Génie Logiciel

MODULE 3 - CONSTRUCTION DE LA SOLUTION

VENDREDI 20/11

Découpage de ce cours

Cours + TD1

Overview d'un projet informatique

Cours + TD2

Spécification et architecture technique de la solution

Cours + TD3

Construction de la solution

- Modularité dans l'architecture applicative
- Bonnes pratiques d'implémentation
- Tests unitaires

Cours + TD4

Tests de la solution

Quels sont les entrants et les livrables ?

ENTRANTS

- **Quoi ?**
  - Exigences, SFD, STD
  - Architecture technique opérationnelle
  - Règles et normes de codages (méthodes documentées)
- **Qui ?**
  - Référent fonctionnel (Business Analyst)
  - Architecte technique / Référent technique
- **Pourquoi ?**
  - Concevoir le système
  - Développer le système
  - Tester

LIVRABLES

- **Quoi ?**
  - Release prête à être testée
- **Qui ?**
  - Référent technique
  - Développeurs
  - Référent fonctionnel (rôle de support)

01

Modularité dans l'architecture applicative

Architecture technique
2 points de vue

2 points de vue architecturaux essentiels

Architecture applicative

On s'intéresse aux **applications**

- Leur organisation interne
- Leurs interactions

Architecture technique

On s'intéresse aux **composantes techniques accueillant les applications**

- Leurs interactions

10 min / groupe

Comment définiriez-vous un module ? Un système modulaire ?

- Quels seraient les critères permettant de vérifier qu'un système est modulaire ?
- Proposez des règles pour construire un système modulaire ?

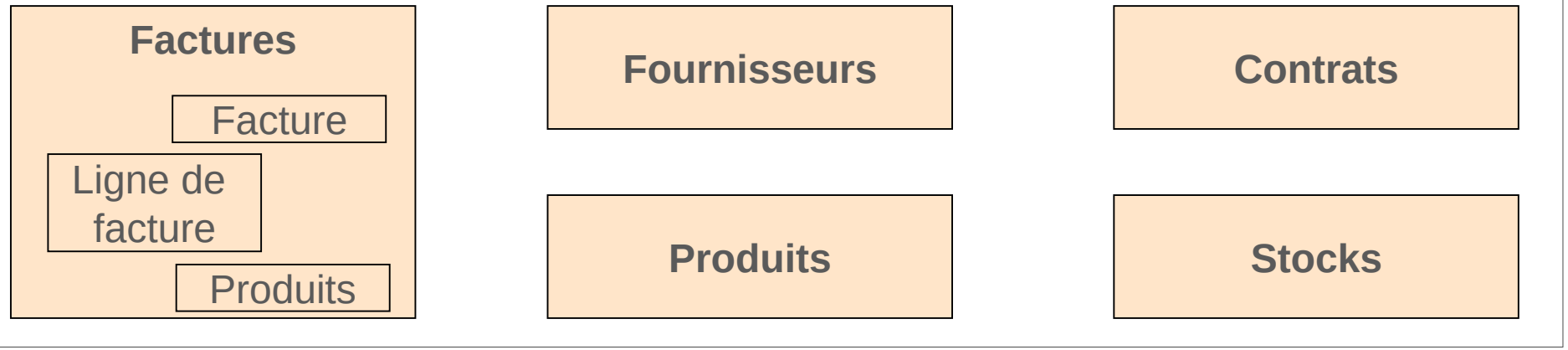
Quels sont les avantages/inconvénients d'un système modulaire ?

Système modulaire

Exemple : le backoffice d'un grand magasin

- **Sur une application :**
  - Découpage des fonctions, méthodes, traitements
  - Un module = un corps + une interface
  - Modules de tailles différentes
  - Développement du code de chaque module par des équipe indépendantes
  - Tests indépendants
- **Dans votre contexte (plus restreint) :**
  - 1 module = une classe Java

Exemple : backoffice magasin d'un grand groupe

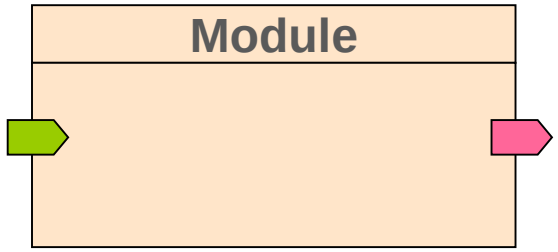


Module
Définition

Un **module** est une unité **autonome** qui procure des **services** au travers **d'interfaces** spécifiées par un **contrat**.

Un composant a un comportement externe : **INTERFACE**

Un composant a une réalisation interne : **CORPS**

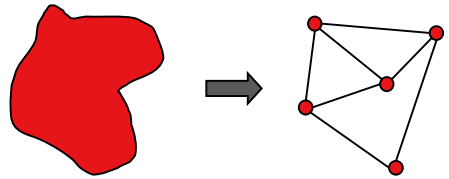


Un module peut être vu comme une boite **noire**

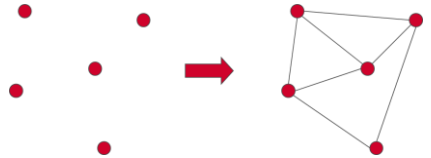
Un module peut être vu comme une boite **blanche**

Modularité
Définition

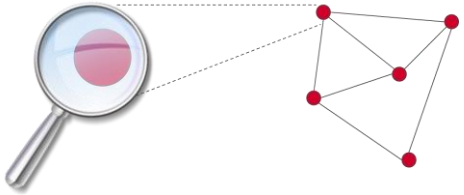
- **La notion de modularité peut se définir par les notions :**
  - Décomposable (approche top/down) : Décomposer un problème complexe en plusieurs autres plus simples et autonomes



  - Composable (approche bottom/up) : Recomposer un système en connectant des composants indépendants



