GENIE LOGICIEL PRAGMATIQUE

Fabrice RAZAFINDRAIBE
Développeur logiciel

Université de Fianarantsoa
Faculté des sciences
Parcours : Mathématiques appliquées
Mention : Mathématiques et informatique

Génie Logiciel: *c'est quoi?*

Définition Wikipédia:

> « *Le terme **génie logiciel** désigne l'ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel, au-delà de la seule activité de programmation.* »

L'art et la manière de créer un logiciel.

Objectifs du cours

- Présenter un **aperçu de l'état de l'art** en matière de génie logiciel.
- Exposer les **principaux courants de pensées** en matière de développement logiciel.
- Proposer un **ensemble de pratiques pragmatiques** qui permettent de survivre à un projet de développement de logiciel.
- Vous donner envie de devenir développeur de logiciels!

Thèmes du cours

Les principaux thèmes abordés:

- Pourquoi un cours de génie logiciel?
- C’est quoi un bon logiciel bien fait?
- Processus et activités
- Intégration continue
- Gestion de configuration et de faits techniques
- Travail d’équipe
- Gestion de projet

Le génie logiciel est un far-west

De **nombreuses manières** d'envisager le génie logiciel:

- Les hackers anarchistes et solitaires,
- La communauté open-source,
- Les praticiens de l'agilité,
- Les « modélisateurs »,
- Les générateurs,
- Les héros, les artistes, les pragmatiques …
- …

« Il n'y a pas une méthode unique pour étudier les choses. » Aristote.

Le jargon du métier

Spécification, exigences et cas d'utilisation, recueil de besoins	Tests de validation, de recette, d'acceptation	Natif et croisé
Conception, architecture, plan	Boite blanche et boite noire	Indicateurs, métriques, avancement
Codage, implémentation	Processus, cycle de vie	Documentation
Tests	Gestion de configuration, de version	Standards, guides, « best practices »
Intégration	Gestion des faits techniques	Outils, ateliers, IDE
Validation, recette, acceptation	Cycle en V, cascade, développement prédictif	Model-Driven, test-driven, use-case-driven, behaviour-driven, agile, eXtreme-Programming
Tests unitaires	Cycle itératif et incrémental	Orienté-objet
Tests d'intégration	Traçabilité et couverture	…

Pourquoi?

Pourquoi un cours de génie logiciel?
Pourquoi a-ton besoin de génie logiciel?

Pourquoi participer? *1/6*

Objectifs « **court terme** » des étudiants :

- Prendre du recul sur des expériences vécues en
  - stages,
  - mini-projets,
  - travaux pratiques.

  Formaliser votre expérience du développement logiciel.

- Note à l'**examen**.

Pourquoi participer? *2/6*

Objectifs « **moyen terme** » des étudiants :

- Faire des **choix**:
  - Chercher un emploi dans l'informatique, dans le logiciel?
  - Développer?

Pourquoi participer? *3/6*

Objectifs « **moyen terme** » des étudiants :

- Pour ceux qui choisissent de ne pas développer:
  - Comprendre comment sont construits les logiciels qu'ils vont utiliser;
  - Être en mesure d'exprimer des besoins et de suivre un développement de logiciel.

Pourquoi participer? *4/6*

Objectifs « **moyen terme** » des étudiants :

- Pour ceux qui choisissent le **hw** ou le **système**:
  - Comprendre les « softeux » avec qui ils **vont** travailler.
  - S'inspirer de **pratiques** de « softeux » qui s'adaptent bien à d'autres domaines.
  - Selon les organisations, vous ferez peut-être **aussi** du sw.

Pourquoi participer? *5/6*

Objectifs « **moyen terme** » des étudiants :

- Pour ceux qui choisissent le développement sw:
  - Établir des **contacts** pour stages et emplois.
  - Identifier le **vocabulaire** qui séduira les employeurs.
  - Renforcer une **culture générale** qui permettra de s'intégrer en douceur dans une équipe de professionnels.

Pourquoi participer? *6/6*

Ce cours arrive t-il trop tôt ou trop tard dans le cursus?

3 manières d'insérer ce cours dans un cursus:

- **Tôt, théorique et antérieur à la pratique**
  - Risque d'être trop abstrait, sans résonnance concrète.
- **En continue, en alternance avec de la pratique**
  - Probablement l'optimum, mais très ambitieux à mettre en place.
- **Tard, après la pratique**
  - Permet de formaliser les impressions vécues. Permet de réagir sur la base de l'expérience.

Pourquoi le génie logiciel? *1/8*

Pour répondre à cette question, il faut d'abord **mesurer l'impact d'un développement logiciel** pour une entreprise:

- Que représente le **poids financier** d'un projet de développement logiciel?
- Est-ce qu'un projet de développement logiciel est une **aventure tranquille**?

Pourquoi le génie logiciel? *2/8*

Ordre de grandeur dans le meilleur des mondes:

- 1 H/An = 1350h,
- 1h = ~50€,
- Productivité = ~2à5 lignes/h

Dimension	Nb lignes	Heures	Cout	Hommes/An
Petit	30_000	6_000	300K€	4
Gros	500_000	100_000	5M€	74

Pour comparer:

- 1 voiture ~15K€
- 1 maison ~150K€

Pourquoi le génie logiciel? *3/8*

Nous avons vu que:

- 1h = ~50€,
- Productivité = ~2 à 5 lignes/h

Donc, le code suivant:

```
ActivityMap total = new ActivityMap();
    Iterator iter = this.workMonths_.iterator();
    while (iter.hasNext()) {
        WorkPeriod month = (WorkPeriod) iter.next();
        total.add(month.getActivities());
    }
```

met **1h** à traverser le processus de développement et coûte **50€**!

Pourquoi le génie logiciel? *4/8*

Selon « The Standish Group Report», en 1999

Project Duration, Team Size Affect Project Success

Project Size	People	Time (mos.)	Success Rate
Less than $750K	6	6	55%
$750K to $1.5M	12	9	33%
$1.5M to $3M	25	12	25%
$3M to $6M	40	18	15%
$6M to $10M	+250	+24	8%
Over $10M	+500	+36	0%

