Architecture et développement logiciels

Yassamine Seladji

yassamine.seladji@gmail.com

24 septembre 2018

Idées reçus erronés

- Une idée général sur le logiciel

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne $\Longrightarrow$travail terminé.

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne $\Longrightarrow$travail terminé.
- La spécification change

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne $\Longrightarrow$travail terminé.
- La spécification change $\Longrightarrow$ facilité de gestion.

Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne $\Longrightarrow$travail terminé.
- La spécification change $\Longrightarrow$ facilité de gestion.
- Le projet prends du retards

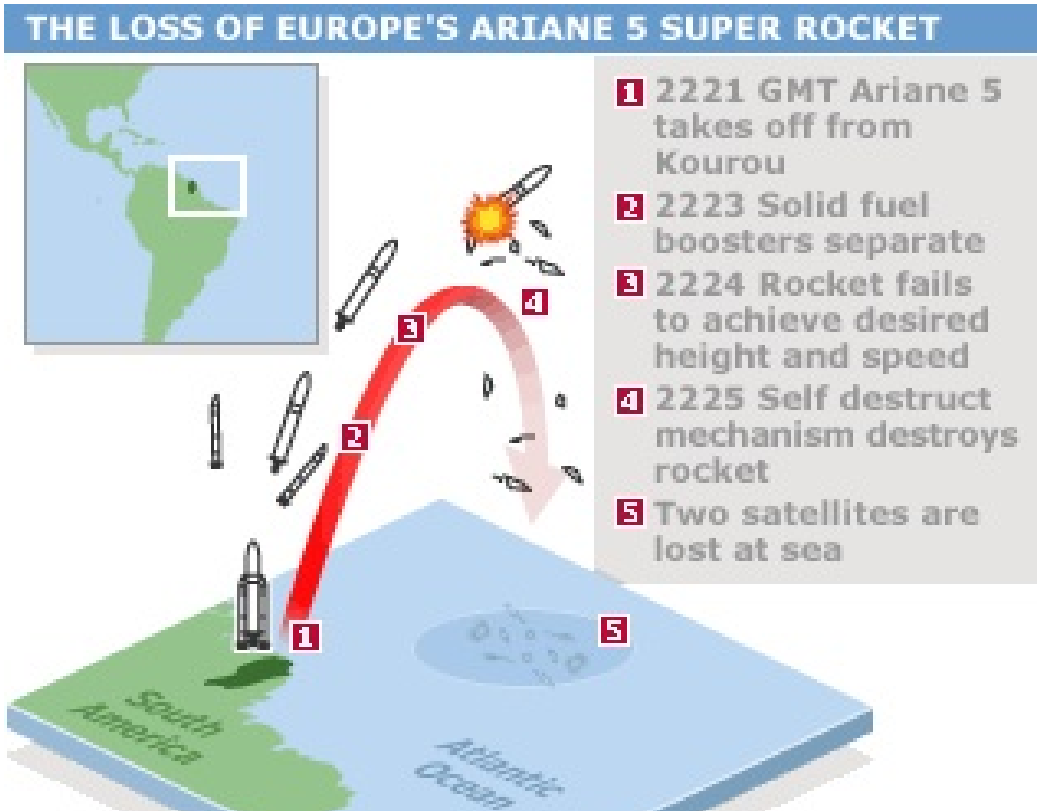
Idées reçus erronés

- Une idée général sur le logiciel $\Longrightarrow$ début de la programmation.
- Le logiciel fonctionne $\Longrightarrow$travail terminé.
- La spécification change $\Longrightarrow$ facilité de gestion.
- Le projet prends du retards $\Longrightarrow$ ajout de programmeurs.

Contexte

Un problème industriel

- le crash d'Ariane 5 : causé par un dépassement de capacité (arithmétique overflow).
  $\Longrightarrow$ 700 Million d'euro de perte.



Critère de qualité

- Validité : compatible avec le cahier de charge.

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.
- Performance : un bon compromis entre rapidité de réponse et taille de la mémoire utilisée.

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.
- Performance : un bon compromis entre rapidité de réponse et taille de la mémoire utilisée.
- Fiabilité : gestions des pannes et erreurs .

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.
- Performance : un bon compromis entre rapidité de réponse et taille de la mémoire utilisée.
- Fiabilité : gestions des pannes et erreurs .
- Sécurité : protection des accés et données.

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.
- Performance : un bon compromis entre rapidité de réponse et taille de la mémoire utilisée.
- Fiabilité : gestions des pannes et erreurs .
- Sécurité : protection des accés et données.
- Maintenabilité : Facilité de correction et d'ajout de nouvelles fonctionnalités.

Critère de qualité

- Validité : compatible avec le cahier de charge.
- Ergonomie : facile d'utilisation.
- Performance : un bon compromis entre rapidité de réponse et taille de la mémoire utilisée.
- Fiabilité : gestions des pannes et erreurs .
- Sécurité : protection des accés et données.
- Maintenabilité : Facilité de correction et d'ajout de nouvelles fonctionnalités.
- Portabilité : possibilité de changer d'environnement matériel ou logiciel.

Principe d'Ingénierie pour le Logiciel

- Rigueur : s'assurer de faire ce qui est demander

Principe d'Ingénierie pour le Logiciel

- Rigueur : s'assurer de faire ce qui est demander
- Abstraction : raisonner sur les concepts généraux et implémenter les solutions sur les cas particuliers.

Principe d'Ingénierie pour le Logiciel

- Rigueur : s'assurer de faire ce qui est demander
- Abstraction : raisonner sur les concepts généraux et implémenter les solutions sur les cas particuliers.
- Décomposition en sous problèmes : traiter chaque aspect séparément.