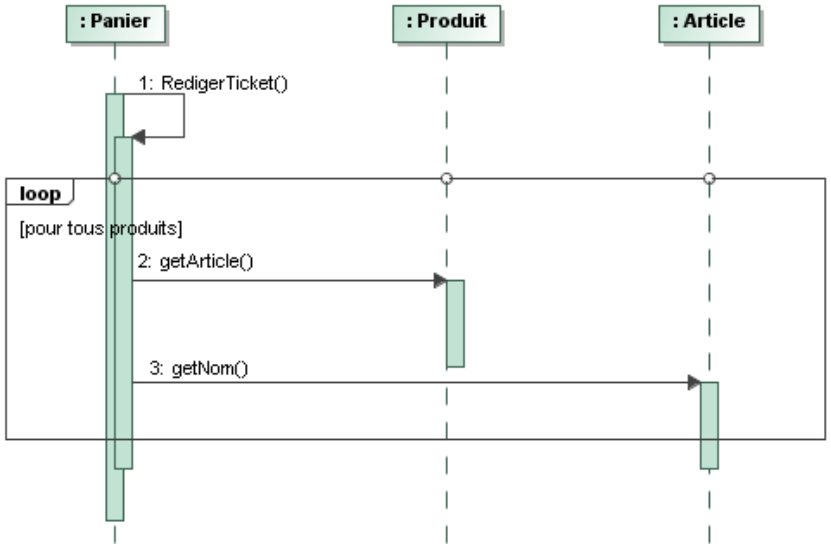
  - Compréhensible : chaque composant est compréhensible sans avoir à connaître les autres (ou très peu)



  - Basé sur 3 principes :
    - Ouvert/fermé : accepte les évolutions mais en gardant les accès stables
    - Faible couplage
    - Forte cohésion

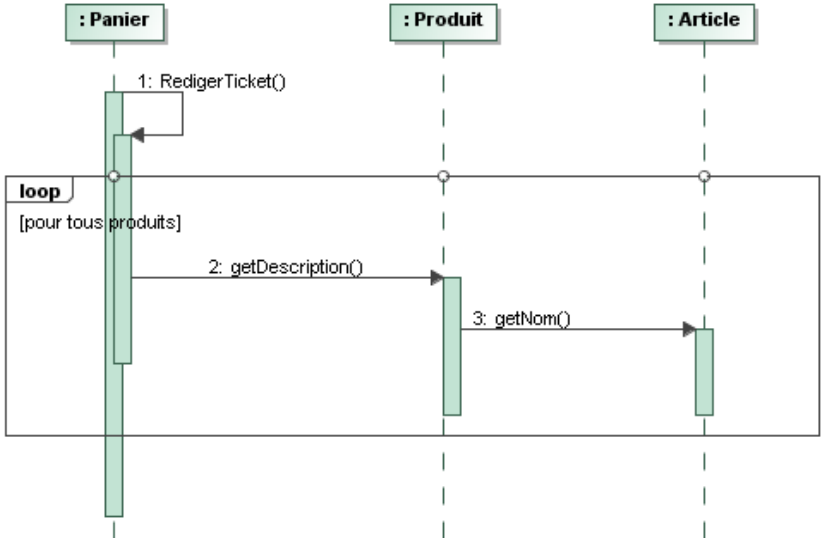
Principe général n°1
Faible couplage

- **Fort couplage vs Faible couplage**



Solution introduisant un couplage inutile entre Panier et Article

monpanier.getArticle().getNom()

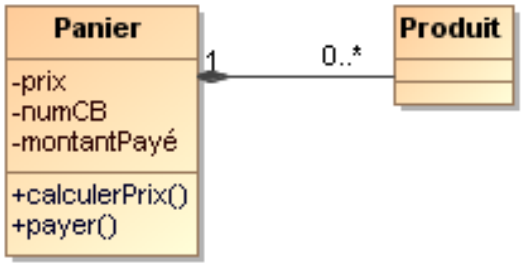


Solution s'appuyant sur le couplage intrinsèque du modèle

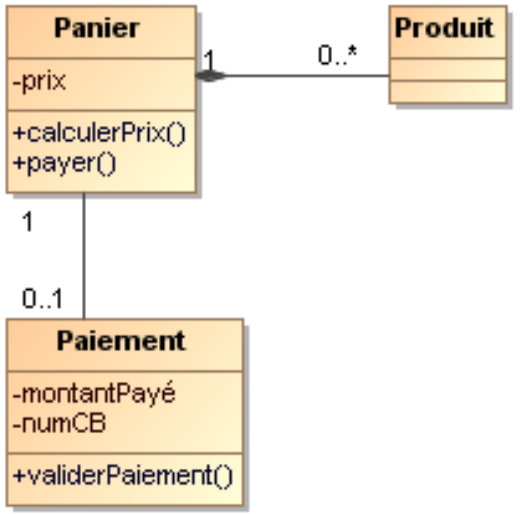
monpanier.getDescription()

Principe général n°2
Forte cohésion

Faible cohésion vs Forte cohésion



Panier mélange 2 considérations très différentes : constituer son panier et payer celui-ci



Panier conserve la constitution de celui-ci car c'est sa structure intrinsèque. Par contre on externalise dans Paiement les considérations financières

