Interrogation - EPSI – Design Pattern

Nom :

Prénom :

Il faut cocher les ronds qui possèdent les bonnes réponses. Il peut y avoir de 1 à n réponses par question. Toute question avec une erreur ne rapporte pas de point. Vous avez le droit à tous supports.

1. Quelles assertions sont vraies à propos des « design pattern » ?

* Ils introduisent un vocabulaire
* Les patrons de conception == « design pattern » en anglais
* Les « Design Pattern » fournissent une solution technique liée à une problématique d’architecture et/ou de code
* Les « design patten » sont dépendants du langage de programmation
* Les « Design Pattern » permettent de faciliter la communication entre développeurs

1. Quelles assertions sont vraies ?

* Il y a deux grandes familles de Design Pattern, les pattern « GRASP » et les seconds les patterns du « Gang of Four »
* Les anti-patterns montrent ce qu'il faut vraiment faire.
* GRASP signifie General Responsability Assignment Software Patterns
* GoF signifies du Gang Of Five.

1. Quels sont les difficultés avec les patterns ?

* Risque de se tromper de vocabulaire
* Nécessite plusieurs années de pratique avant de bien les maîtriser.
* Risque d’utiliser un pattern inadapté à la situation et donc de complexifier le code
* N’en connaitre qu’un seul et l’utiliser partout
* Beaucoup de pattern se ressemblent et certains sont la compositions/évolution de pattern existant

1. Quelles affirmations sont vraies à propos de ce code

|  |
| --- |
| public static decimal CalculateBasketAmount(IList<BasketLineArticle> basketLineArticles)  {  var amountTotal = 0;  foreach (var basketLineArticle in basketLineArticles)  {  var article = GetArticleDatabase(basketLineArticle.Id);  var amount = 0;  switch (article.Category)  {  case "food":  amount += article.Price \* 100 + article.Price \* 12;  break;  case "electronic":  amount += article.Price \* 100 + article.Price \* 20 + 4 \* 100;  break;  case "desktop":  amount += article.Price \* 100 + article.Price \* 20;  break;  }  amountTotal += amount \* basketLineArticle.Number;  }  return amountTotal;  } |

**Exemple de code**

* Cette méthode est très cohérente
* Cette méthode réalise l’appel à une procédure qui récupère des articles d’une base de données et réalise des règles métier
* Cette méthode possède du code impératif
* Cette méthode ne réalise pas d’appel à une procédure
* Cette méthode possède de la duplication de code

1. Un train roule sur des rails, il possède des wagons qui possèdent lui-même des sièges. Les réservations sont en cours pour le train Paris/Brest. Vous êtes en train de coder une application exposée par borne SNCF hébergée dans la gare. Quel bout de code semble le plus approprié afin de réserver un siège dans le train depuis votre « controller ». Il faut savoir qu’il existe des règles métier qui font que les sièges sont remplis en fonction de règles liés aux Wagons (par exemple on commence à remplir un wagon à partir du moment ou le premier wagon est rempli a plus de 50%).

* var result = rail.Reserver()
* var result = siege.Reserver()
* var result =wagon.Reserver()
* var result = train.Reserver()
  1. Cette question est liée à la précédente. Dans votre modèle objets Tain/Siege/Rail/Wagon. Quel objet sera responsable de gérer les « sieges » ?
  + Rail
  + Siege
  + Wagon
  + Train

1. Un garage Norauto possède des garagistes qui ont eu même certaines compétences (pneu, courroie, caisse, etc..). Uniquement le directeur du magasin a le droit d’ajouter une compétence à un garagiste. Un garage répare au quotidien des dizaines de voitures. Quels sont les bouts de code qui semble bien respecter les rôles et responsabilités de chacun d’un point de vu objet et possèdent assez d’informations pour bien fonctionner.

