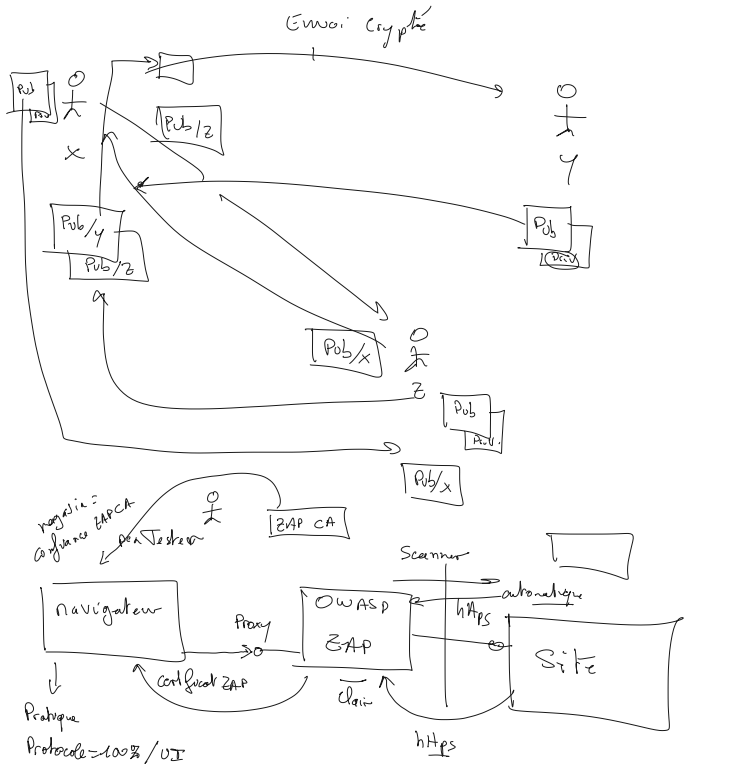
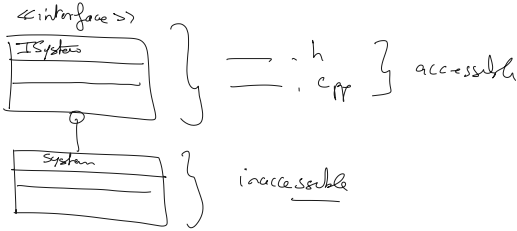
Secure C++


Envoi Crypté
Pub
Pub/2
Pub/4
Pub/3
Pub/x
Pub
A.V.
Pub/x
x
y
z
ZAP CA
Scanner
automatique
navigateur
Proxy
OWASP
ZAP
https
Site
configuré ZAP
Client
https
Pratique
Protocole = 100% / UI

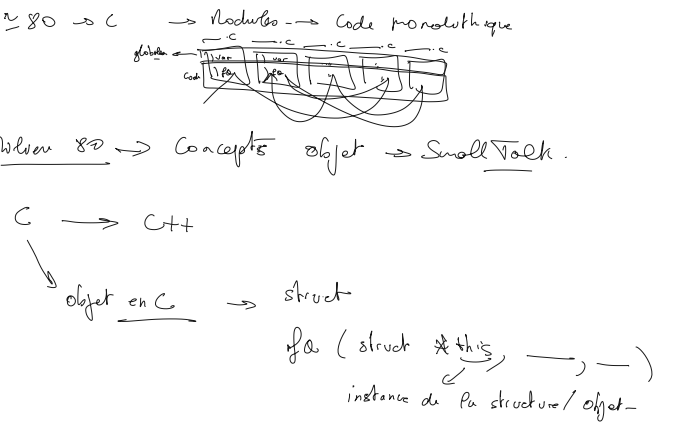
Encapsulation : "mettre dans une capsule"