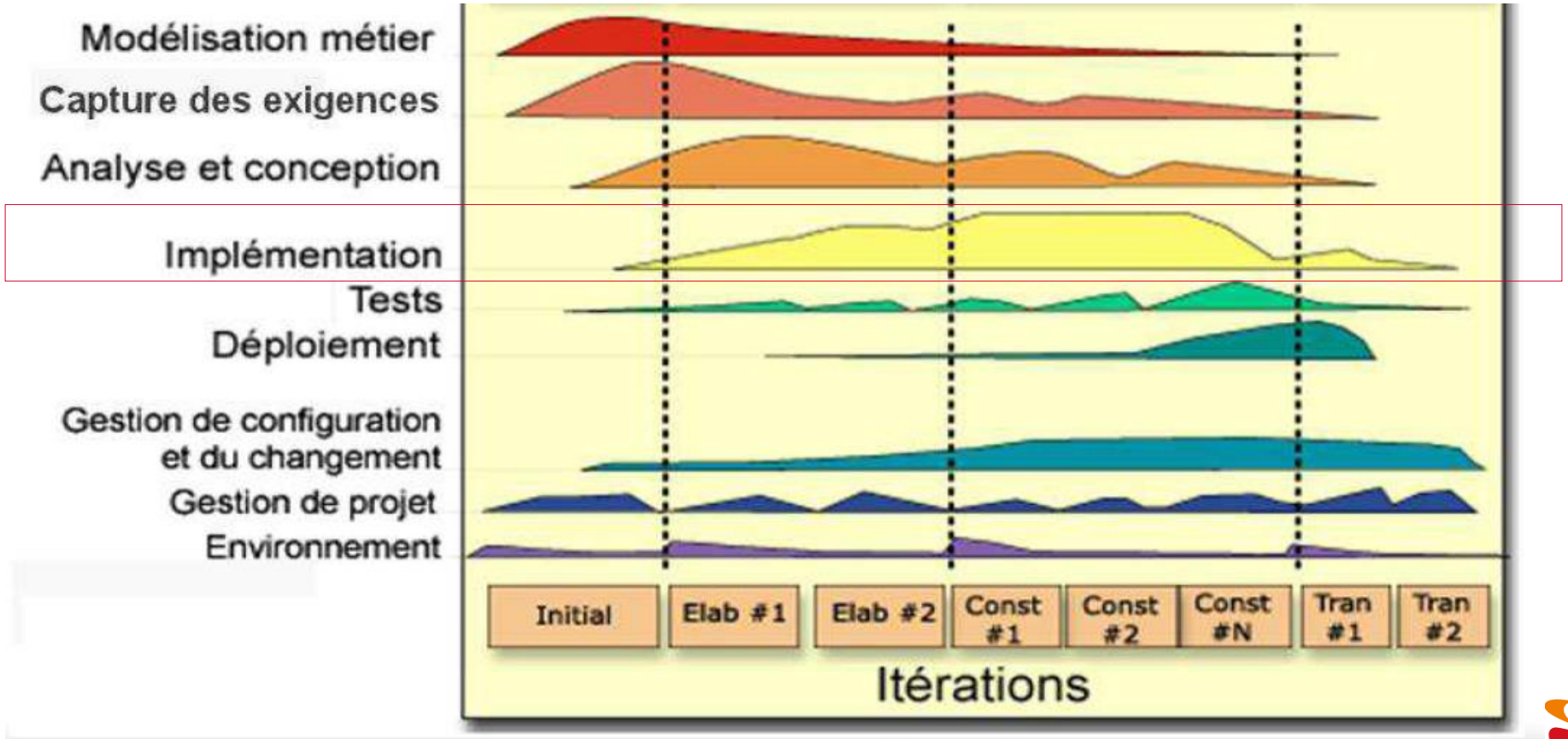
02

Bonnes pratiques d’implémentation

Maintenant que vous avez fait l'architecture technique et applicative.
Implémentez !

Quelles sont les outils de développement que vous utilisez ?
Quelles sont les bonnes pratiques de développement que vous appliquez ?