* var result = garage.AjoutCompetence(‘’id-garagiste‘’, ‘’pneu’’) ;
* var result = garage.AjoutCompetence(‘’pneu’’) ;
* var result = garagiste.AjoutCompetence(‘’pneu’’) ;
* var result = siegeSocialFrance.AjoutCompetence(‘’id-magasin’’, ‘’pneu’’) ;
* var result = voiture.AjoutCompetence(‘’id-garagiste‘’, ‘’pneu’’) ;
  1. Cette question est liée à la précédente. Quelles assertions sont les bonnes et respectent les « patterns » d’affectation de responsabilité de GRASP (modèle objet).
  + L’objet « Garage » est géré par « SiegeSocialFrance »
  + L’objet « Directeur » est géré par « Garage »
  + L’objet « Garagiste » est géré par « Garage »
  + L’objet « Compétence » est géré par « Voiture »
  + L’objet « Compétence » est géré par « SiegeSocialFrance »
  + L’objet « Compétence » est géré par « Garagiste »

1. Quel pattern GRASP faut-il utiliser afin de protéger mon application du code pas très performant qui permet de logger ci-dessous ?

|  |
| --- |
| public static class ExternalLogger  {  private static void Write(string message)  {  var codeBase = Assembly.GetExecutingAssembly().CodeBase;  var uri = new UriBuilder(codeBase);  var path = Uri.UnescapeDataString(uri.Path);  var assemblyDirectory = Path.GetDirectoryName(path);  string filePath = Path.Combine(assemblyDirectory, "log.txt");  File.AppendAllText(filePath, message, Encoding.UTF8);  }  public static void LogInformation(string message)  {  Thread.Sleep(300);  Write($"Information: {message}{Environment.NewLine}");  }  public static void LogError(Exception exception,string message)  {  Thread.Sleep(700);  Write($"Error: {message} {exception}");  }  } |

**Exemple de code**

* Protection des variations
* Contrôleur
* Polymorphisme
* Pure invention
* Indirection

1. Quel pattern GRASP allez-vous maintenant utiliser afin de de continuer à implémenter proprement toutes les règles de calcul des articles et ainsi augmenter la cohérence du code (ainsi que ça scalabilité). Sachant que vous allez devoir coder une 100ène de catégorie d’article et que vous avez actuellement le code ci-dessous :

|  |
| --- |
| public class Article  {  public Article(string id, int price, string category)  {  Id = id;  Category = category;  Price = price;  }  public int Price { get; }  public string Id { get; }  public string Category { get; }  public int CalculateAmout()  {  switch (Category)  {  case "food":  return Price \* 100 + Price \* 12;  case "electronic":  return Price \* 100 + Price \* 20 + 4;  case "desktop":  return Price \* 100 + Price \* 20;  case "toy":  var newPrice = Price \* 100 - Price \* 30;  return (newPrice \* 100 + newPrice \* 20) / 100;  default:  throw new NotImplementedException();  }  }  } |

**Exemple de code**

* Protection des variations
* Contrôleur
* Command
* Polymorphisme
* Pure invention
* Prototype
* Indirection

1. Pour rendre le code plus cohérent avez dû extraire la classe ci-dessous. A quel « pattern » répond cette classe d’un point de vu GRASP ?

|  |
| --- |
| public class ArticleDatabaseJson  {  public ArticleDatabase GetArticle(string id)  {  var codeBase = Assembly.GetExecutingAssembly().CodeBase;  var uri = new UriBuilder(codeBase);  var path = Uri.UnescapeDataString(uri.Path);  var assemblyDirectory = Path.GetDirectoryName(path);  var jsonPath = Path.Combine(assemblyDirectory, "article-database.json");  IList<ArticleDatabase> articleDatabases =  JsonConvert.DeserializeObject<List<ArticleDatabase>>(File.ReadAllText(jsonPath));  var article = articleDatabases.First(articleDatabase => articleDatabase.Id == id);  return article;  }  } |

**Exemple de code**

* Protection des variations
* Contrôleur
* Factory
* Polymorphisme
* Inversion Of Control
* Pure invention
* Indirection

1. Quelles Assertions sont vraies à propos du Test Driven Development ?

* Permet de développer plus vite car automatise le fait de tester le bout de code que l’on développe (plus besoin de réaliser 15 étapes manuelles pour tester ces modifications)
* On réalise le code du test après l’implémentation
* Permet de développer de manière itérative (test, code, refactor et ainsi de suite)
* On réalise le code du test avant l’implémentation

1. Quelles Assertions sont vraies ?

* Lorsque vous cliquer sur les pages de votre site web, vous réalisez un test unitaire
* Lorsque vous cliquer sur les pages de votre site, vous réaliser un test d’intégration
* Un test pour être unitaire doit être lié à une ressource externe (disque dur, appel réseaux, etc.)
* Un test d’intégration entraine la réalisation de vrai appel à une base de données
* Un test d’intégration ne va pas vraiment envoyer d’email via un serveur smtp comme la fonctionnalité le demande.
* Un test d’intégration va réellement accéder à des données présentent sur un disque dur.
* Un test unitaire va utiliser ce que l’on appelle des « mock » pour simuler les appels aux ressources externes.

