la méthode `getIterator()`. Remarquons par contre, que la méhode `end()` de la classe `Plateau` qui remplace la méthode `isDone()` (et qui était une méthode de la classe `Iterator` dans l'interface précédente). Cela nécessite la présence de l'opérateur **`operator`**`!=` dans la classe `const_iterator` qui permet de savoir si deux itérateurs pointent sur le même élément. Par contre la classe `iterator` n'a pas besoin de stocker l'information qui permet de savoir quand le parcours est terminé.