10 min / groupe



Le mode « debug » pour le back

Bouton pour avancer ligne par ligne / inspecter le contenu d'une autre méthode

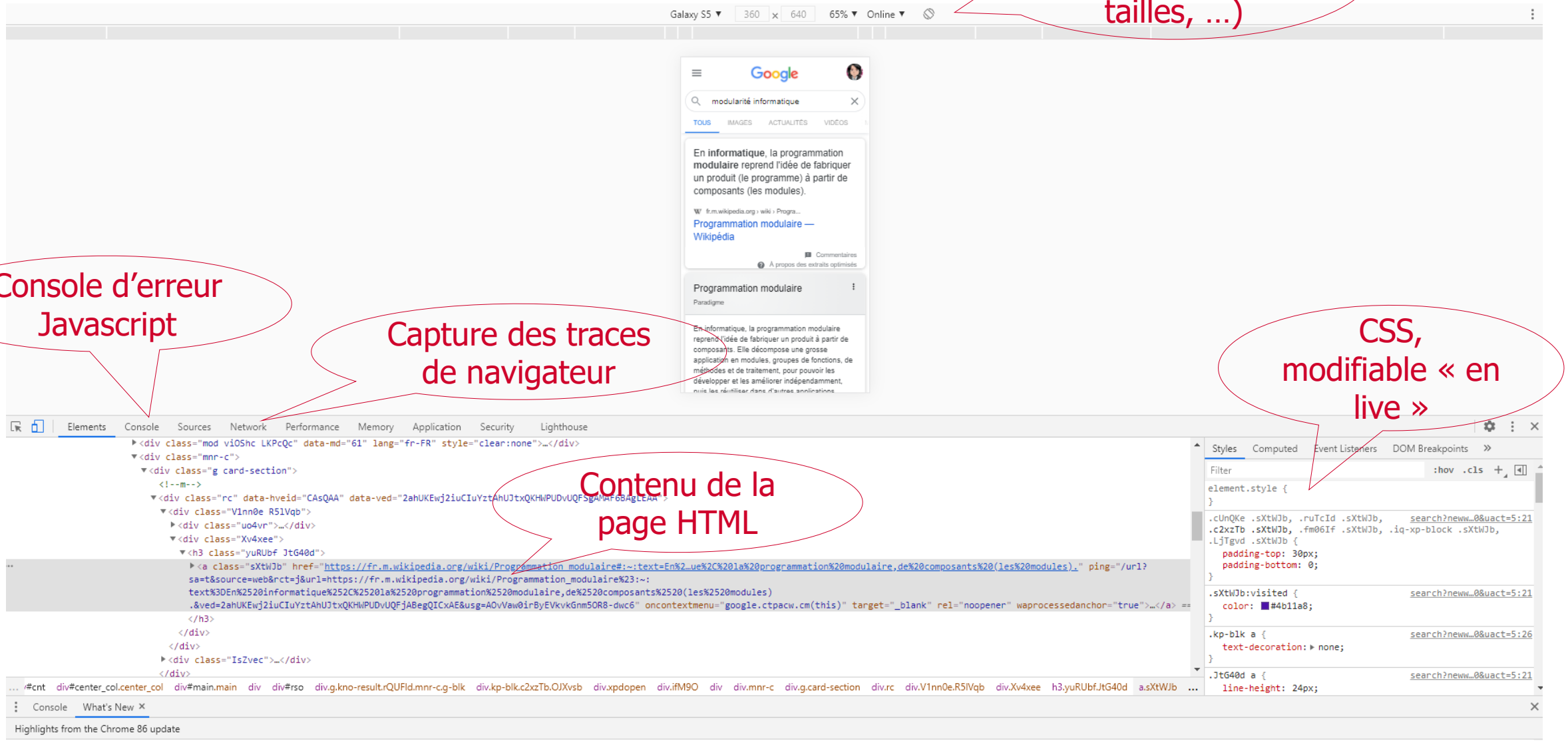
Pile des méthodes

Ligne de code en cours d'exécution

Les variables accessibles avec leur contenu

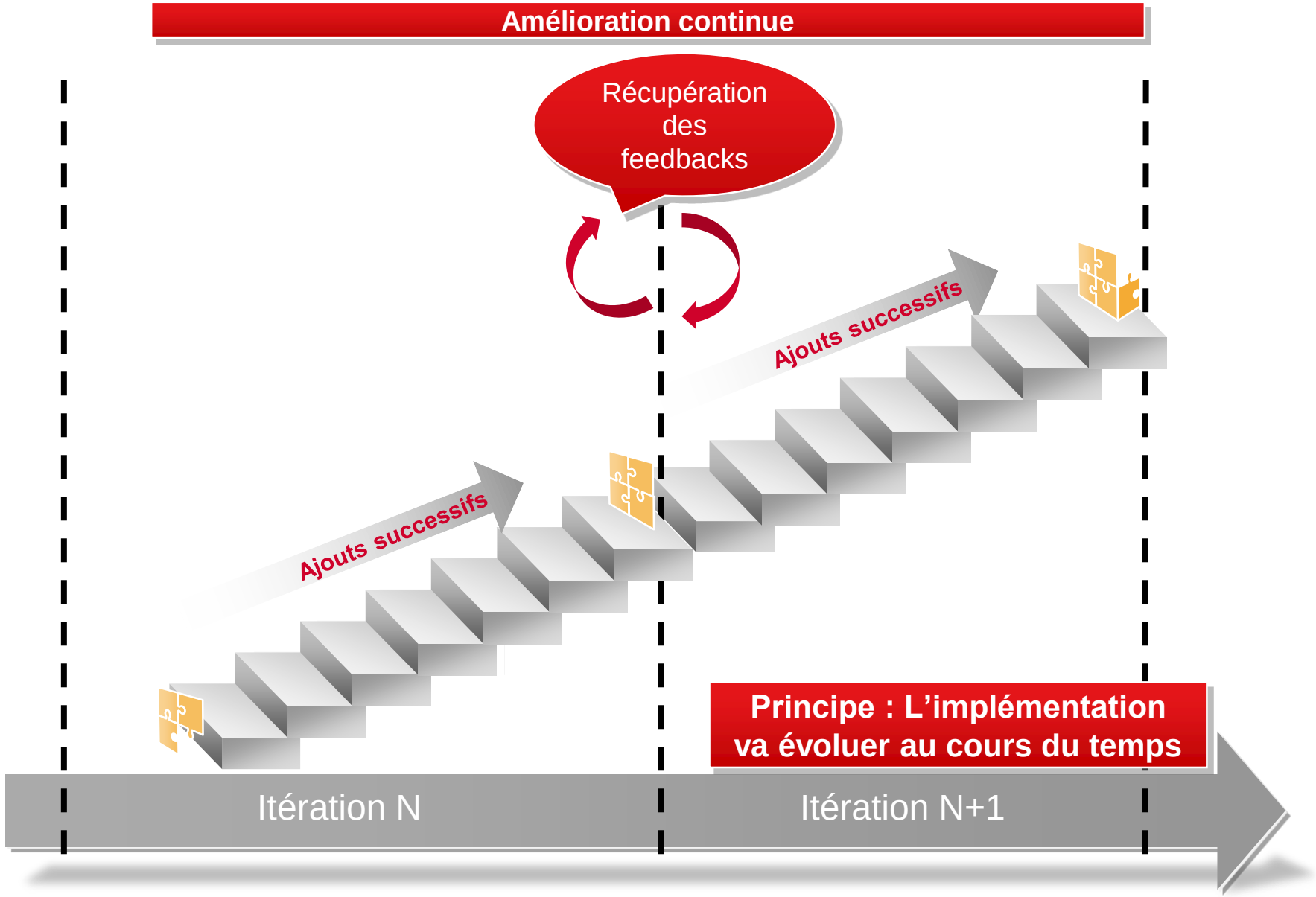
L'affichage de la variable sélectionnée

Le « F12 » pour le front



Implémentation itérative

Vue générale



Bonnes pratiques

Maîtrise fonctionnelle et qualité de code

- Maitriser le **fonctionnel** :
  - S'assurer de la compréhension des exigences
  - Obtenir des précisions sur la spécification qui peut être insuffisamment précise
  - Comprendre le sens global de la solution

- Maitriser l'**architecture technique et applicative** de la solution et s'assurer qu'elles soient bien respectées lors des revues de code

- S'assurer de la **qualité** du code :
  - **Tracer les exigences** (décrites dans les SFD) dans le code : pour détecter les impacts dans le code en cas modification de l'exigence
  - Mettre des traces applicatives *(cf slide suivante)*
  - Prendre en compte la sécurité *(cf slide suivante)*
  - Partager les connaissances grâce a des revues de code (avec GIT par exemple)