1. Pour qu’une application soit maintenable il faut :

* Fort couplage
* Faible cohérence
* Beaucoup de duplication de code
* Utiliser le ou les « patterns » adaptés à la situation
* Faible couplage
* Réutiliser les mêmes objets pour deux besoins différents

1. Pour qu’une application soit maintenable il faut :

* Des nommages avec de nombreux raccourcis
* Une forte cohérence
* Un objet doit être lié à beaucoup d’autre objet
* Injecter en entrée des méthodes uniquement ce qui est nécessaire
* Du code très lisible
* Ne pas réutiliser le même objet pour deux besoins différents

1. Parmi les « design pattern » de comportement du « Gang Of Four ». Lequel permet de réaliser deux composants graphiques qui communiquent entre eux sans pour autant se connaître directement ?

* Chain of Responsability
* Command
* Interpreter
* Iterator
* Mediator
* Memento
* Observer
* State
* Strategy
* Template

1. Quel Design pattern Gof (« Gang Of Four ») permet d’encapsuler la création de classe dérivée ? et comment s’appelle ce type de d’objet dans les design pattern GRASP ?

* Abstract Factory
* Builder
* Factory
* Polymorphisme
* Prototype
* Singleton
* Pure Invention
* Forte cohesion
* Indirection
* Protection de variation
* Faible Couplage

1. Que faites-vous lorsque vous réalisez du refactoring ?

* Vous corrigez des bogues
* Vous améliorer la lisibilité de votre code
* Vous supprimez la duplication de code
* Vous ajoutez de nouvelles fonctionnalités
* Vous simplifiez l’algorithmie du code et réduisez le couplage

1. Quel est le code ci-dessous qui est le moins couplée ?

* Code 1

|  |
| --- |
| public class Basket  {  public void AddGift(IList<Article> articles)  {  Articles.Add(new Gift());  }  } |

**Code 1**

* Code 2

|  |
| --- |
| public class Basket  {  Private readonly IList<Article> \_articles;  public Basket(IList<Article> articles)  {  \_articles = articles ;  }  public IList<Article> AddGift(IList<Article> articles)  {  \_articles.Add(new Gift());  Return \_articles;  }  } |

**Code 2**

* Code 3

|  |
| --- |
| public class Basket  {  public IImmuableList<Article> AddGift(IImmuableList<Article> articles)  {  Return articles.Add(new Gift());  }  } |

**Code 3**

1. Quels sont les avantages qu’apporte le pattern représenté par le diagramme UML ci-dessous ? et dans quel cas il est pratique de l’utiliser ?



**UML class diagram**

* Permet de séparer celui qui invoquer une action et celui qui va recevoir le résultat
* Permet de gérer le cycle de vie des objets
* Permet de récupérer toujours la même instance d’un objet
* Pratique réaliser une queue de tâches à réaliser (par exemple pour un batch)

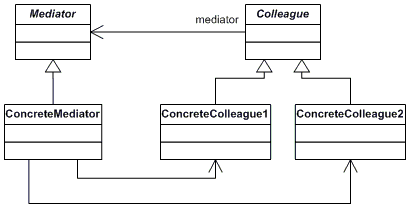
1. Quel sera le message affiché dans la console à l’exécution de ce programme ? et dans quel cas il est pratique de l’utiliser ?

|  |
| --- |
| 1. class MainApp 2. { 3. static void Main() 4. { 5. Singleton s1 = Singleton.Instance(); 6. Singleton s2 = Singleton.Instance();         if (s1 == s2)   1. { 2. Console.WriteLine("Objects are the same instance"); 3. } else { 4. Console.WriteLine("Objects are not the same instance");   }    2. Console.ReadKey(); 3. } 4. } 6. class Singleton 7. { 8. private static Singleton \_instance; 10. protected Singleton() 11. { 12. } 14. public static Singleton Instance() 15. {         // Note: this is not thread safe.   1. if (\_instance == null) 2. { 3. \_instance = new Singleton(); 4. } 6. return \_instance; 7. } 8. } |

**Exemple de code**

* Il sera affiché ("Objects are the same instance");
* Il sera affiché ("Objects are not the same instance");
* Il sert à toujours créer un nouvel objet
* Il sert à toujours retourner la même instance d’un objet
* Il sert à toujours créer un objet avec un « type » différent à chaque fois

1. Quels sont les avantages qu’apporte le pattern représenté par le diagramme UML ci-dessous ? et dans quel cas il est pratique de l’utiliser ?