Principe d'Ingénierie pour le Logiciel

- Rigueur : s'assurer de faire ce qui est demander
- Abstraction : raisonner sur les concepts généraux et implémenter les solutions sur les cas particuliers.
- Décomposition en sous problèmes : traiter chaque aspect séparément.
- Modularité : partition du logiciel en modules interagissant, remplissant une fonction et ayant une interface cachant l'implémentation aux autres modules.

Principe d'Ingénierie pour le Logiciel

- Rigueur : s'assurer de faire ce qui est demander
- Abstraction : raisonner sur les concepts généraux et implémenter les solutions sur les cas particuliers.
- Décomposition en sous problèmes : traiter chaque aspect séparément.
- Modularité : partition du logiciel en modules interagissant, remplissant une fonction et ayant une interface cachant l'implémentation aux autres modules.
- Construction incrémentale : construction pas a pas, intégration progressive.

Principe d'Ingénierie pour le Logiciel

- Généricité : proposer des solutions plus générales pour pouvoir les réutiliser et les adapter à d'autres cas.

Principe d'Ingénierie pour le Logiciel

- Généricité : proposer des solutions plus générales pour pouvoir les réutiliser et les adapter à d'autres cas.
- Anticipation des évolutions : lier à la généricité et à la modularité, prévoir les ajouts et modifications de fonctionnalités dans le future.

Principe d'Ingénierie pour le Logiciel

- Généricité : proposer des solutions plus générales pour pouvoir les réutiliser et les adapter à d'autres cas.
- Anticipation des évolutions : lier à la généricité et à la modularité, prévoir les ajouts et modifications de fonctionnalités dans le future.
- Documentations : essentielle pour le suivi et l'évolution du projet.

Principe d'Ingénierie pour le Logiciel

- Généricité : proposer des solutions plus générales pour pouvoir les réutiliser et les adapter à d'autres cas.
- Anticipation des évolutions : lier à la généricité et à la modularité, prévoir les ajouts et modifications de fonctionnalités dans le future.
- Documentations : essentielle pour le suivi et l'évolution du projet.
- Standardisation/Normalisation : facilité de communication pour le développement et la maintenance.

Des Principes à la Pratique

Les principes deviennent pratique en utilisant des méthodes et outils adaptés à chaque étape du processus du développement d'un logiciel :

Des Principes à la Pratique

Les principes deviennent pratique en utilisant des méthodes et outils adaptés à chaque étape du processus du développement d'un logiciel :

- Analyse des besoins : Comprendre les besoins du client.
- Spécification : Définir le quoi faire.
- Conception : définir le comment faire.
- Programmation : réalisation de la solution conçue.
- Validation et vérification : assurer que la spécification est satisfaite et qu'elle correspond aux besoins du clients.
- Maintenance : assurer la mise à jour du logiciel.

Des Principes à la Pratique

Les principes deviennent pratique en utilisant des méthodes et outils adaptés à chaque étape du processus du développement d'un logiciel :

- Analyse des besoins : Comprendre les besoins du client.
- Spécification : Définir le quoi faire.
- **Conception : définir le comment faire.**
- **Programmation : réalisation de la solution conçu.**
- Validation et vérification : assurer que la spécification est satisfaite et qu'elle correspond aux besoins du clients.
- Maintenance : assurer la mise à jour du logiciel.

La conception Orienté Objet

Une bonne conception doit :

- Définir les objets pertinents, concevoir les classes correspondantes avec la bonne granularité.

La conception Orienté Objet

Une bonne conception doit :

- Définir les objets pertinents, concevoir les classes correspondantes avec la bonne granularité.
- Définir les relations hiérarchiques entre les différentes classes et interfaces.

La conception Orienté Objet

Une bonne conception doit :

- Définir les objets pertinents, concevoir les classes correspondantes avec la bonne granularité.
- Définir les relations hiérarchiques entre les différentes classes et interfaces.
- Répondre précisément au problème proposé toute en restant le plus général possible afin de s'adapter facilement à l'évolution des besoins (ajout/modification).

Les Designs Patterns

Pattern

Un patron décrit à la fois un problème qui se produit très fréquemment dans l'environnement et l'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois sans jamais l'adapter deux fois de la même manière.

C.Alexander

Les Designs Patterns

Pattern

Un patron décrit à la fois un problème qui se produit très fréquemment dans l'environnement et l'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois sans jamais l'adapter deux fois de la même manière.

C.Alexander

Problème + Environnement $\Longrightarrow$ Solution (patron adéquat).

Les Designs Patterns

- Les designs patterns décrivent une solution simple et élégante à des problèmes spécifiques en conception orienté objet.

Les Designs Patterns

- Les designs patterns décrivent une solution simple et élégante à des problèmes spécifiques en conception orienté objet.
- L'utilisation des designs patterns rend la conception plus flexible, modulable, réutilisable et surtout compréhensible.

Les Designs Patterns

- Les designs patterns décrivent une solution simple et élégante à des problèmes spécifiques en conception orienté objet.
- L'utilisation des designs patterns rend la conception plus flexible, modulable, réutilisable et surtout compréhensible.
- Les designs patterns utilise un niveau d'abstraction élevé qui permet d'élaborer des constructions logiciels de meilleurs qualité.

Les Designs Patterns

- Les designs patterns décrivent une solution simple et élégante à des problèmes spécifiques en conception orienté objet.
- L'utilisation des designs patterns rend la conception plus flexible, modulable, réutilisable et surtout compréhensible.
- Les designs patterns utilise un niveau d'abstraction élevé qui permet d'élaborer des constructions logiciels de meilleurs qualité.
- Une bonne connaissance des designs patterns est nécessaire pour une utilisation optimale.

Les Designs Patterns

Les designs patterns sont caractérisés par :

- **Nom** : Permet une standardisation pour une meilleure utilisation.

Les Designs Patterns

Les designs patterns sont caractérisés par :

- **Nom** : Permet une standardisation pour une meilleure utilisation.
- **Problème** : définis le contexte d'utilisation le mieux adapté.