Pourquoi le génie logiciel? *5/8*

Selon l'étude « The Standish Group Report» en 2000

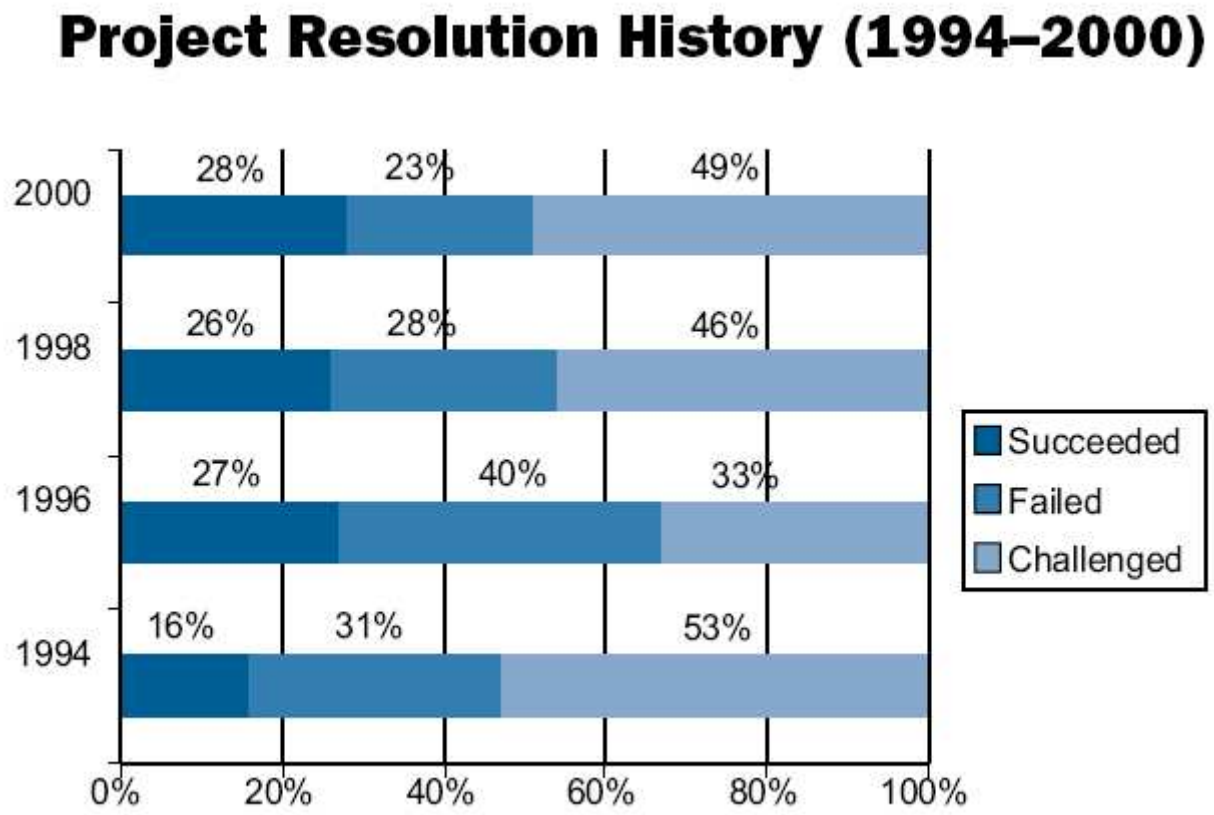
The CHAOS Ten

Executive Support	18
User Involvement	16
Experienced Project Manager	14
Clear Business Objectives	12
Minimized Scope	10
Standard Software Infrastructure	8
Firm Basic Requirements	6
Formal Methodology	6
Reliable Estimates	5
Other	5

Each factor has been weighted according to its influence on a project's success. The more points, the lower the project risk.

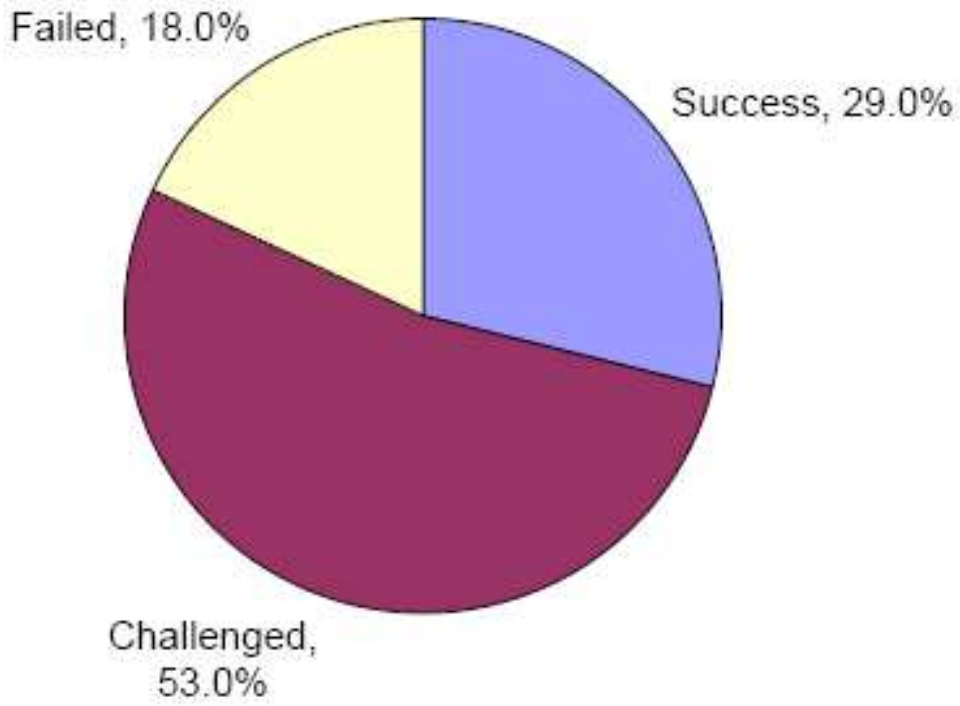
Pourquoi le génie logiciel? *6/8*

Selon l'étude « The Standish Group Report»



Pourquoi le génie logiciel? *7/8*

CHAOS 2004 Survey of Software Projects, 2004



Pourquoi le génie logiciel? *6/6*

En conclusion:

Le développement d'un logiciel est une **entreprise risquée** car:

- Cela coûte **cher** *(et plus que prévu)*
- Cela dure **longtemps** *(et plus que prévu)*
- Cela n'est même **pas sûr d'aboutir**! *(pas sûr d'obtenir ce qui a été demandé)*

Réussite/échec *1/3*

Selon l'étude « The Standish Group Report» (1995)

Les principales raisons de **réussite** d'un projet sont:

- L'implication des **utilisateurs**
- Soutient de la hiérarchie
- Besoins clairs du **client**

Réussite/échec *2/3*

Selon l'étude « The Standish Group Report» (1995)

Les principales **d'échec** d'un projet sont:

1. Manque d'informations des utilisateurs
2. Besoin client incomplet
3. Besoin client changeant

Réussite/échec *3/3*

Conclusions:

- Les **utilisateurs** d'un logiciel doivent être **impliqués** dans son développement
- Les **besoins** du client sont **imprécis et changeants**:
  - Faut-il faire un effort pour préciser et figer le besoin du client en début de projet?
  - Faut-il développer de manière à être tolérant aux imprécisions et aux changements de besoins?

2 pratiques différentes du génie logiciel **s'opposent sur la manière de traiter ces 2 problèmes**.