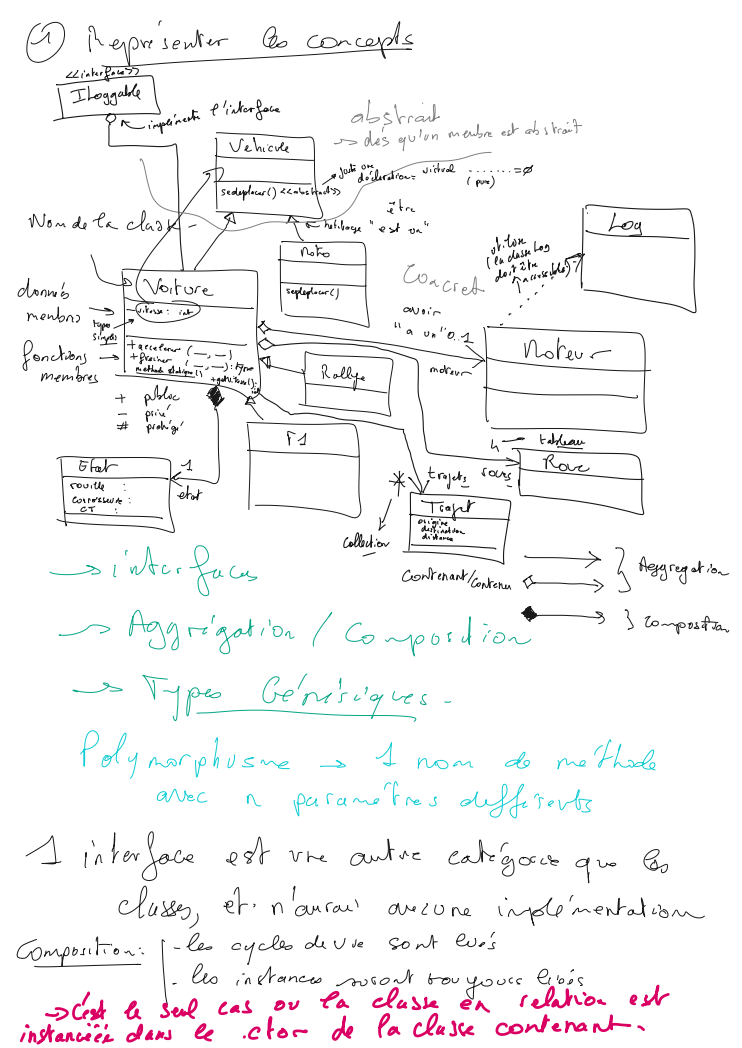
Public
Privé
Protégé
(interne)