Les Designs Patterns

Les designs patterns sont caractérisés par :

- **Nom** : Permet une standardisation pour une meilleure utilisation.
- **Problème** : définis le contexte d'utilisation le mieux adapté.
- **Solution** : présente de manière abstraite les éléments de la solution, leurs relations, responsabilités et collaboration.

Les Designs Patterns

Les designs patterns sont caractérisés par :

- **Nom** : Permet une standardisation pour une meilleure utilisation.
- **Problème** : définis le contexte d'utilisation le mieux adapté.
- **Solution** : présente de manière abstraite les éléments de la solution, leurs relations, responsabilités et collaboration.
- **Conséquences** : les résultats et compromis retournés par l'utilisation du design pattern.

Les Designs Patterns

Il existe plusieurs designs patterns, ils se décomposent en 3 groupes :

- Les patterns de création : relis au processus de création d'objets.

Les Designs Patterns

Il existe plusieurs designs patterns, ils se décomposent en 3 groupes :

- Les patterns de création : relis au processus de création d'objets.
- Les patterns de structure : définis la compositions des classes et des objets.

Dans chaque groupes, il y des patterns qui traitent les relations entre objets et d'autres qui traitent ceux entre classes.

Les Designs Patterns

Il existe plusieurs designs patterns, ils se décomposent en 3 groupes :

- Les patterns de création : relis au processus de création d'objets.
- Les patterns de structure : définis la compositions des classes et des objets.
- Les patterns de comportement : caractérise les interactions entre classes/objets, ainsi que la distribution des responsabilités.

Dans chaque groupes, il y des patterns qui traitent les relations entre objets et d'autres qui traitent ceux entre classes.

Les Designs Patterns

	Création	**Structure**	**Comportement**
Classe	Factory Method	Adapter	Interpreter Template Method
Objet	Abstract Factory Builder Singleton	Adapter Composite Decorator Proxy	Iterator Observer Strategy Visitor

Les Designs Patterns

- **Les patterns de création de classes** reportent une partie de la création d'objet aux sous classes, alors que cette création est reportée à un autre objet pour les **patterns de création d'objets**.

Les Designs Patterns

- **Les patterns de création de classes** reportent une partie de la création d'objet aux sous classes, alors que cette création est reportée à un autre objet pour les **patterns de création d'objets**.
- **Les patterns de structure de classes** utilisent l'héritage pour composer les classes, alors que **ceux d'objets** décrivent la manière d'assembler les objets entre eux.

Les Designs Patterns

- **Les patterns de création de classes** reportent une partie de la création d'objet aux sous classes, alors que cette création est reportée à un autre objet pour les **patterns de création d'objets**.
- **Les patterns de structure de classes** utilisent l'héritage pour composer les classes, alors que **ceux d'objets** décrivent la manière d'assembler les objets entre eux.

Les Designs Patterns

- **Les patterns de création de classes** reportent une partie de la création d'objet aux sous classes, alors que cette création est reportée à un autre objet pour les **patterns de création d'objets**.
- **Les patterns de structure de classes** utilisent l'héritage pour composer les classes, alors que **ceux d'objets** décrivent la manière d'assembler les objets entre eux.
- **Les patterns de comportement de classes** utilisent l'héritage pour définir les algorithmes, ou **les patterns de comportement d'objets** montrent les interactions entre les objets afin de réaliser une fonctionnalité spécifique.