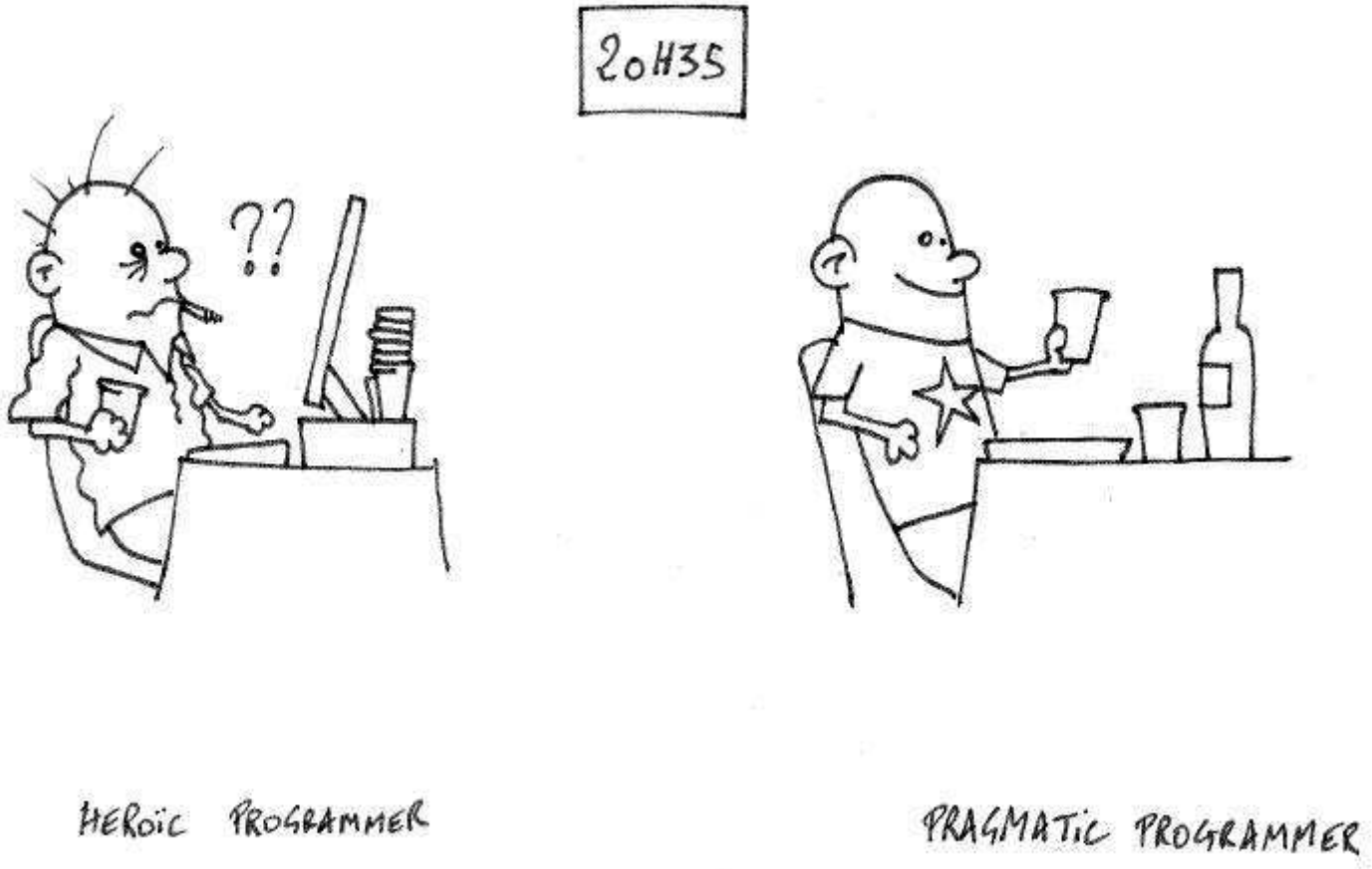
Quel est l'enjeu du génie logiciel?

- Essayer de maitriser la **complexité** et le **coût** d'un développement de logiciel.

- Augmenter la probabilité de **réussite** d'un projet de développement de logiciel.

- **Bien développer** le **bon logiciel**.
  - Bien développer: maitriser coût/délai/qualité.
  - Bon logiciel: celui attendu par les utilisateurs.

Avec ou sans génie logiciel?



Est-ce applicable?

- **Mythe**: *« Nous n'avons pas le temps ni le budget pour travailler proprement et de manière disciplinée. »*

- **Non!** Travailler de manière propre et disciplinée fait gagner du temps et de l'argent.

Est-ce toujours applicable?

Non, pour les logiciels très simples cela peut être superflu.

Mais, on emploi pas des professionnels du développement logiciel pour développer des logiciels simples …

Objectif:

Le bon logiciel bien fait

C’est quoi un bon logiciel bien fait? *1/11*

Un des enjeux d’un développement est de **bien faire le bon produit**.

- C’est quoi un **bon logiciel**?
- C’est quoi un **logiciel bien fait**?

C’est quoi un bon logiciel bien fait? *2/11*

Facteurs de qualité *(Bertrand Meyer):*

Correction
*« La correction est la capacité que possède un produit logiciel de **mener à bien sa tâche**, telle qu’elle a été définie par sa spécification. »*

C’est quoi un bon logiciel bien fait? *3/11*

Facteurs de qualité *(Bertrand Meyer):*

Robustesse
« *La robustesse est la capacité qu’offrent des systèmes logiciels à* ***réagir de manière appropriée à la présence de conditions anormales****.* »

C’est quoi un bon logiciel bien fait? *4/11*

Facteurs de qualité *(Bertrand Meyer):*

Extensibilité
*« L’extensibilité est la facilité d’**adaptation** des produits logiciels aux changements de spécifications. »*

C’est quoi un bon logiciel bien fait? *5/11*

Facteurs de qualité *(Bertrand Meyer):*

Réutilisabilité
« *La réutilisabilité est la capacité des éléments logiciels à servir à la construction de plusieurs applications différentes.* »

C’est quoi un bon logiciel bien fait? *6/11*

Facteurs de qualité *(Bertrand Meyer):*

Compatibilité
« La compatibilité est la facilité avec laquelle des éléments logiciels peuvent être combinés à d’autres. »

C’est quoi un bon logiciel bien fait? *7/11*

Facteurs de qualité *(Bertrand Meyer)*:

Efficacité
« L’efficacité est la capacité d’un système logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l’espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication. »

C’est quoi un bon logiciel bien fait? *8/11*

Facteurs de qualité *(Bertrand Meyer):*

Portabilité
« La portabilité est la facilité avec laquelle des produits logiciels peuvent être transférés d’un environnement logiciel ou matériel à l’autre. »

C’est quoi un bon logiciel bien fait? *9/11*

Facteurs de qualité *(Bertrand Meyer):*

Facilité d’utilisation
« La facilité d’utilisation est la facilité avec laquelle des personnes présentant des formations et des compétences différentes peuvent apprendre à utiliser les produits logiciels et s’en servir pour résoudre des problèmes. *Elle recouvre également la facilité d’installation, d’opération et de contrôle. »*

C’est quoi un bon logiciel bien fait? *10/11*

Facteurs de qualité *(Bertrand Meyer):*

Ponctualité
« La ponctualité est la capacité d’un système logiciel à être livré au moment désiré par ses utilisateurs, ou avant. »

C’est quoi un bon logiciel bien fait? *11/11*

Donc, **bon logiciel bien fait** est un logiciel

- **correct**,
- robuste,
- **extensible**,
- compatible avec d’autres logiciels,
- efficace,
- portable,
- **facile à utiliser**,
- **ponctuel** et
- avec un code réutilisable.

Que va-t-on faire pour bien faire le bon logiciel?

Comment faire des logiciels?

Processus de développement et activités

Processus de développement: *c'est quoi?*

Processus:

*« Ensemble d'**activités** coordonnées et régulées, en partie ordonnées, dont le but est de créer un produit. »*

L'ordre et la manière d'enchaîner les étapes d'un développement est le **processus de développement**.

Processus de développement: *2 choses*

Un processus décrit 2 choses importantes:

- Les **activités** (étapes) *(= quoi?)*
- L'**enchainement** des activités *(= quand?)*

Processus de développement*: pourquoi?*

Un logiciel à développer est un **problème très complexe à résoudre.**

Pour appréhender cette complexité, il faut arriver à **rendre les choses simples**.

Pour cela, il faut **séparer les problèmes** qui peuvent l'être afin de les **résoudre de manière presque indépendante**.

Analogie: « Diviser pour mieux régner. »

Processus de développement*: diviser pour mieux régner*

Un processus définit un enchainement d'activités pendant lesquelles on traite des **problèmes différents**.

Ces problèmes ne sont pas complètement indépendants, c'est pourquoi **l'enchainement est important**.

Quels sont les problèmes à résoudre?