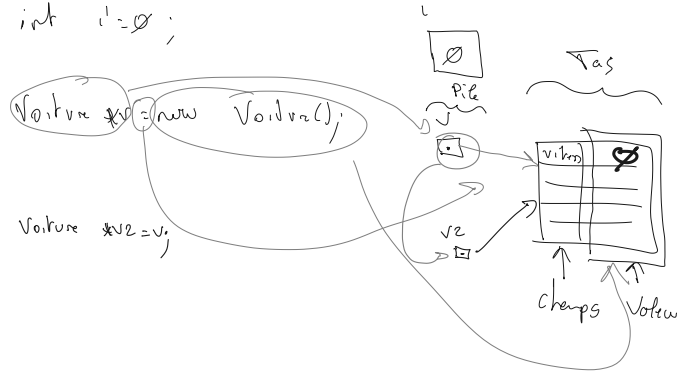


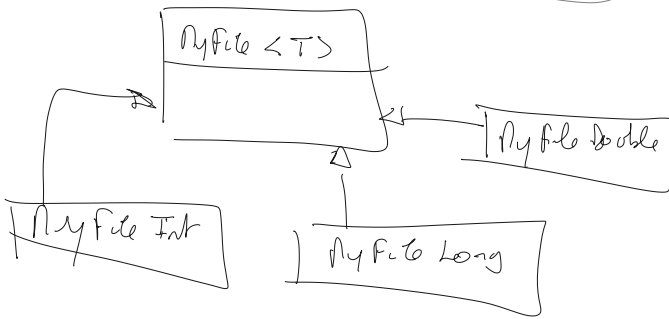
- Couplages
  - Fort
    - Simple, facile à comprendre
    - Problématique pour l'évolution et les mises à jour
  - Faible
    - Le client peut décider de quelle implémentation d'une interface obtenir.
    - Les évols doivent respecter l'interface.
    - La mise à jour de l'interface implique la mise à jour des clients.
    - Nécessite l'implémentation de pattern.
  - Lâche
    - Le client demande des implémentations d'un type d'interface. Il en trouvera, ou non.
    - Une implémentation pourrait ne pas être trouvée, il faudra faire avec.
  - Découplé
    - Chaque composant est développé sans tenir compte des autres, sauf rester conforme aux specs lors des évolutions (respect du versioning sémantique)
    - Ils doivent publier des interfaces, et une documentation, qui restera toujours ce qu'elle est au fil du temps, au risque de casser le système.
- SOLID :
  - Single responsibility principle
    - Une classe ou méthode ne doit avoir qu'une responsabilité
  - Open/closed principle
    - Une entité (Classe, méthode, ..) doit être fermée à la modification, mais permettre l'extension
  - Liskov substitution principle
    - Héritage : B doit pouvoir être étendue par D, sans modifier son comportement.
  - Interface segregation principle
    - Préférer n interfaces spécifiques pour un parent, plutôt qu'une générale.
  - Dependency inversion principle
    - Il faut dépendre des abstractions, par des implémentations.

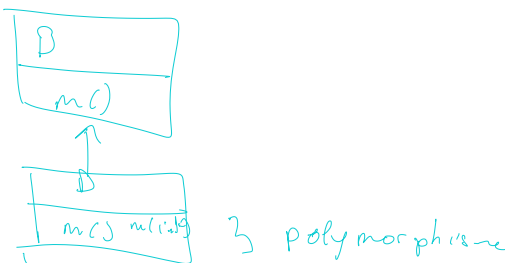
- GRASP - General responsibility assignment software patterns
  - Est une boite à outil mentale de bonnes pratiques
  - Contrôleur :
    - Les évènements du système doivent être traités par une classe non UI
    - Il se trouve derrière l'UI et coordonne aux objets sous jacents.
  - Créateur (Composition en UML) si :
    - Forte cohésion et dépendance entre les deux classes
    - La classe composée a une durée de vie associée
    - Ne sera jamais retransmise
    - Sera souvent privée pour le reste du système.
  - Forte cohésion :
    - Les tâches d'une classe lui sont fortement ciblées. Il ne fera pas autre chose.
    - Une faible cohésion est une classe qui fait de tout, trop de choses.
  - Indirection :
    - prends en charge le couplage faible entre deux objets en utilisant un objet intermédiaire.
    - Par exemple le Modèle sert de relation entre la Vue et le Controller
  - Spécialiste de l'information
    - Place la responsabilité d'une classe sur celle qui a le plus de renseignements pour l'accomplir. Elle aura alors des méthodes etc pour le faire.
  - Couplage faible
    - Peu de dépendances entre les classes
    - Un changement dans l'une ne devrait pas avoir d'impact dans l'autre
    - Forte réutilisabilité
  - Polymorphisme
    - Vertical (héritage) ou Horizontal (méthodes), il faut utiliser des types est méthodes selon ce que l'on doit faire, et non utiliser des branchements
  - Variations protégées
    - Déclarer en premier lieu des interfaces pour créer des implémentations.
  - Pure invention
    - Est l'équivalent d'un Service en Domain Driven Design.




1 Représenter les concepts
<<interface>>
Iloggable
implémente l'interface
abstrait
→ dès qu'un membre est abstrait
Vehicule
seDeplacer() <<abstract>>
peut être déclaration virtuel (pure) = ∅
Nom de la classe
être hérité "est un"
Log
Noté
seDeplacer()
Concret
avoir "a un" 0..1
Voiture
Citroen : cit
données membres
type signé
fonctions membres
+ accelerer (— , —)
+ Freiner (— , —) : type
méthode statique ()
seDeplacer()
+ public
- privé
protégé
Moteur
Rallye
F1
Etat
roulle :
carrosserie :
CT :
1
etat
tableau
Rue
trajets sans
Trajet
origine
destination
distance
Collection
Contenant/Contenu
Aggregation
Composition
→ interfaces
→ Aggrégation / Composition
→ Types Génériques.
Polymorphisme → 1 nom de méthode avec n paramètres différents
1 interface est une autre catégorie que les classes, et n'aura aucune implémentation
Composition:
- les cycles de vie sont liés
- les instances naissent toujours liés
→ C'est le seul cas où la classe en relation est instanciée dans le .ctor de la classe contenant.