Les Patterns de Création

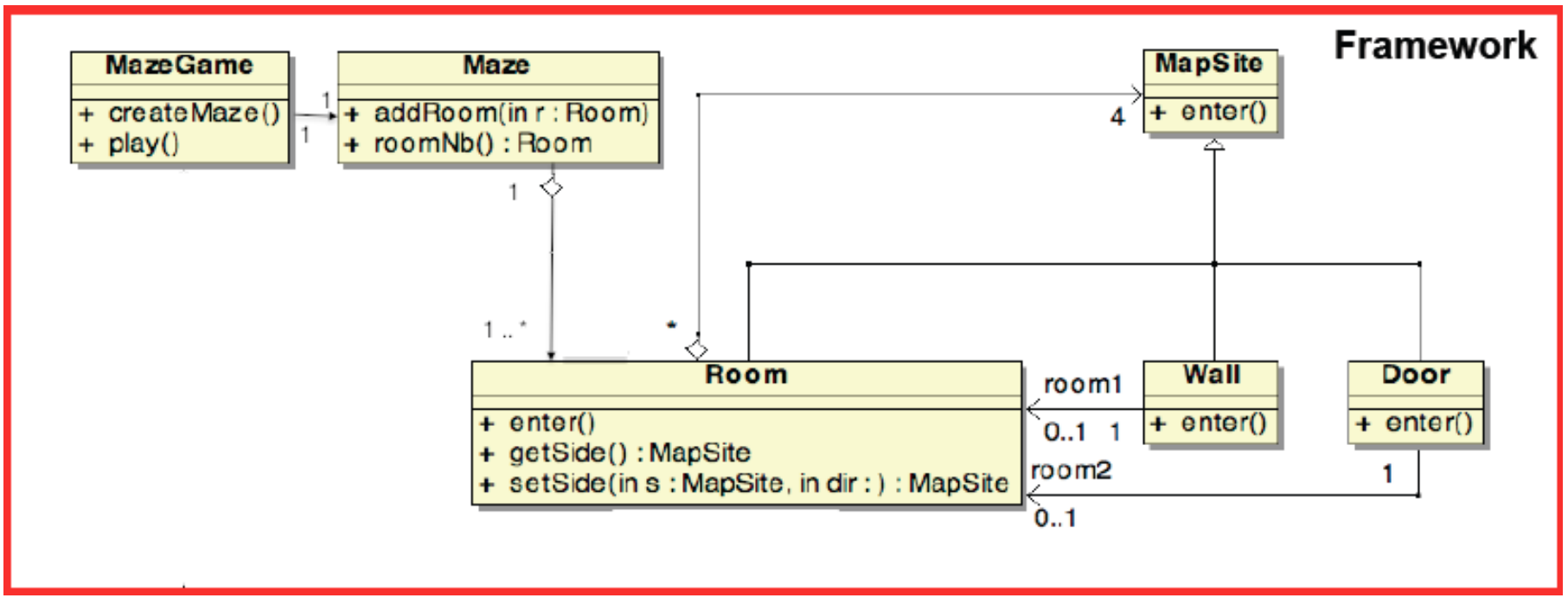
Les Designs Patterns

- Déléguer à d'autres classes la création des objets.
- Abstraire le processus d'instanciation.
- Rendre le système indépendant de la façon dont les objets sont crées, composés, assemblés, représentés.
  - Encapsuler les informations de la classe concrète que le système utilise.
  - Rendre le système flexible en cachant les information sur ce qui est créé, par qui, quand et comment.

Example : Labyrinthe

- On se concentre sur la création du labyrinthe(maze).
- Le labyrinthe est représente par un ensemble de chambres(rooms), de murs(walls) et de portes(doors).
- Une chambre a connaissance de ses voisins (chambres, murs, portes).
- Une chambre a 4 côtés (sides) : nord, sud, est, ouest.

Example : Labyrinthe

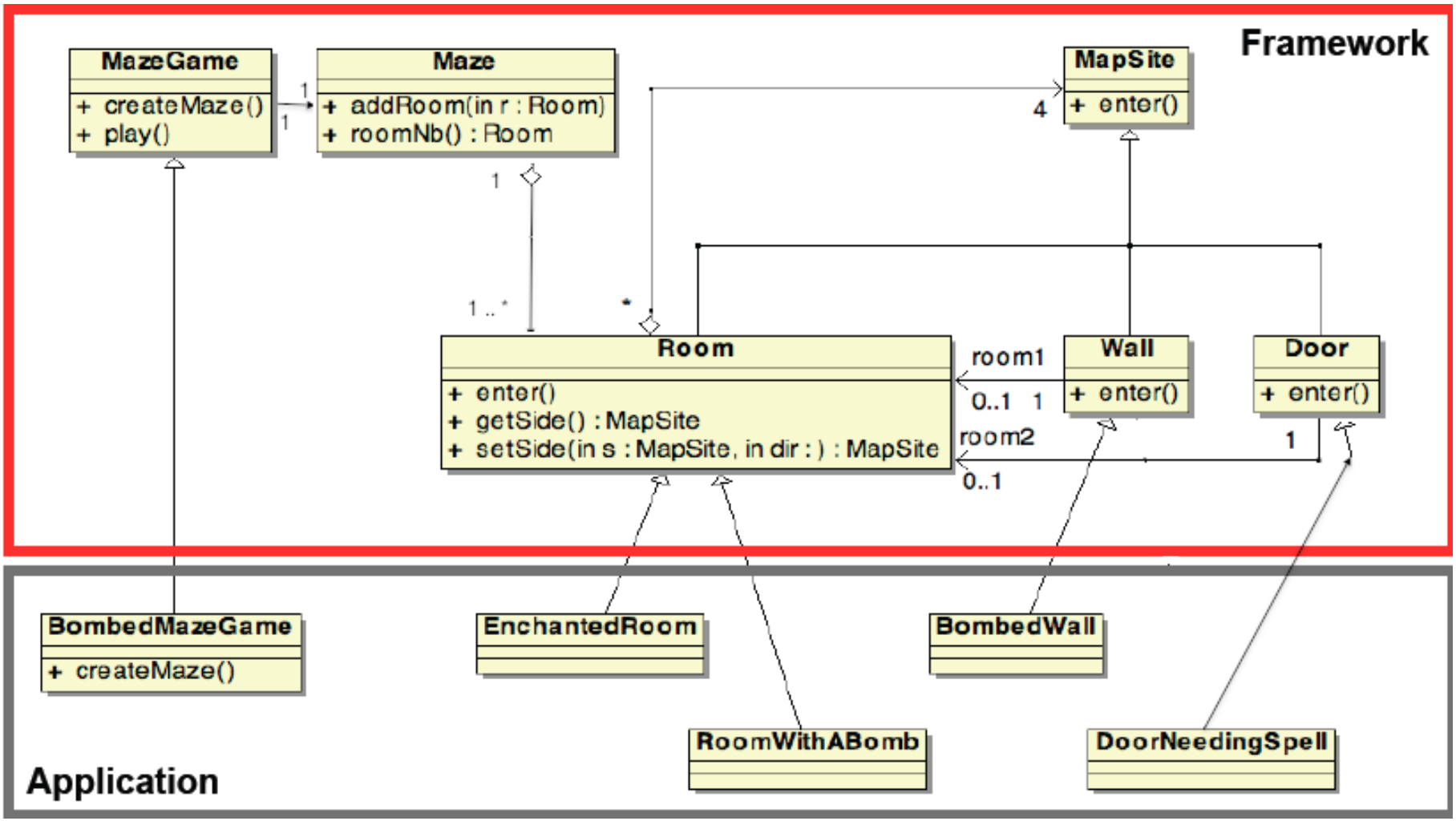


Example : Labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new Wall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new Wall());
      r1.setSide(West, new Wall());
      r2.setSide(North, new Wall());
      r2.setSide(East, new Wall());
      r2.setSide(South, new Wall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

Example : Labyrinthe



Example : Labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new Wall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new Wall());
      r1.setSide(West, new Wall());
      r2.setSide(North, new Wall());
      r2.setSide(East, new Wall());
      r2.setSide(South, new Wall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {
…
   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new RoomWithABomb(1);
      Room r2 = new RoomWithABomb(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new BombedWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new BombedWall());
      r1.setSide(West, new BombedWall());
      r2.setSide(North, new BombedWall());
      r2.setSide(East, new BombedWall());
      r2.setSide(South, new BombedWall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

Example : Labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new Wall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new Wall());
      r1.setSide(West, new Wall());
      r2.setSide(North, new Wall());
      r2.setSide(East, new Wall());
      r2.setSide(South, new Wall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {
…
   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new RoomWithABomb(1);
      Room r2 = new RoomWithABomb(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new BombedWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new BombedWall());
      r1.setSide(West, new BombedWall());
      r2.setSide(North, new BombedWall());
      r2.setSide(East, new BombedWall());
      r2.setSide(South, new BombedWall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

Manque de flexibilité.