Maitrise de la qualité de code

Gestion des traces applicatives

- Il est nécessaire de tracer les informations importantes sur le déroulement d'une application
- Le Build, par paramétrage de construction et/ou d'exécution, pourra ne pas déclencher certains niveaux
- Utiliser les mécanismes hiérarchisant les traces (cf. tableau ci-contre) fournis par l'architecture technique
  - Ex : Log4J en Java
- Trace et Exception : Ne pas tracer 2 fois la même chose
- En production, en général les traces de type ERROR ou FATAL sont supervisées

Niveau	Description
TRACE	Pour suivre le cheminement dans l'application
DEBUG	Pour extraire de l'information complète permettant la mise au point
INFO	Information à conserver lors du fonctionnement normal de l'application
WARN	Pour informer de situations anormales mais ne mettant pas en péril le fonctionnement
ERROR	Pour informer d'un dysfonctionnement localisé
FATAL	Pour informer d'un dysfonctionnement global

Maitrise de la qualité de code

Gestion des traces applicatives - Exemple


```
Private Log LOGGER = new Log();
```

```
public ContenuFichier charger(final String cheminRepertoire, final String nomFichier) {
        if (!StringUtils.startsWith(nomFichier, « debutDuNomDeMonFichierACharger")) {
            return null;
        }
        final File fichier = new File(cheminRepertoire + "/" + nomFichier);
        Workbook book = null;
        try {
            book = WorkbookFactory.create(new PushbackInputStream(new FileInputStream(fichier)));
            final Sheet sheet = book.getSheet(“NomDeMonOngletACharger");
            return new ContenuFichier(recupererNom(sheet), recupererDescription(sheet), recupererInput(sheet), recupererOutput(sheet)));
        } catch (final Exception e) {
        } finally {
            try {
                if (book != null) {
                    book.close();
                }
            } catch (final IOException e) {
          }
        }
        return null;
    }
```

Exemple d’utilisation du logger :

LOGGER.debug(« Le texte à logguer en niveau debug »)

LOGGER.info(« Le texte à logguer en niveau info »)

LOGGER.warn(« Le texte à logguer en niveau warn »)

LOGGER.error(« Le texte à logguer en niveau error »)

Maitrise de la qualité de code

Gestion des traces applicatives – Exemple

```
Private Log LOGGER = new Log();


public ContenuFichier charger(final String cheminRepertoire, final String nomFichier) {
        LOGGER.debug(« Debut fonction charger à : » + new Date() + « chemin : » + cheminRepertoire + « - nom : » + nomFichier) ;
        if (!StringUtils.startsWith(nomFichier, « debutDuNomDeMonFichierACharger")) {
            LOGGER.warn(« Le fichier à charger ne commence pas par le bon nom »);
            return null;
        }
        final File fichier = new File(cheminRepertoire + "/" + nomFichier);
        LOGGER.debug(« Le fichier à charger » + fichier);
        Workbook book = null;
        try {
            book = WorkbookFactory.create(new PushbackInputStream(new FileInputStream(fichier)));
            final Sheet sheet = book.getSheet(“NomDeMonOngletACharger");
            return new ContenuFichier(recupererNom(sheet), recupererDescription(sheet), recupererInput(sheet), recupererOutput(sheet)));
        } catch (final Exception e) {
            LOGGER.error(« Exception lors du chargement du fichier » + e.printstacktrace());
        } finally {
            try {
                if (book != null) {
                    book.close();
                }
            } catch (final IOException e) {
                LOGGER.error(« Exception lors de la fermeture du fichier » + e.printstacktrace());
            }
        }
        return null;
    }
```

Maitrise de la qualité de code

Code smells, refactoring, rework

- Identifier les « odeurs de code » (« ***code smells*** »)
  - Code dupliqué
  - Code mort
  - Méthodes trop longues
  - Volume exagéré de code
  - Couplage fort
  - Nommage non représentatif
  - Algorithme peu performant
  - Commentaires trop longs
  - NullPointer (en Java)

➔ Les IDE proposent des fonctionnalités permettant d'identifier ces « code smells »

- Techniques de **refactoring**
  - Factoriser le code dupliqué dans une méthode unique
  - Découper une méthode trop longue en plusieurs méthodes plus petites
  - Décomposer un élément d'implémentation trop volumineux en plusieurs éléments d'implémentation
  - Donner des noms significatifs à tous les niveaux
  - Chaque technologie a ses propres techniques de refactoring