Quoi?	*Qu'est-ce que le logiciel doit faire?* *Comment s'assurer qu'il fait bien ce qu'il doit faire?* *Le logiciel fait-il ce qu'il doit faire?* *Comment s'assurer qu'on développe le bon logiciel?* *A-t-on développé le bon logiciel?*
Comment?	*Comment organiser et construire le logiciel pour qu'il fasse ce qu'il doit faire?* *Comment s'assurer que le logiciel est organisé et construit de manière à faire ce qu'il doit faire?* *Le logiciel est-il organisé et construit de manière à faire ce qu'il doit faire?* *Comment traduit-on cette organisation en code source?* *Le code source est-il bien écrit?*

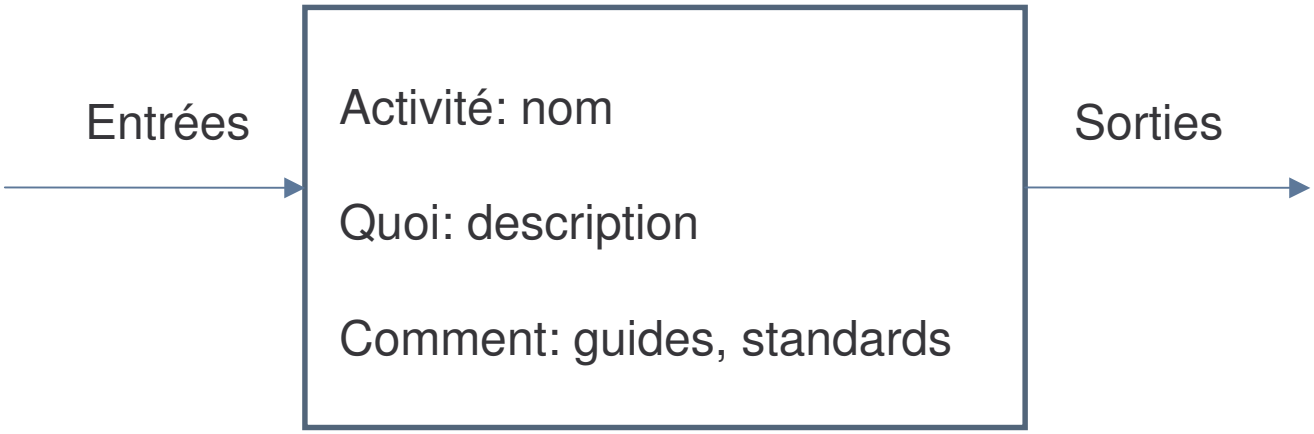
Ces problèmes sont-ils indépendants?

Activité *1/2*

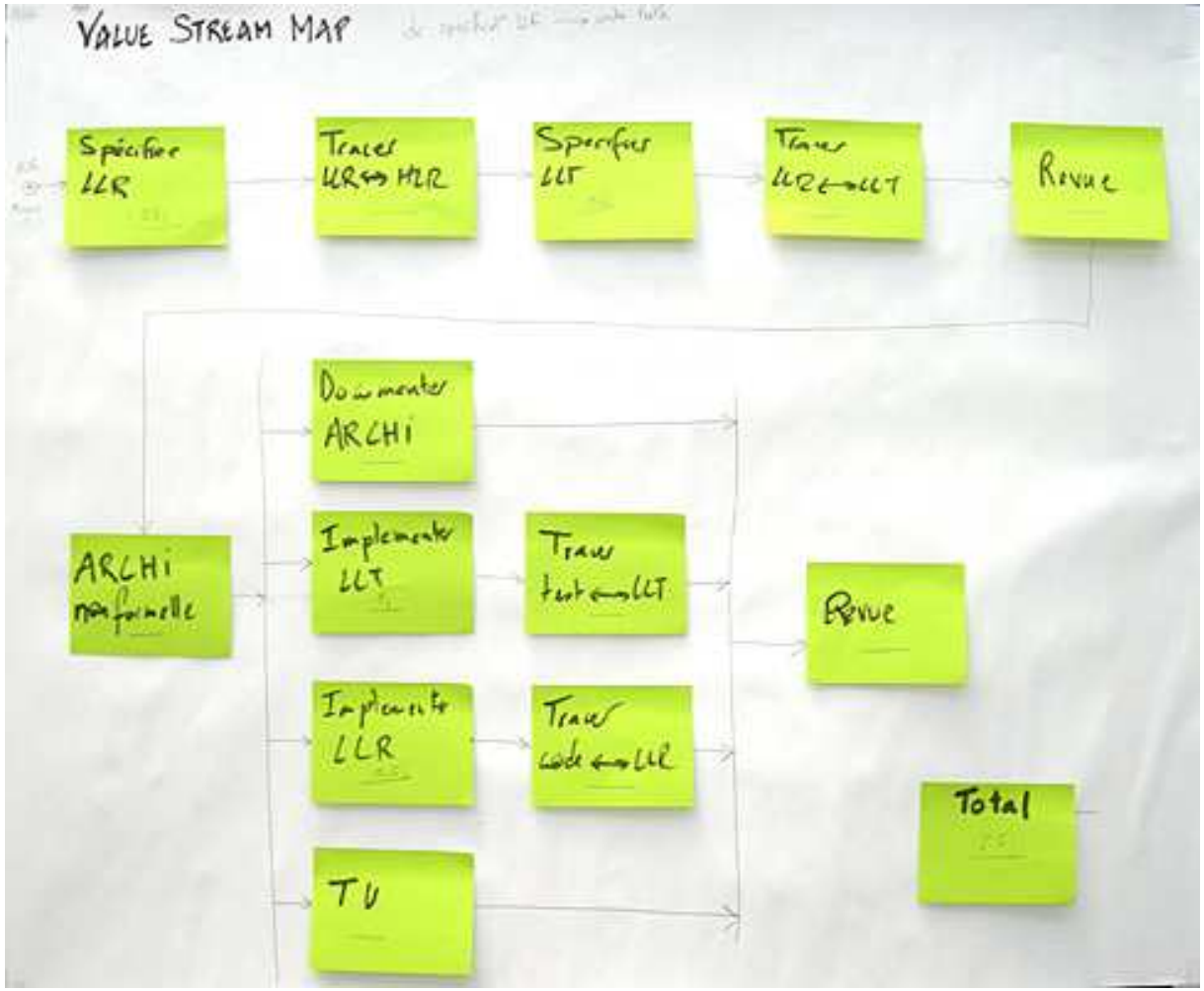
Les activités d'un processus sont souvent décrites en termes de:

Entrées de l'activité *(matière première)*
Sorties de l'activité *(résultat)*
Intervenants et rôles *(qui)*
Description de l'activité *(quoi – quel est le problème à traiter?)*
Standards, guides, « best practices » à appliquer *(comment)*