Le Modèle de Fabrique

Nom :

Le Modèle de Fabrique (The Factory Pattern).

Problème :

On utilise la modèle de fabrique lorsque :

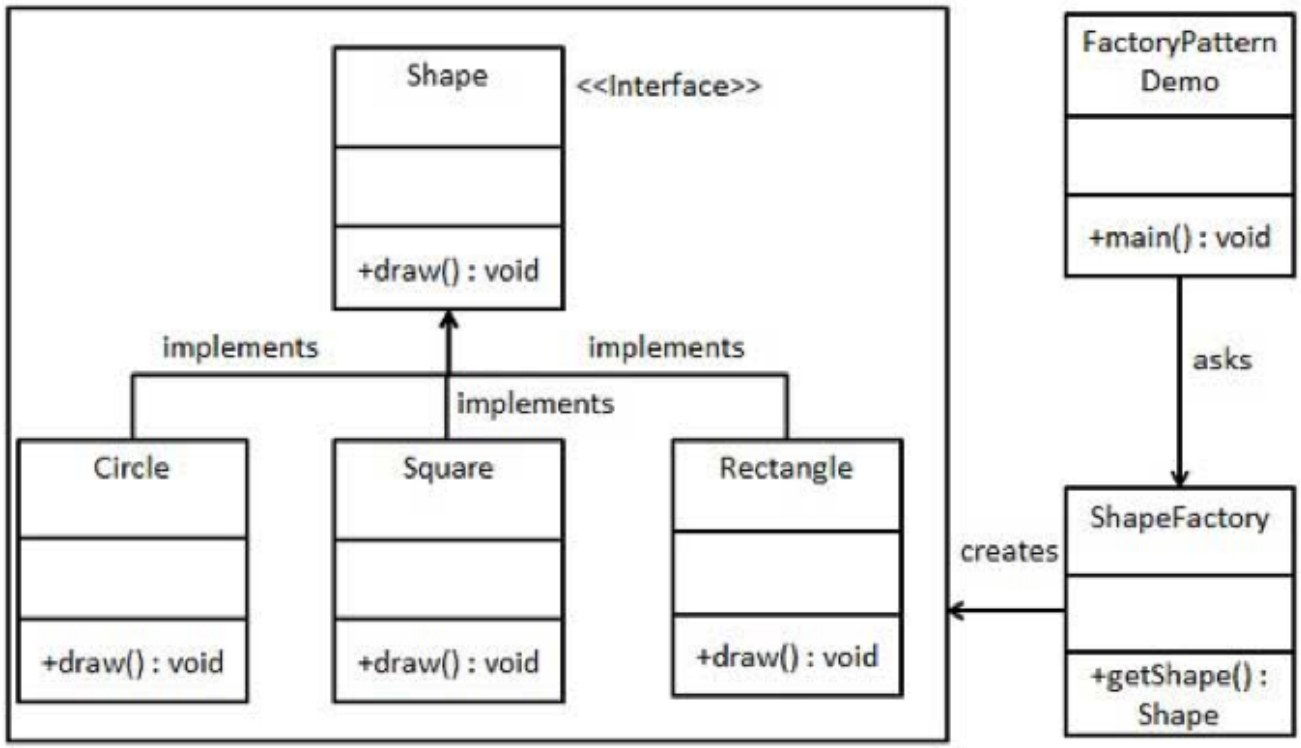
- Une classe ne peut anticiper la classe de l'objet qu'elle doit créer.
- Une classe délègue la responsabilité de la création à ses sous classes, tout en concentrant l'interface dans une classe unique.

Le Modèle de Fabrique

Conséquences :

- Avantage : Élimination du besoin de code spécifique à l'application dans le code du framework (uniquement l'interface du **Product**).
- Inconvénient : Augmentation du nombre de classes.

Le Modèle de Fabrique : Exemple d'application



Le Modèle de Fabrique

```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = makeMaze();
      Room r1 = makeRoom(1);
      Room r2 = makeRoom(2);
      Door theDoor = makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, makeWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, makeWall());
      r1.setSide(West, makeWall());
      r2.setSide(North, makeWall());
      r2.setSide(East, makeWall());
      r2.setSide(South, makeWall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {
…
   public Wall makeWall() {
      return new BombedWall();
   }

   public Room makeRoom(int i) {
      return new RoomWithABomb(i);
   }
}
```

```
class EnchantMazeGame extends MazeGame {
…
   public Wall makeWall() {
      return new EnchantWall();
   }

   public Room makeRoom(int i) {
      return new EnchantedRoom (i);
   }
}
```

Le Modèle de Fabrique

```
class  ImplemMazeFactory  {
--
  public static void     main      () {
    Maze labyrinthe = Null;
   System.out.println(" veuillez indiquer le type du labyrinthe");
   Scanner sc = new Scanner(System.in);
              int rep = sc.nextInt();
  If (rep == 1) then labyrinthe = BombeMazeGame.createMaze();
  else if (rep == 2) then labyrinthe = EnchantMazeGame.createMaze();
   else labyrinthe = MazeGame.createMaze();
}
```

La Fabrique Abstraite

Nom :

Le Modèle de Fabrique Abstraite (The Abstract Factory Pattern).