int i=0;
i
Pile
Tas
Voiture *v1=new Voiture();
Voiture *v2=v1;
v1
v2
Champs
Voitures


MyFile<T>
MyFileInt
MyFileDouble
MyFileLong


B
m()
m() m(int)
} polymorphisme
surcharge (overload)

classes 100% concrètes

classes abstraites ↗ n% concrètes
↘ 100-n% abstraites

classes purement abstraites → 100% methodes abstraites (possibilité de données membres)

||

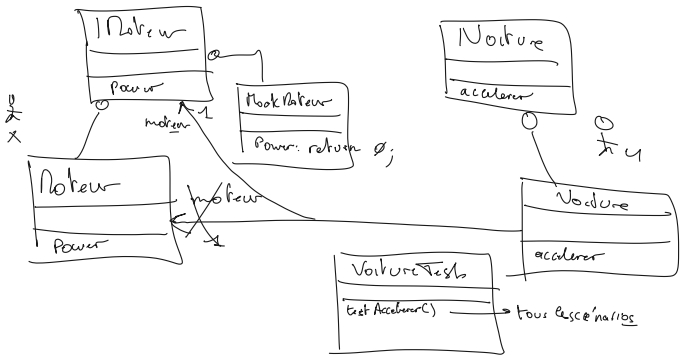
1 interface est "comme" une classe purement abstraite et sans données membres

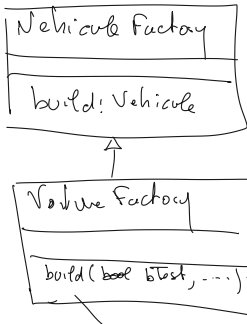
→ mot clef interface au lieu de class

Une interface est comme un définition un "contrat"

---

interface et tests unitaires

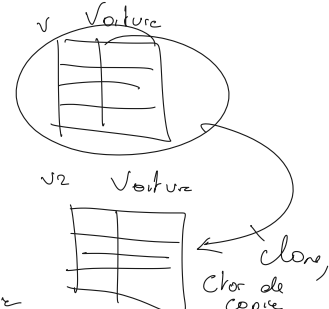




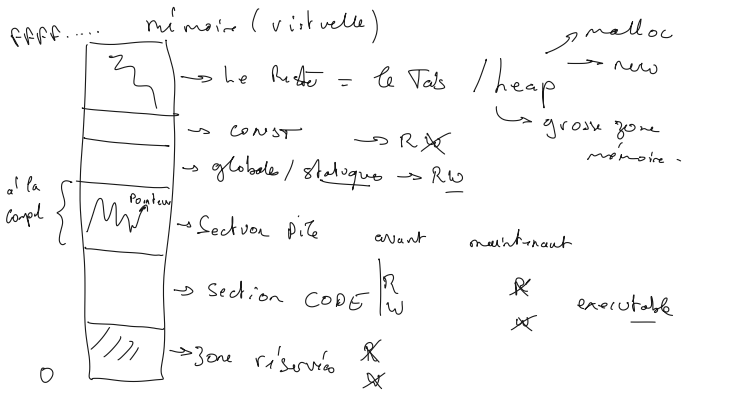

```
void build (bool bTest, ---
{ Vehicule *v;
  if (bTest)
  {
    v = new Voiture();
    v -> moteur = new RadMoteur();
```

Voiture v;

Voiture v2 = v;




```
class Voiture
{
  Voiture (Voiture & v)
  {
  -> définir le clonage via copie

  operator= (Voiture & v)
```