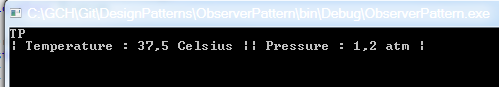
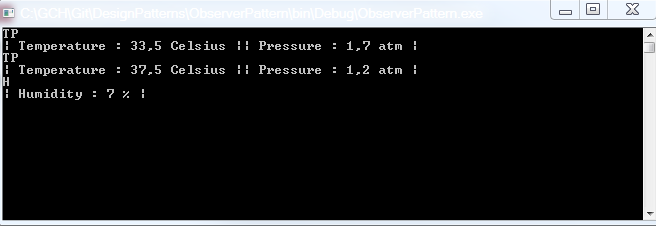
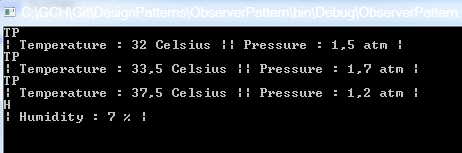
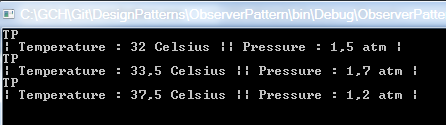
**UML class diagram**

* Ce pattern permet d’enlever le couplage entre deux objets
* Ce pattern permet à deux objets de communiquer directement de façon couplée
* Ce pattern sert à exécuter des tâches concrètes
* Ce pattern est très utilisé dans les interfaces graphiques et permet de faire des composants indépendants mais qui communique entre eux

1. Question Bonus (compte pour 2 points) : Quel est le message affiché dans la console à l’exécution de ce programme ?

|  |
| --- |
| static class Program  {  static void Main()  {  var provider = new WeatherSupplier();  var observer1 = new WeatherMonitor("TP");  var observer2 = new WeatherMonitor("H");  provider.WeatherConditions(32.0, 0.05, 1.5);  observer1.Subscribe(provider);  provider.WeatherConditions(33.5, 0.04, 1.7);  observer2.Subscribe(provider);  provider.WeatherConditions(37.5, 0.07, 1.2);  }  }  internal class Unsubscriber<TWeather> : IDisposable  {  private readonly List<IObserver<TWeather>> \_observers;  private readonly IObserver<TWeather> \_observer;  internal Unsubscriber(List<IObserver<TWeather>> observers, IObserver<TWeather> observer)  {  \_observers = observers;  \_observer = observer;  }  public void Dispose()  {  if (\_observers.Contains(\_observer))  \_observers.Remove(\_observer);  }  }  class Weather  {  public double Pressure { get; }  public double Humidity { get; }  public double Temperature { get; }  public Weather(double humd, double pres, double temp)  {  Temperature = temp;  Pressure = pres;  Humidity = humd;  }  }  sealed class WeatherMonitor : IObserver<Weather>  {  private IDisposable \_cancellation;  private readonly string \_name;  public void Subscribe(WeatherSupplier provider)  {  \_cancellation = provider.Subscribe(this);  }  public void Unsubscribe()  {  \_cancellation.Dispose();  }  public WeatherMonitor(string name)  {  \_name = name;  }    public void OnCompleted()  {  throw new **NotImplementedException**();  }  public void OnError(Exception error)  {  Console.WriteLine("Error has occured");  }  public void OnNext(Weather value)  {  Console.WriteLine(\_name);  if (\_name.Contains("T"))  {  string op = $"| Temperature : {value.Temperature} Celsius |";  Console.Write(op);    }  if (\_name.Contains("P"))  {  string op = $"| Pressure : {value.Pressure} atm |";  Console.Write(op);  }  if (\_name.Contains("H"))  {  string op = $"| Humidity : {value.Humidity\*100} % |";  Console.Write(op);  }  if (!(\_name.Contains("T") || \_name.Contains("P") || \_name.Contains("H")))  {  OnError(new Exception());  }  Console.WriteLine();  }  }  class WeatherSupplier : IObservable<Weather>  {  private readonly List<IObserver<Weather>> \_observers;  private List<Weather> Screens { get; }  private List<Weather> GetScreens()  {  return Screens;  }  public WeatherSupplier()  {  \_observers = new List<IObserver<Weather>>();  Screens = new List<Weather>();  }  public IDisposable Subscribe(IObserver<Weather> observer)  {  if (!\_observers.Contains(observer))  {  \_observers.Add(observer);  foreach (var item in GetScreens())  {  observer.OnNext(item);  }  }  return new Unsubscriber<Weather>(\_observers, observer);  }  public void WeatherConditions(double temp = 0, double humd = 0, double pres = 0)  {  var conditions = new Weather(humd, pres, temp);  foreach (var item in \_observers)  item.OnNext(conditions);  }  } |