Activité *2/2*


Entrées
Activité: nom
Quoi: description
Comment: guides, standards
Sorties

Activités

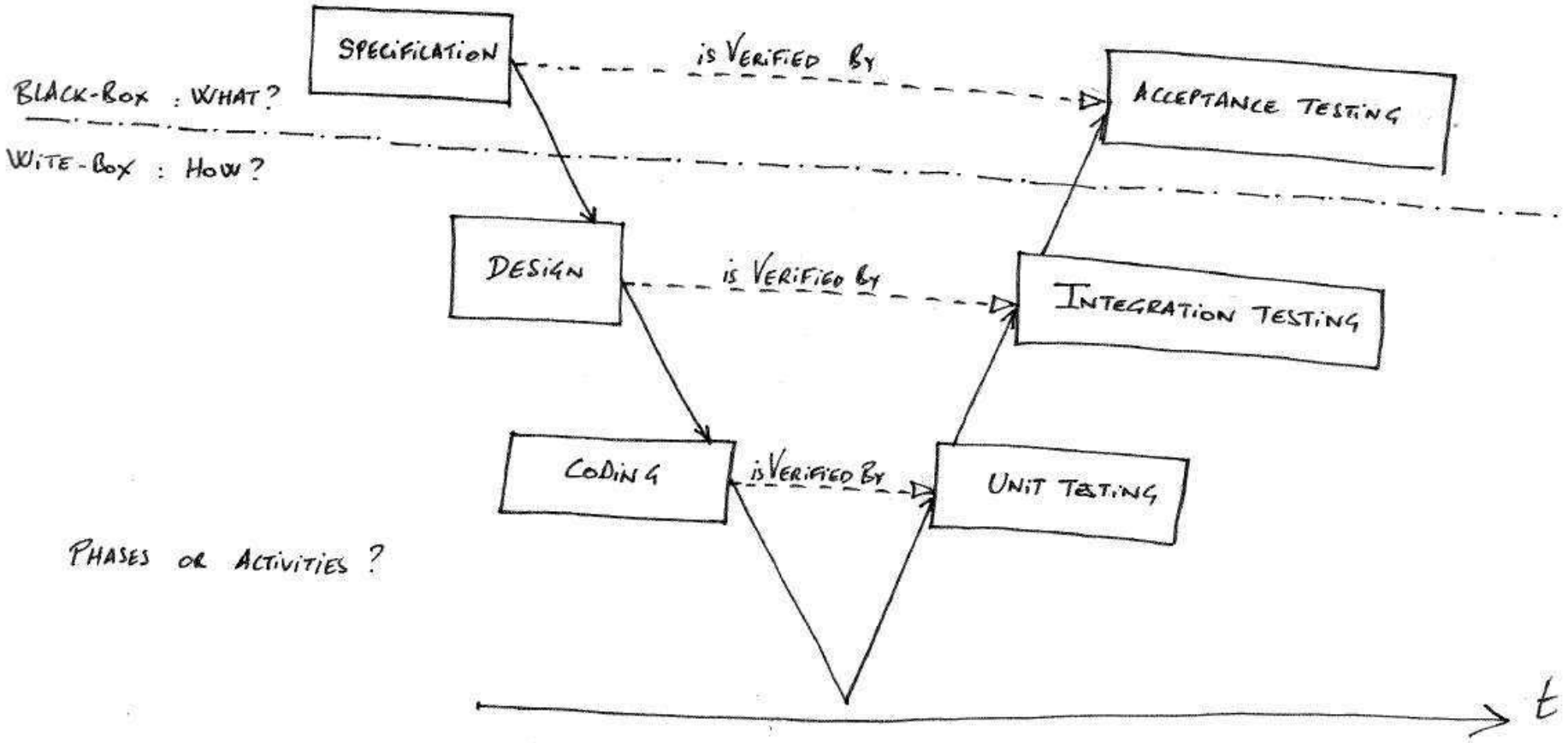

VALUE STREAM MAP
Spécifier LLR
Tracer HLR
Spécifier LLT
Revue
Documenter ARCHI
ARCHI informelle
Implémenter LLT
Revue
Implémenter LLR
Total
TU

Activités: *les classiques*

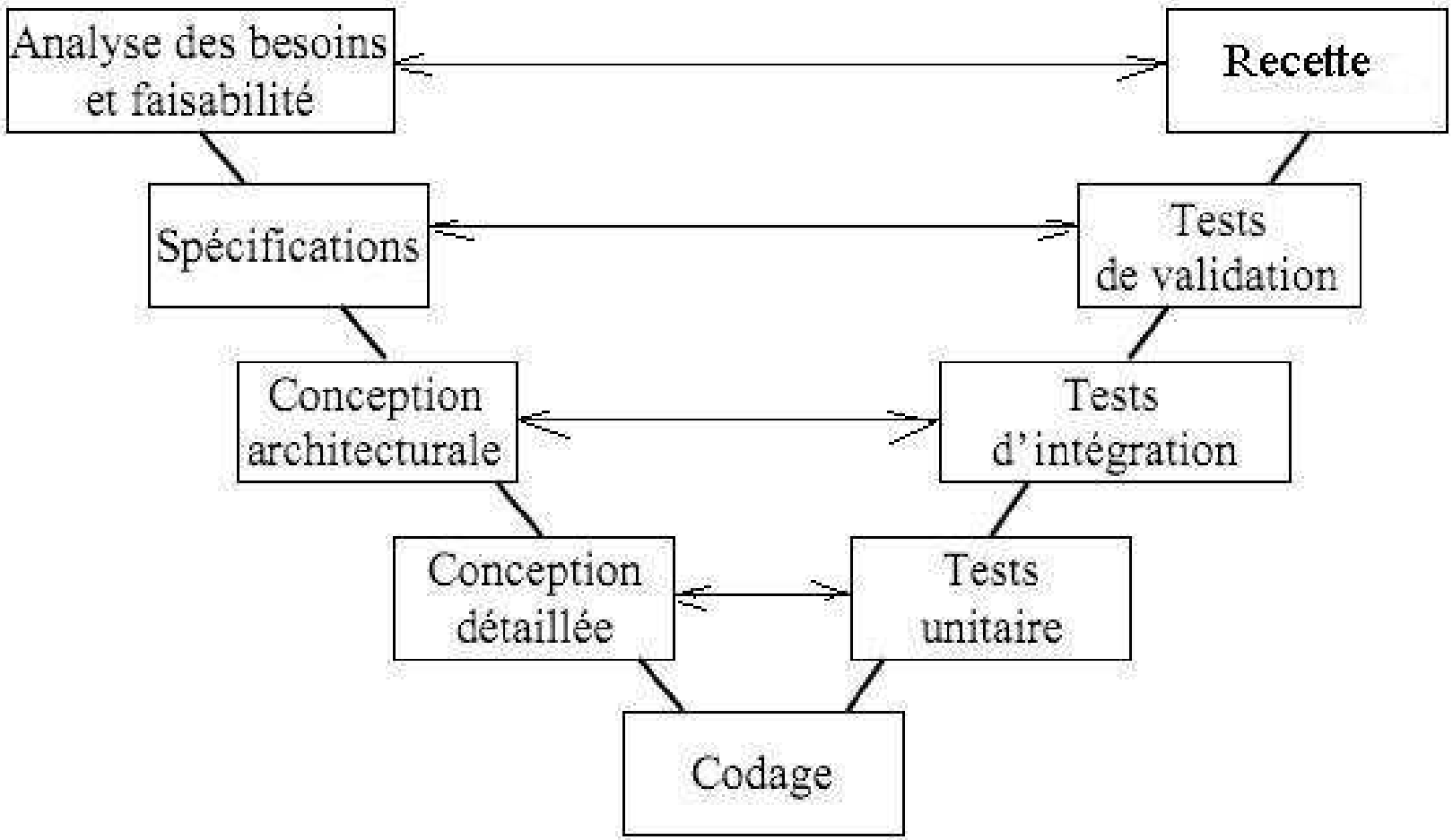
Activités courantes et problèmes traités:

Spécifier *Qu'est-ce que le logiciel doit faire?* *Comment s'assurer qu'il le fait?* *Comment s'assurer qu'on développe le bon logiciel?*	**Valider** *Le logiciel fait-il ce qu'il doit faire?* *Développe t on le bon logiciel?*
Concevoir *Comment organiser le logiciel pour qu'il fasse ce qu'il doit faire?* *Quelles choix techniques faut-il faire pour que le logiciel fasse ce qu'il doit faire?* *Comment s'assurer que le logiciel est organisé et construit de manière à faire ce qu'il doit faire?*	**Intégrer** *Le logiciel est-il organisé et construit de manière à faire ce qu'il doit faire?*
Coder *Comment traduit-on cette organisation en code source?*	**Tester unitairement** *Le code source est-il bien écrit?*

Processus: *le cycle en V*



Le cycle en V*: alternative*



Le cycle en V

- A ce jour, le cycle en V reste **le cycle de vie le plus utilisé**.
- C'est un cycle de vie **orienté test:**
  - A chaque activité créative *(spécification, conception et codage)* correspond une activité de **vérification** *(validation, intégration, tests unitaires)*.
  - La vérification est prise en compte au moment même de la création.