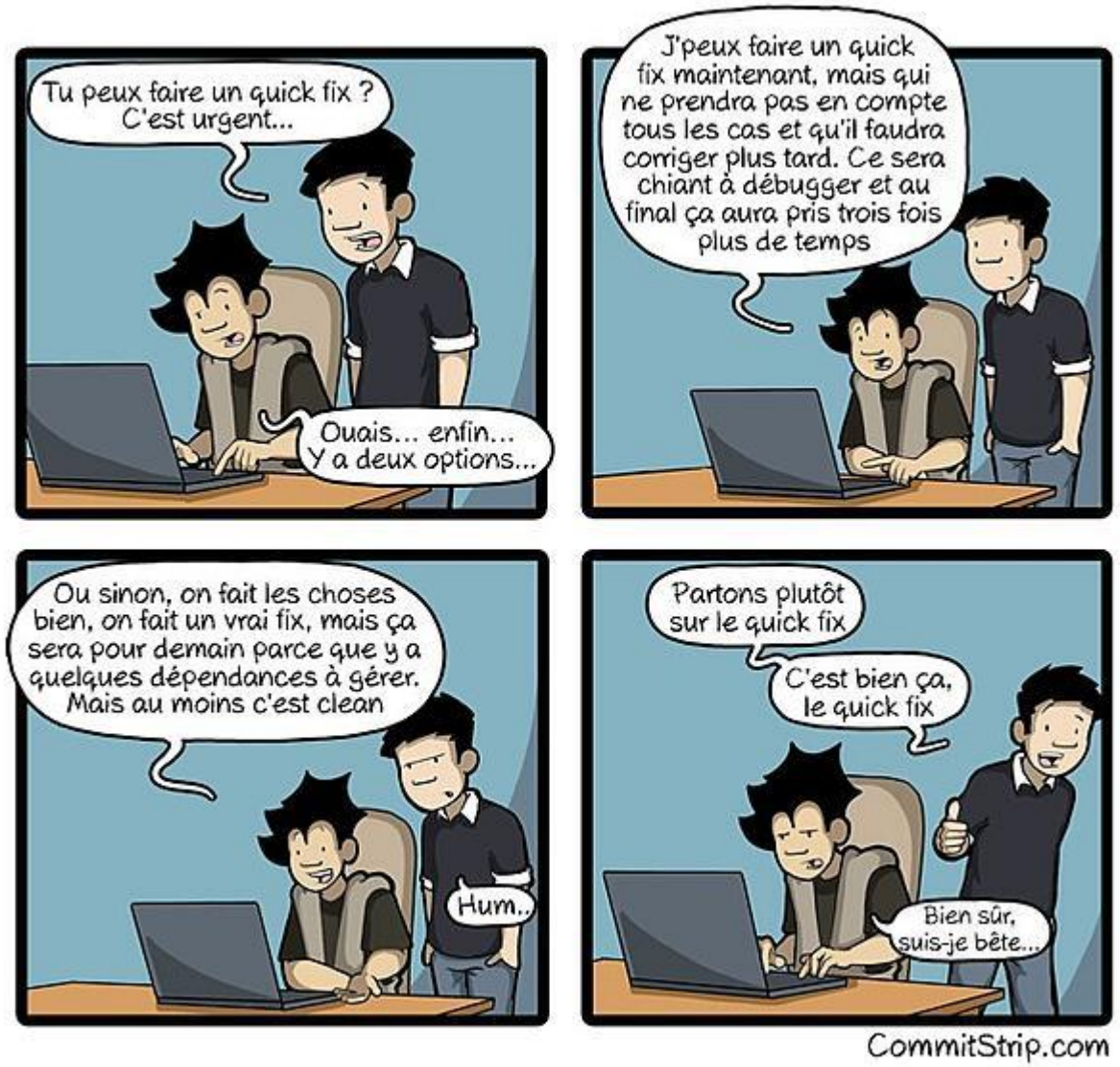
- **Rework**
  - Eviter le rework en utilisant les bons design pattern et en prenant de la hauteur de vue sur les fonctionnalités qui vont arriver ensuite

Maitrise de la qualité de code

Sécurité applicative, robustesse

- Suivre la **dette technique** (le résultat de ce qu'on n'a pas voulu/pu faire correctement…) et la corriger au fil de l'eau

- Prévenir l'introduction de **failles de sécurité** dans le code (ex : OWASP)
- Vérifier les vulnérabilités des frameworks/bouts de code utilisés
- Faire les montées de version des framework utilisés
- Vérifier les performances de l'application (en fonction de son usage final)



Quel type de failles applicatives connaissez-vous ?

Maitrise de la qualité de code

Les 10 principales failles de l'OWASP

1 - Injection	Injecter du code arbitraire (souvent du SQL) pour effectuer des actions qui seront interprétées par l'application. ➔ Utiliser des fonctions sécurisées, valider les entrées utilisateurs.
2 - Piratage de session	Lorsque le système d'authentification a été contourné, par exemple en utilisant la technique de force brute. ➔ Forcer les utilisateurs à utiliser des mots de passe forts, utiliser des cookies de session.
3 - Exposition de données sensibles	Fuite de données, par exemple. ➔ Utiliser des solutions de chiffrement pour sécuriser les données en transit et celles stockées sur l'application.
4 - Entités externes XML (XXE)	Peuvent être utilisées pour accéder à des données internes de l'application normalement non accessibles. ➔ Il est possible de désactiver les entités externes.
5 - Contournement des contrôles d'accès	➔ S'assurer que toutes les pages de l'application ont un contrôle d'authentification
6 - Mauvaises configurations de sécurité	➔ Garder à jour les composants de votre application pour éviter qu'une vulnérabilité ne soit exploitée.
7 - Scripts XSS (cross-site scripting)	Permettent à un attaquant d'injecter du code JavaScript. ➔ Utiliser la validation et la transformation des entrées utilisateurs pour les éviter.
8 - Desérialisation non sécurisée	Peut permettre de mener une attaque d'élévation de privilège, de replay ou encore d'injection. ➔ Il est possible d'implémenter des contrôles sur l'état du code.
9 - Utilisation de composants contenant des vulnérabilités	Principalement lorsque ces failles sont connues. ➔ Maintenir à jour et identifier les éléments de l'application. Ne pas exposer les versions des composants.
10 - Manque de monitoring et de surveillance	➔ Le monitoring et la surveillance permettront de détecter une intrusion ou un comportement suspicieux au plus tôt. Vérifiez régulièrement vos logs et mettez en place des reportings.