Problème :

Généralement le modèle de la fabrique est utilisé lorsque :

- Le système travaille avec un ensemble de produit mais à besoin de rester indépendant du type de ces produits.
- Le système doit être indépendant de la manière dont ses produits sont créés, assemblés, représentés.
- Le système veut définir une interface unique à une famille de produits concrets.

La Fabrique Abstraite

Nom :

Le Modèle de Fabrique Abstraite (The Abstract Factory Pattern).

Problème :

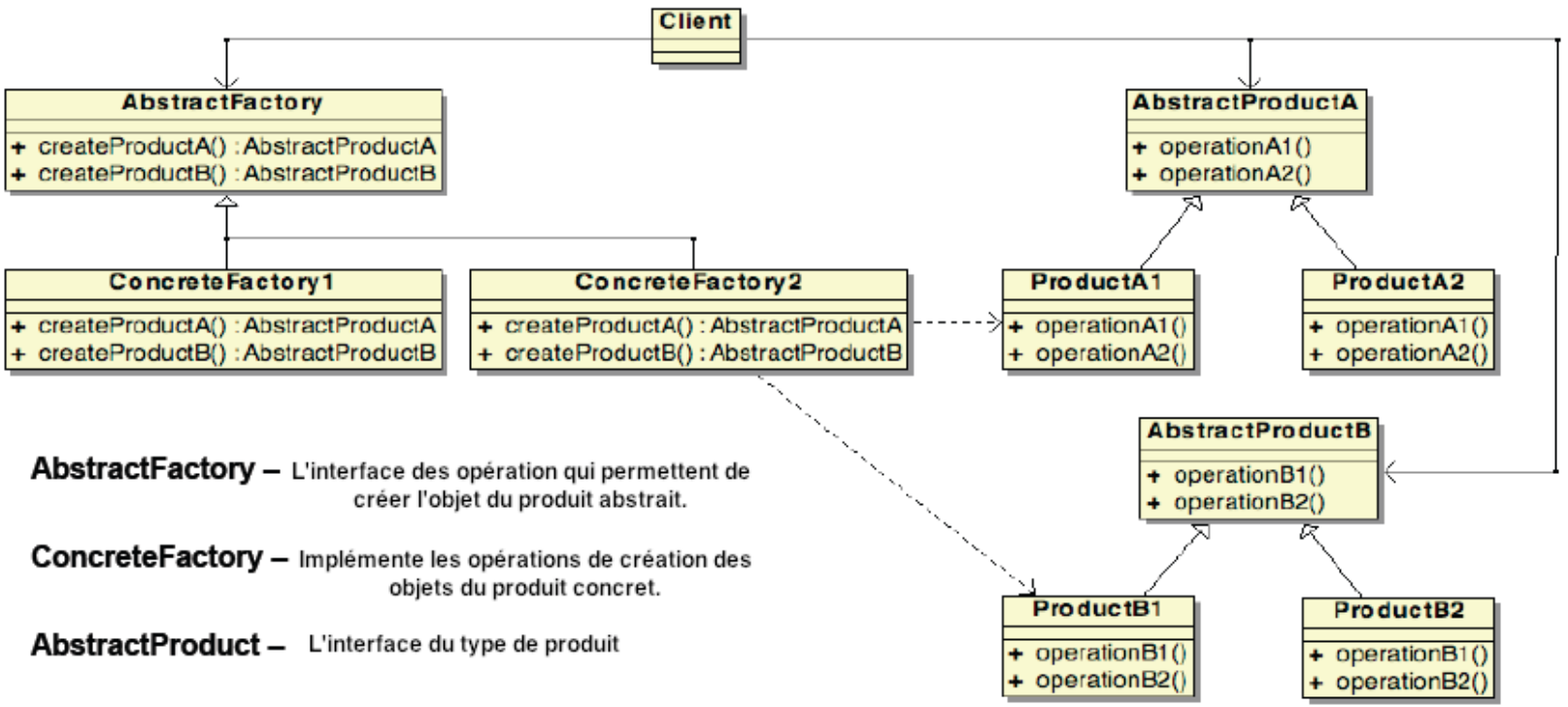
Généralement le modèle de la fabrique est utilisé lorsque :

- Le système travaille avec un ensemble de produit mais à besoin de rester indépendant du type de ces produits.
- Le système doit être indépendant de la manière dont ses produits sont créés, assemblés, représentés.
- Le système veut définir une interface unique à une famille de produits concrets.

La fabrique des fabriques (Super fabrique.

La Fabrique Abstraite

Solution :



AbstractFactory – L'interface des opération qui permettent de créer l'objet du produit abstrait.

ConcreteFactory – Implémente les opérations de création des objets du produit concret.

AbstractProduct – L'interface du type de produit

ConcreteProduct

- définis l'objet produit qui va être creer par la fabrique concréte correspondante.
- Implémente l'interface AbstractProduct