Activités: *spécifier*

Activité	**Spécifier**
Description	Décrire ce que doit faire le logiciel (comportement en boite-noire). Décrire comment vérifier en boite-noire que le logiciel fait bien ce qui est exigé.
Entrées	Client qui a une idée de ce qu'il veut.
Sorties	Une spécification (description des besoins du client). Des procédures de validation.

Une spécification peut suivre de **nombreux** formalismes: cas d'utilisation, modèles UML, user-stories, exigences …

Les procédures de validation peuvent être des procédures manuelles ou des tests.

Spécifier: exemple

En **entrée** de la spécification, il y a un **besoin d'un client**:

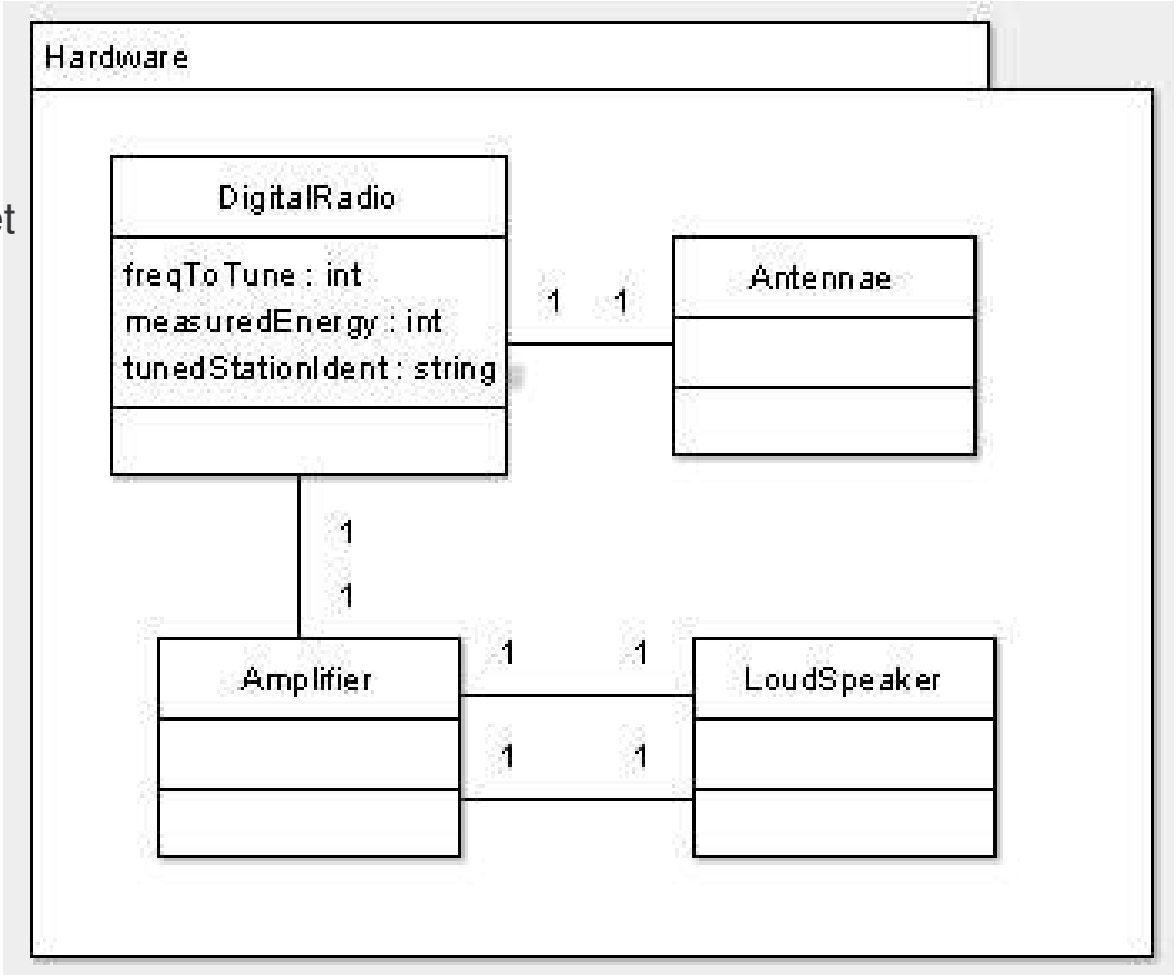
- Le logiciel *DigitalTuner* doit s'interfacer avec la radio numérique *DigitalRadio*.
- Le logiciel affiche la fréquence et le nom de la station à l'écoute.
- Le logiciel peut mémoriser la station à l'écoute.
- L'utilisateur sélectionne une station mémorisée pour l'écouter.
- L'utilisateur incrémente/décrémente la fréquence à l'écoute.
- L'utilisateur passe à la précédente/prochaine station radio dans la bande de fréquence.
- Faire un premier produit embarqué sur automobile. Prévoir un produit embarqué sur téléphone portable, puis un produit embarqué sur lecteur MP3.

Le besoin est trop vague, il faut le clarifier: c'est la **spécification**.

Spécifier: *exemple*

(Diagramme de classes UML)