Maitrise de la qualité de code

Les cyber attaques en temps réel - https://cybermap.kaspersky.com/fr

03

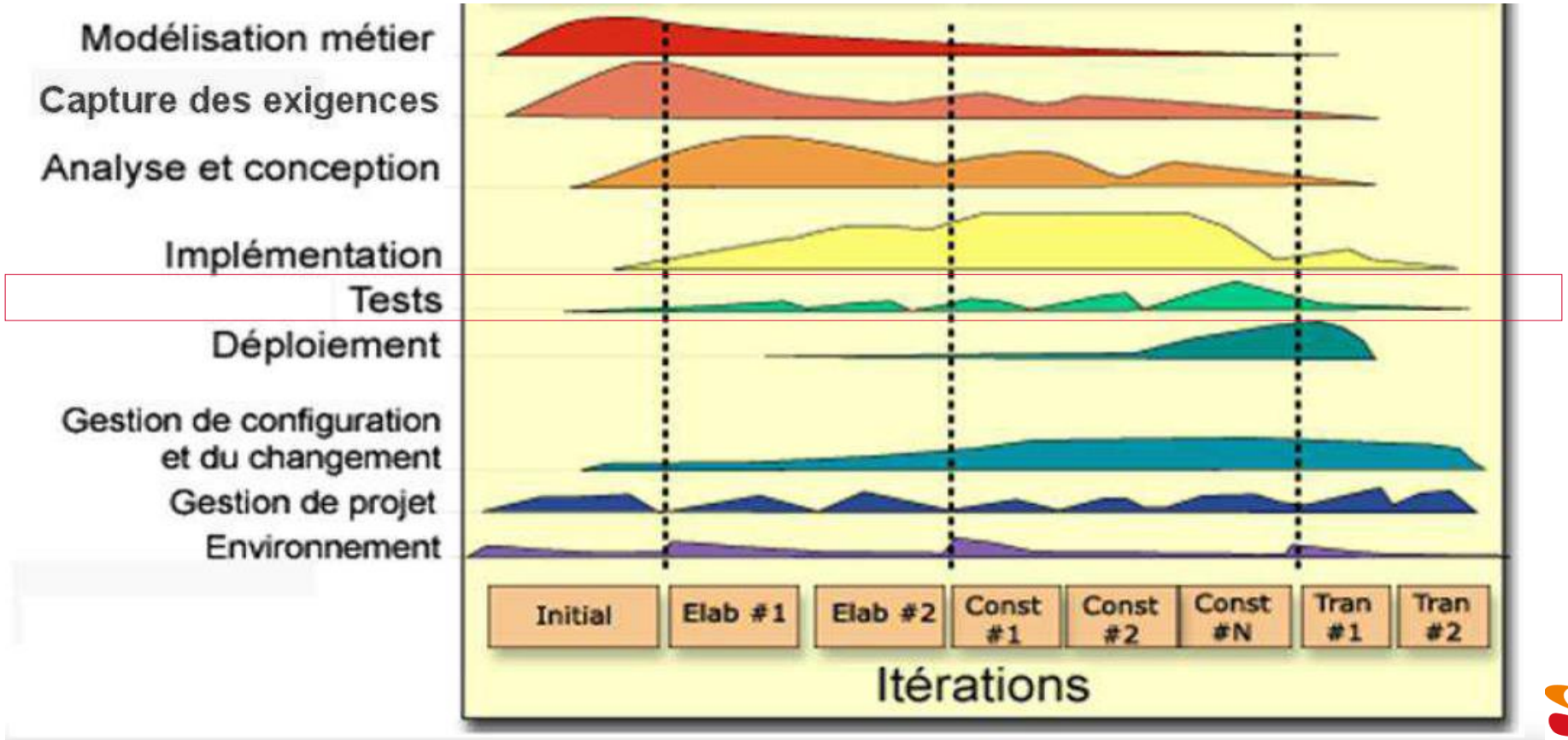
Tests unitaires

Selon vous :

- Que signifie « unitaire » dans « Test Unitaire » ?
- Quels sont les bénéfices de tests « codés » comparativement à des tests « manuels » ?
- Faut-il tout tester ?
- A quel moment écrire un test unitaire ?

Qui aime tester son développement ?

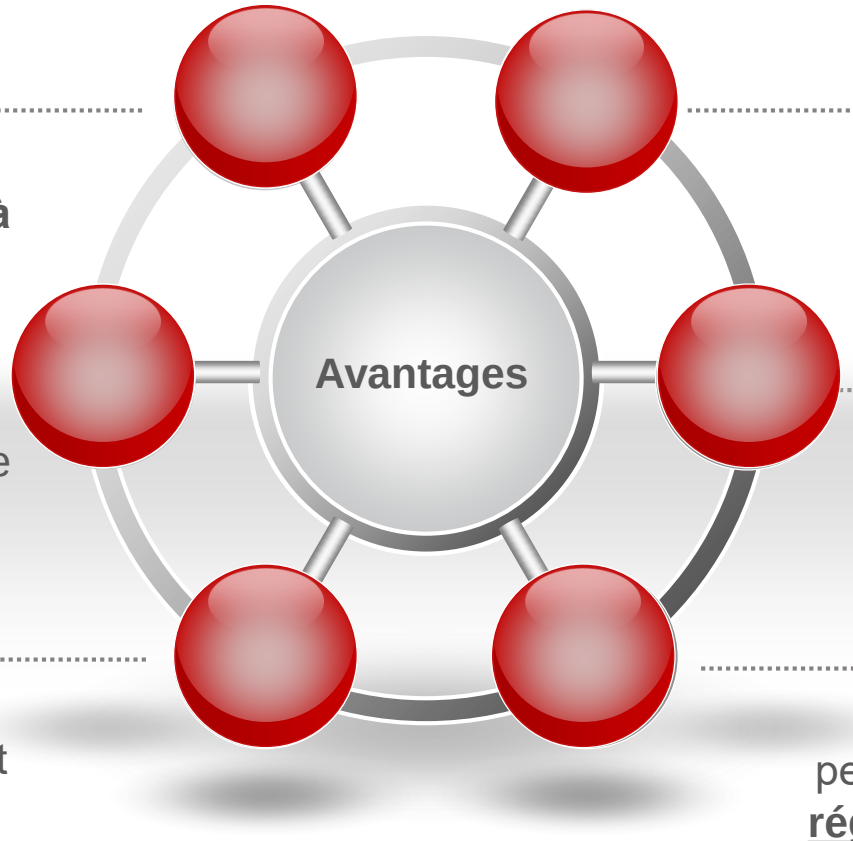
10 min / groupe



Tests unitaire

Automatisation des tests

- Le concept : développer du code qui teste l'implémentation
- Un investissement qui sera vite rentabilisé



Un test automatisé peut être **rejoué à l'infini et à tout moment**

Son exécution est **plus rapide** qu'un test manuel

Son implémentation joue un rôle de **documentation**

Le **résultat** de son exécution est automatiquement **tracé**

Le test est **partagé** avec toute l'équipe (actuelle et future)