**Exemple de code disponible ici:** [https://github.com/abishekaditya/DesignPatterns/](https://github.com/abishekaditya/DesignPatterns/tree/master/SingletonPattern)

* 
* 
* 
* 

1. Question Bonus : Quel est le design pattern utilisé dans la question ci-dessus ?

* Mediator
* Command
* Singleton
* Adapter
* Factory
* Observable

**Le génie logiciel**

Le génie logiciel est apparu dans les années 1970 sous la coordination de l'OTAN. Le génie logiciel a été mis sur pied pour répondre à la crise du logiciel (1968). À cette époque les logiciels n'étaient pas fiables et il était difficile pour les développeurs de fournir dans les délais une application respectant les spécifications du cahier des charges. Même quand le logiciel était bien construit, la difficulté de la maintenance se présentait.

La non qualité des systèmes fournis a entraîné plusieurs conséquences. On peut citer parmi les plus connues :

* Le projet TAURUS. Ce fut un projet d'informatisation de la bourse de Londres. Il a été abandonné définitivement après quatre années de labeur et a engendré environ 100 millions de livres de perte.
* Environ 280 civils ont été abattus lors de la guerre du Golfe. La cause fut que les abatteurs n'ont pas fait la différence entre un Airbus iranien (vol 655 d'Iran Air) transportant ces civils et un avion militaire.
* À cause du remplacement d'une virgule par un point, la mission VENUS a été un échec retentissant.

Cette liste est loin d'être exhaustive. Chaque année plusieurs milliards de dollars sont perdus dans l'industrie du logiciel. D'ailleurs les chiffres révélés par une étude menée par le Standish Group sur le comportement des projets informatiques le prouvent. L'étude menée en 1995 sur 8380 projets révèle que :

* 16% des applications sont construites avec succès ;
* 53% des projets aboutissent mais posent des problèmes tels : la diminution des fonctionnalités fixées de départ, le non-respect des délais spécifiés ou encore l'augmentation des coûts ;
* 31% des projets sont purement et simplement abandonnés.

Le génie logiciel procède en plusieurs étapes dans la fabrication d'un programme informatique. Ceci permet de garantir la qualité du produit et le respect des besoins de l'utilisateur tout en

Respectant les délais et les coûts fixés au départ. Après plusieurs travaux, les chercheurs ont défini la qualité d'un logiciel à partir de plusieurs facteurs, notamment :

* **La validité** : c'est la capacité du logiciel à remplir exactement les tâches énoncées lors de sa spécification.
* **La fiabilité et la robustesse** : c'est l'aptitude du logiciel à fonctionner en continu et même dans des conditions anormales.
* **La compatibilité** : le logiciel doit pouvoir être combiné très facilement à d'autres logiciels.
* **La réutilisabilité** : l'application doit pouvoir être utilisée dans son entièreté ou en partie dans un autre projet.
* **L'efficacité** : le logiciel doit pouvoir utiliser efficacement les ressources matérielles.
* **L'extensibilité** : le logiciel doit être facile à maintenir, il doit pouvoir accepter l'ajout de nouvelles fonctionnalités.
* **La portabilité** : le logiciel doit pouvoir s'exécuter sous plusieurs environnements différents.
* **L'intégrité** : le logiciel doit pouvoir protéger efficacement son code entier ou une partie de son code contre les accès non autorisés.

Il faut noter que certains des facteurs précédents sont contradictoires lors de leur réalisation effective. Vous devez effectuer un choix en fonction du contexte.

Sources : <https://openclassrooms.com/courses/creez-des-applications-de-qualite-avec-le-design-pattern-mvc/le-genie-logiciel-gl>