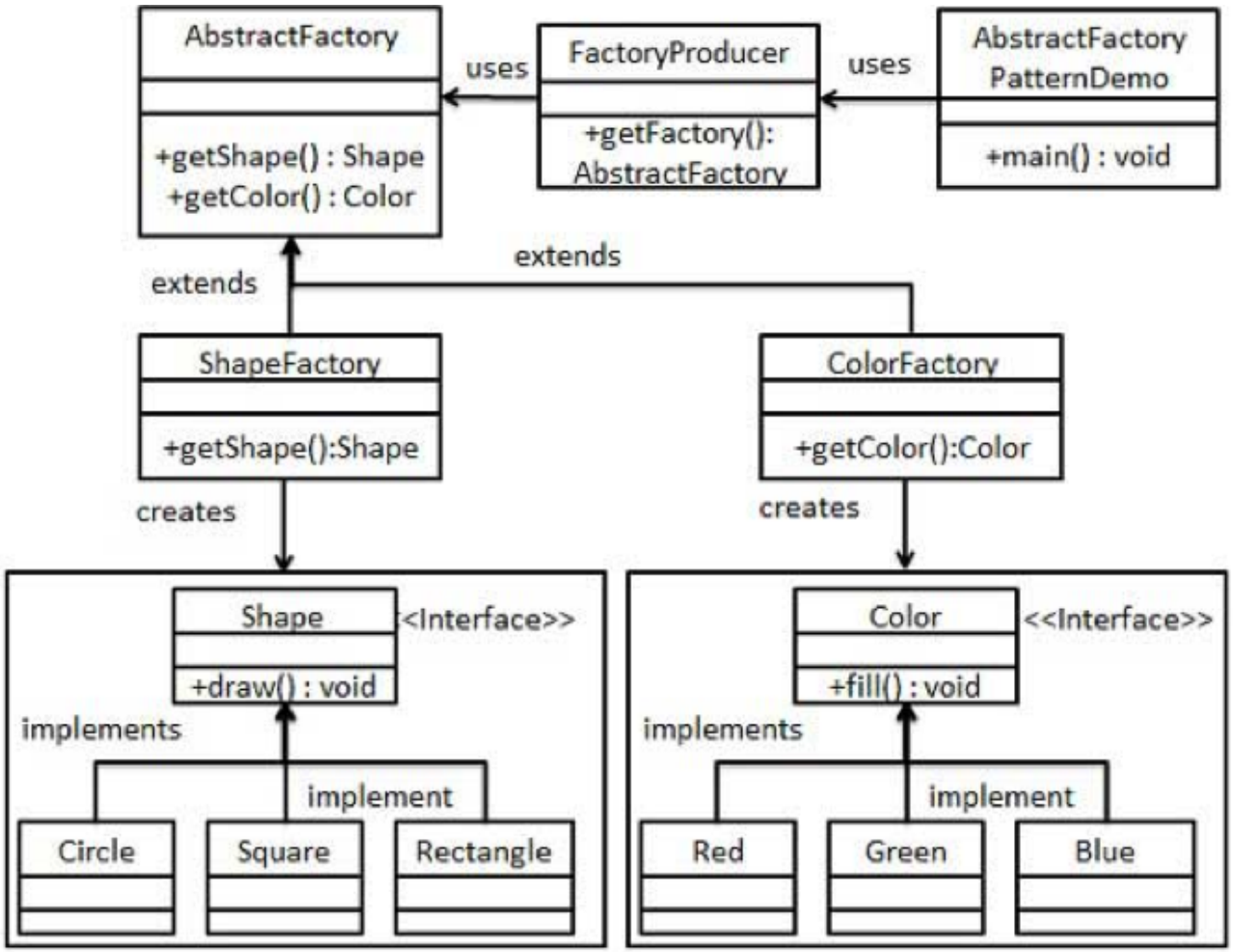
Client – Utilise les inetrface AbstractFactory et AbstractProduct.

La Fabrique Abstraite

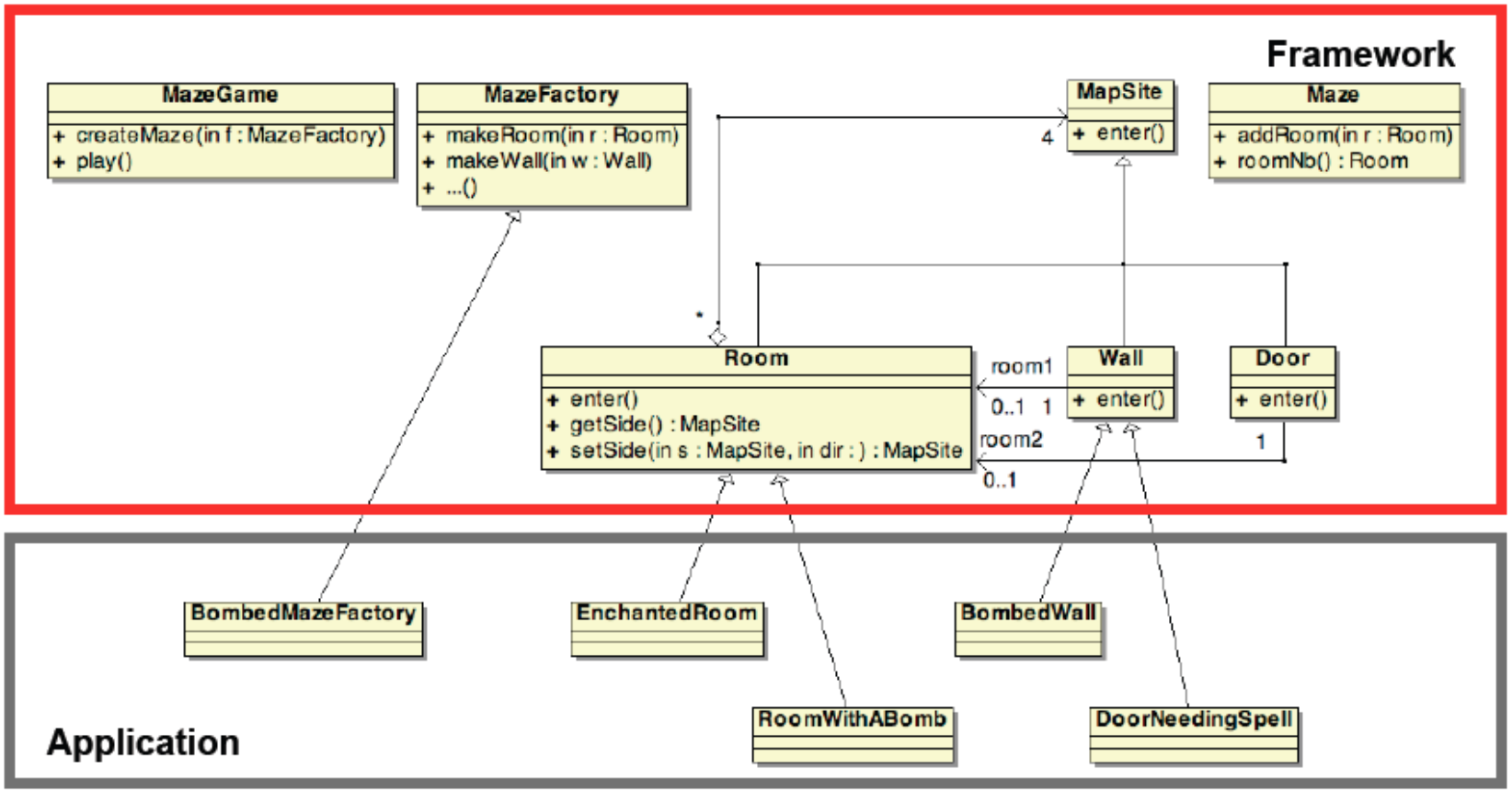
Conséquences :