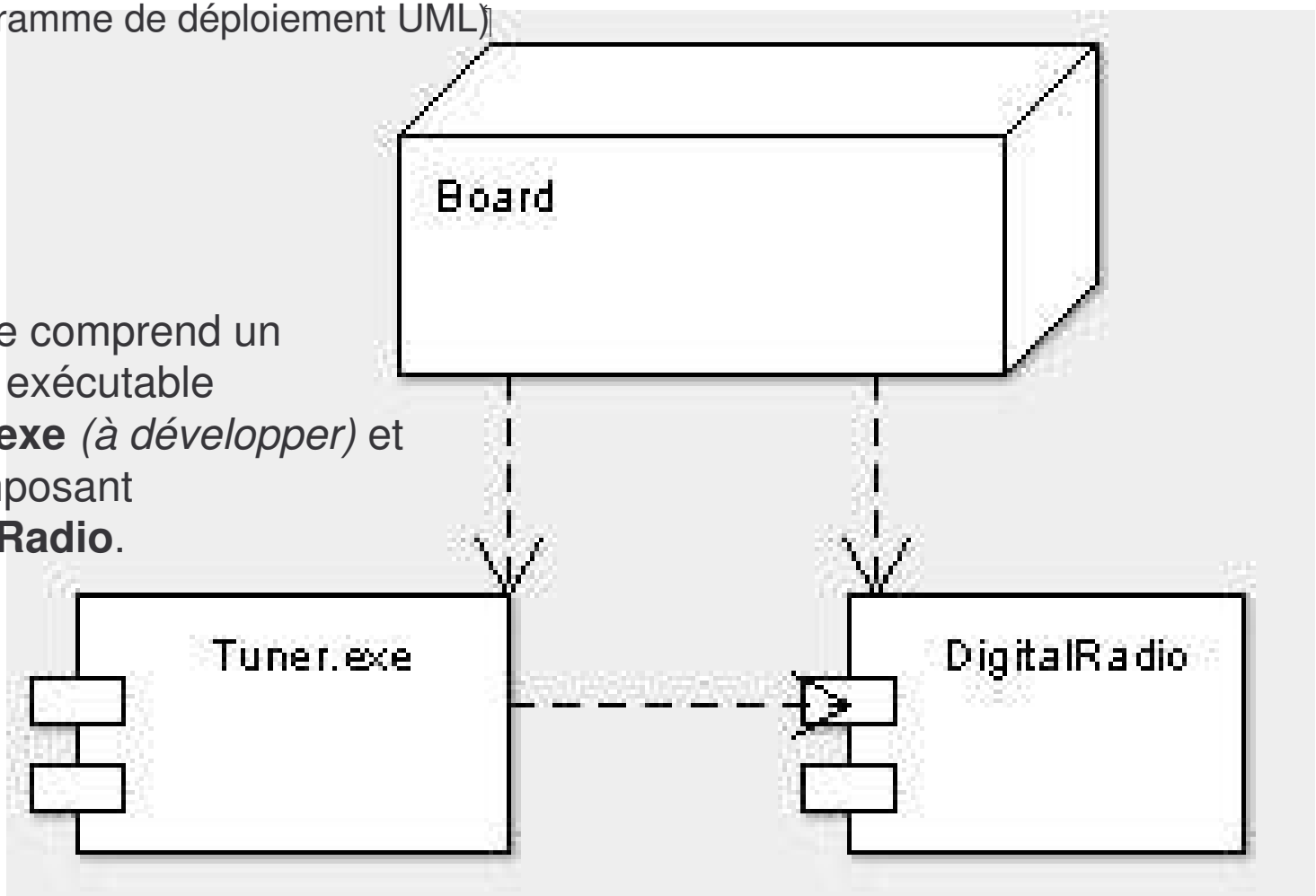
Le composant DigitalRadio permet de:
- positionner la fréquence à capter
- lire l'énergie du signal capté
- lire le nom de la station captée



Spécifier: *exemple*

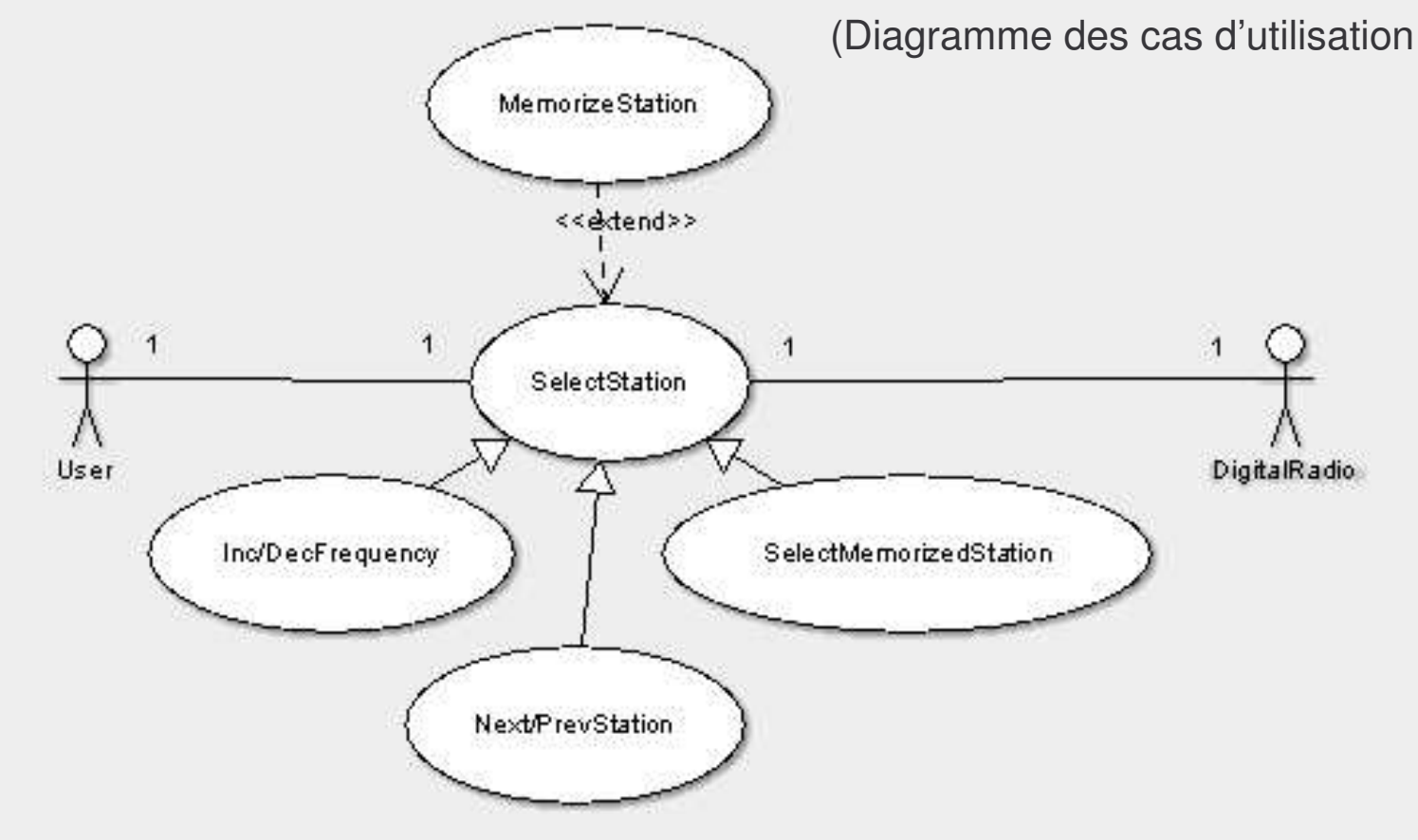
(Diagramme de déploiement UML)

La carte comprend un logiciel exécutable **Tuner.exe** *(à développer)* et un composant **DigitalRadio**.