---

cout vs std::cout

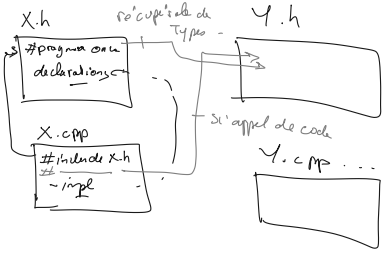
namespace ; <=> "internal" en C# (lié au module)

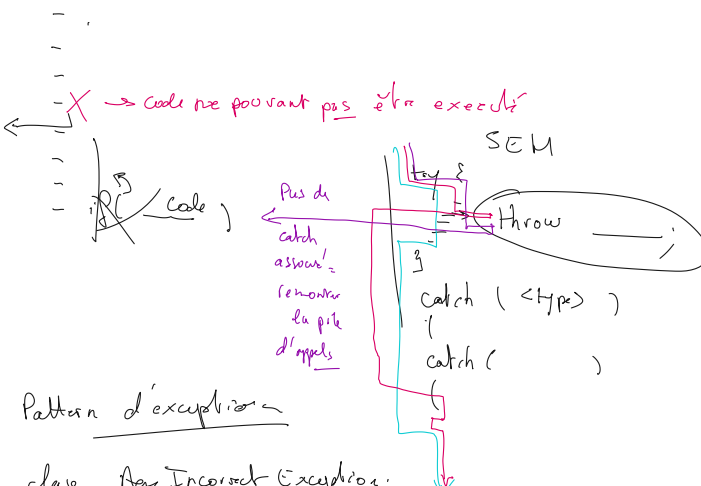
{

...

}

Organisation physique :

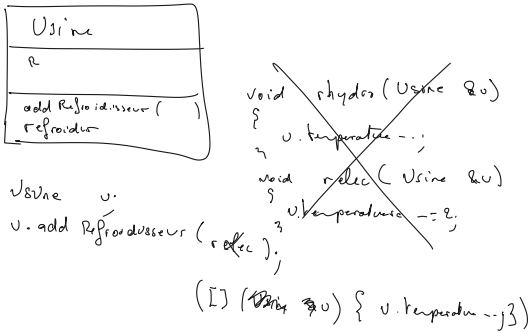




Pattern d'exception

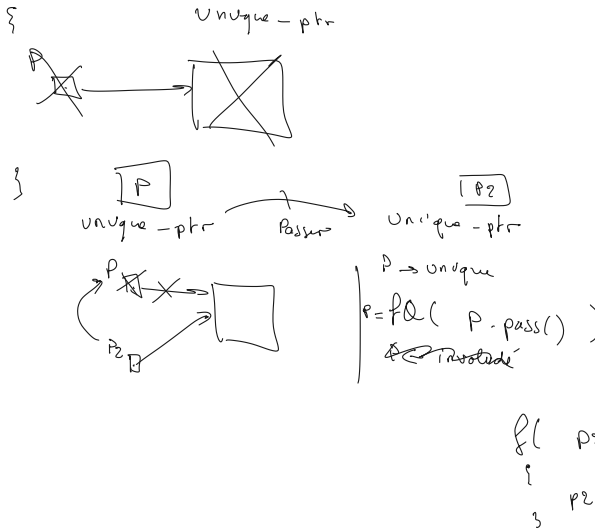
```
class AgeIncorrectException
{


}
```

std::Exception



Exercice :
Créez une collection de doubles avec des valeurs par défaut,
Demandez à effectuer une transformation std::transfom en passant en lambda une fonction qui multiple par 2 cette collection
Puis affichez le contenu via un "foreach" (for modern par itération)
ou la fonction std::foreach

```
for (int number : numbers) {
 std::cout << number << " ";
}
```




```
class Personne
{
public:
    int age;
    Personne(int _age) : age(_age) {};
};

unique_ptr<Personne> ReadPersonne(unique_ptr<Personne> p)
{
    cout << p->age;
    return move(p);
}

int main()
{
    unique_ptr<Personne> p = make_unique<Personne>(50);   // allouer un smart pointer
unique
    p = ReadPersonne(move(p));
    cout << p->age;
}
```