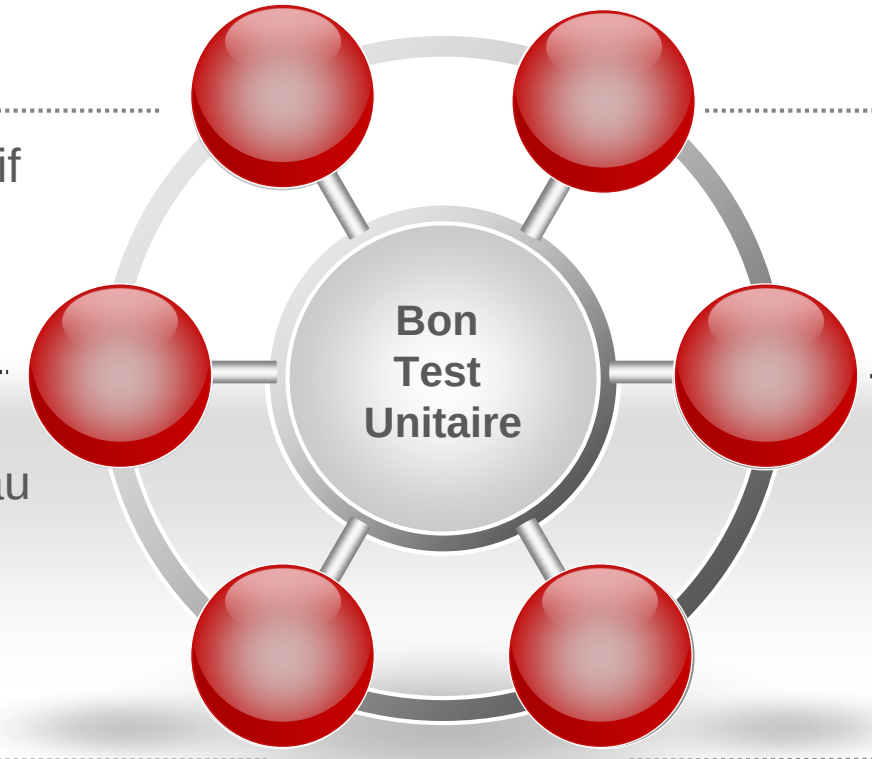
Un test automatisé permettra d'assurer la **non régression** à moindre coût

Tests unitaire

Qu'est-ce qu'un bon test unitaire ?

Un test a pour objectif principal d'**identifier les défauts**

Un test est **indépendant des autres tests**



Un test est **utile au développeur** (aide au design, moins de stress lors des changements)

Le test est **compréhensible et facile à maintenir**

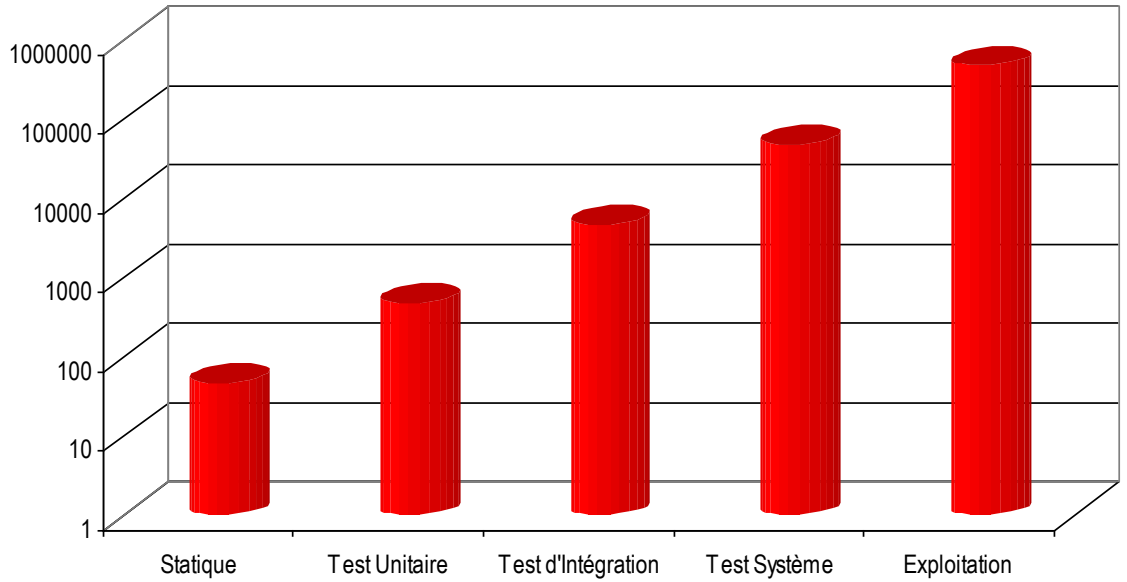
Un test permet de **mesurer objectivement la conformité**

Un test **cible** surtout **ce qui est risqué**

Un test unitaire est intelligent !

Tests unitaire
Coûts

- Le coût de correction d'une anomalie augmente avec le temps



Coût (en €) du diagnostic et de la correction d'une anomalie par rapport aux niveaux de test
(source : UK Orange services)

L'impasse sur les tests unitaire n'est jamais gagnante

La définition du « DONE »

A quel moment mon développement peut faire partie d'une release ?

- Le « DONE » doit être défini **avant** de démarrer les développements, il peut être propre à chaque projet / type de projet

- A minima (en entreprise) :
  - Le code correspond fonctionnellement à la user story / exigence
  - Le code doit répondre aux exigences d'architecture définie
  - Le code doit avoir été revu techniquement par un autre membre de l'équipe (merge GIT)
  - Il ne doit pas y avoir d'erreur de compilation ou d'erreur bloquante
  - Le code doit avoir une couverture de TU correcte, les TU doivent être passants

- Si toutes les conditions sont réunions alors le RAF (reste à faire) peut être mis à 0j et la tâche être clôturée et peut partir en phase de test suivante (intégration, fonctionnelle)

Les mots clés du cours