- Avantage : Séparation entre classes concrètes et classe clients.
  - Les nom des classes produits n'apparaissent pas dans le code client.
  - Facilite l'échange de familles de produits.
  - Favorise la cohérence entre produits.
  - Le processus de création est isolé dans une classe.
- Inconvénient : La mise en place de nouveaux produits dans l'AbstractFactory n'est pas facile.

La Fabrique Abstraite : Exemple d'application



La Fabrique Abstraite : Le labyrinthe



La Fabrique Abstraite : Le labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze(MazeFactory f) {
      Maze aMaze = f.makeMaze();
      Room r1 = f.makeRoom(1);
      Room r2 = f.makeRoom(2);
      Door theDoor = f.makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, f.makeWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, makeWall());
      r1.setSide(West, makeWall());
      r2.setSide(North, makeWall());
      r2.setSide(East, makeWall());
      r2.setSide(South, makeWall());
      r2.setSide(West, theDoor);
      return aMaze;
  }
}
```

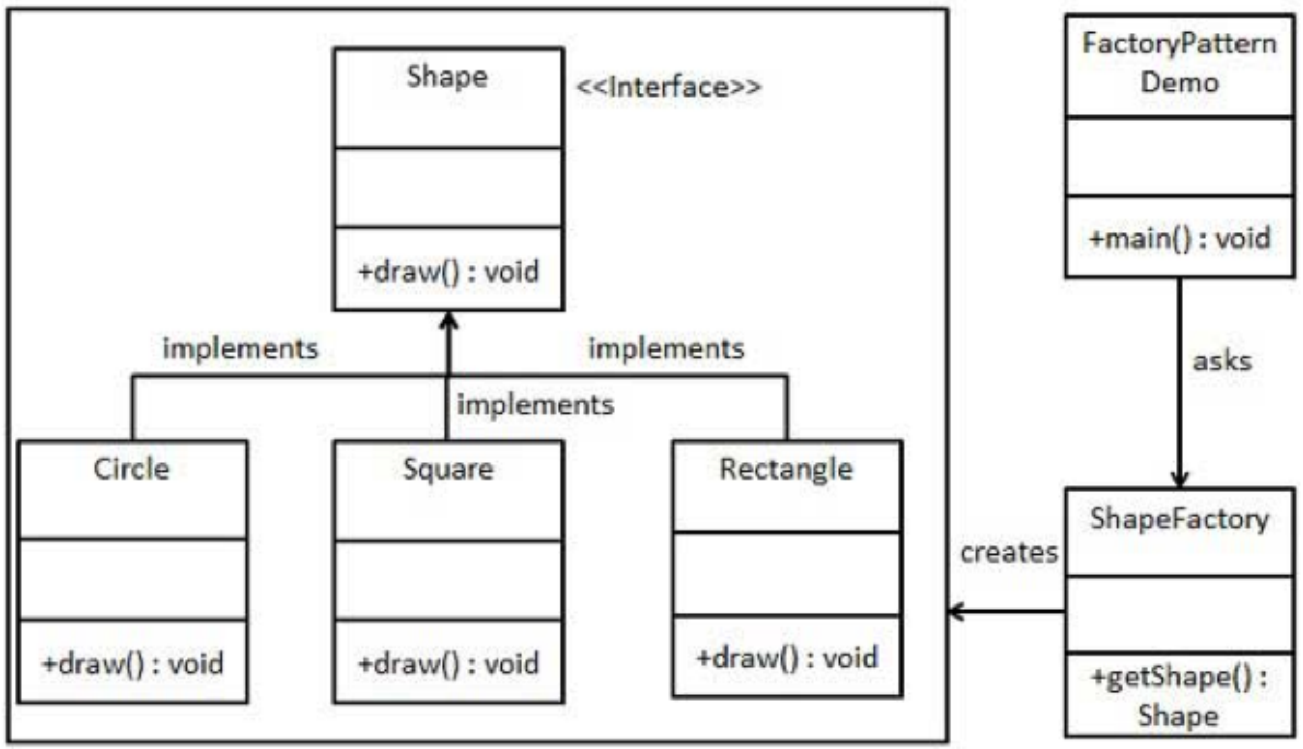
```
class BombedMazeFactory extends MazeFactory {
…
   public Wall makeWall() {
      return new BombedWall();
   }

   public Room makeRoom(int i) {
      return new RoomWithABomb(i);
   }
}
```

Exemple : Le labyrinthe

- Le modèle de fabrique abstraite : la méthode de fabrique CreateMaze a un objet en paramètre utilisé pour créer les composants du Labyrinthe, la structure du labyrinthe peut changer en passant un autre paramètre.
- Le modèle de fabrique : la méthode CreateMaze appelle des fonctions virtuelles au lieu de constructeurs pour créer les composants du Labyrinthe, il est alors possible de modifier la création en dérivant une nouvelle classe et en redéfinissant ses fonctions virtuelles.

Le Modèle de Fabrique : Exemple d'application



proposer une implémentation à ce modèle.