Spécifier: *exemple*

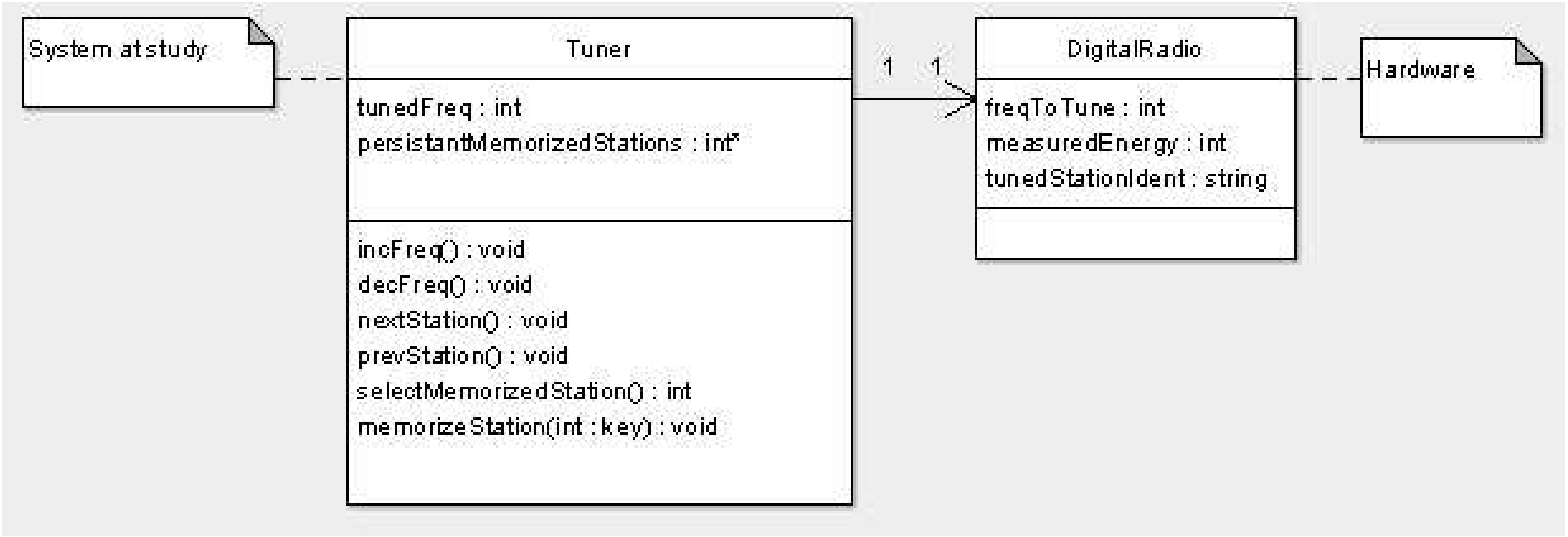
(Diagramme des cas d'utilisation UML)



L'utilisateur utilise le logiciel Tuner pour sélectionner une station à capter.
Il y a différentes manières de faire (inc/dec, next/prev, selectMemorized)/
Eventuellement, il peut mémoriserla station captée.

Spécifier: *exemple*

(Diagramme de classes UML)



Spécifier: *exemple*

Interface du composant ***DigitalRadio***:

- Le nom de la station correspondant à la fréquence captée est donné par la sortie ***tunedStationIdent***.
- Si la fréquence captée ne correspond à aucune station, la sortie ***tunedStationIdent*** vaut « ».
- Le nom d'une station radio est disponible **N s** après avoir positionné la fréquence à capter.
- Une station est identifiée par un pic d'énergie sur la sortie ***measuredEnergy*** du composant *DigitalRadio*.
- La commande ***freqToTune*** est lue et les sorties ***tunedStationIdent*** et ***measuredEnergy*** sont écrites cycliquement à la fréquence **F**.

Spécifier: *exemple*

Exigences allouées au logiciel *DigitalTuner (1/3)*

[1.1] L'utilisateur peut utiliser le logiciel *DigitalTuner* pour incrémenter la fréquence captée. Dans ce cas, la commande *freqToTune* du composant *DigitalRadio* est incrémentée de *dF Hz*.

[1.2] Si l'utilisateur incrémente la fréquence à l'écoute au-delà de *Fmax*, alors la fréquence à l'écoute devient *Fmin*.

[2.1] L'utilisateur peut utiliser le logiciel *DigitalTuner* pour décrémenter la fréquence captée. Dans ce cas, la commande *freqToTune* du composant *DigitalRadio* est décrémentée de *dF Hz*.

[2.2] Si l'utilisateur décrémente la fréquence à l'écoute au-dessous de *Fmin*, alors la fréquence à l'écoute devient *Fmax*.

Spécifier: *exemple*

Exigences allouées au logiciel *DigitalTuner* (2/3)

[3.1] L'utilisateur peut utiliser le logiciel *DigitalTuner* pour capter la prochaine station sur la bande de fréquences *[Fmin .. Fmax].*

[3.2] S'il ne reste plus de signal à capter jusqu'à *Fmax*, la recherche repart de *Fmin*.

[4.1] L'utilisateur peut utiliser le logiciel Tuner pour capter la précédente station sur la bande de fréquences *[Fmin .. Fmax]*.

[4.2] S'il ne reste plus de signal à capter jusqu'à *Fmin*, la recherche repart de *Fmax*.

Spécifier: *exemple*

Exigences allouées au logiciel *DigitalTuner* (3/3)

[5.1] L'utilisateur peut mémoriser la fréquence captée.

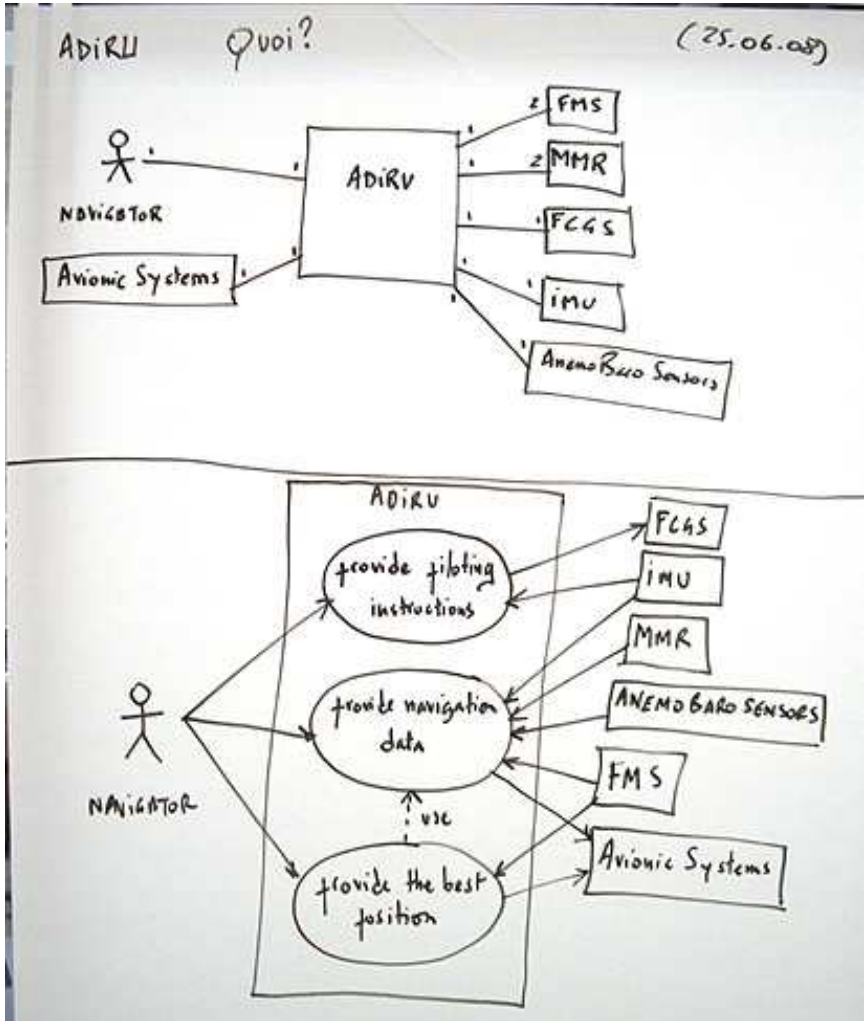
[5.2] Même après avoir rallumé la radio, l'utilisateur peut sélectionner une fréquence précédemment mémorisée. La commande *freqToTune* du composant *DigitalRadio* est positionnée à la fréquence mémorisée.

[5.3] A la mise sous tension, le logiciel DigitalTuner demande à DigitalRadio de capter la fréquence écoutée à la fin de la dernière mise sous tension.

[6.1] L'utilisateur peut lire la fréquence du signal capté.

[6.2] Si la fréquence captée est une station, alors l'utilisateur peut lire le nom de la station.

Spécifier = *le quoi*

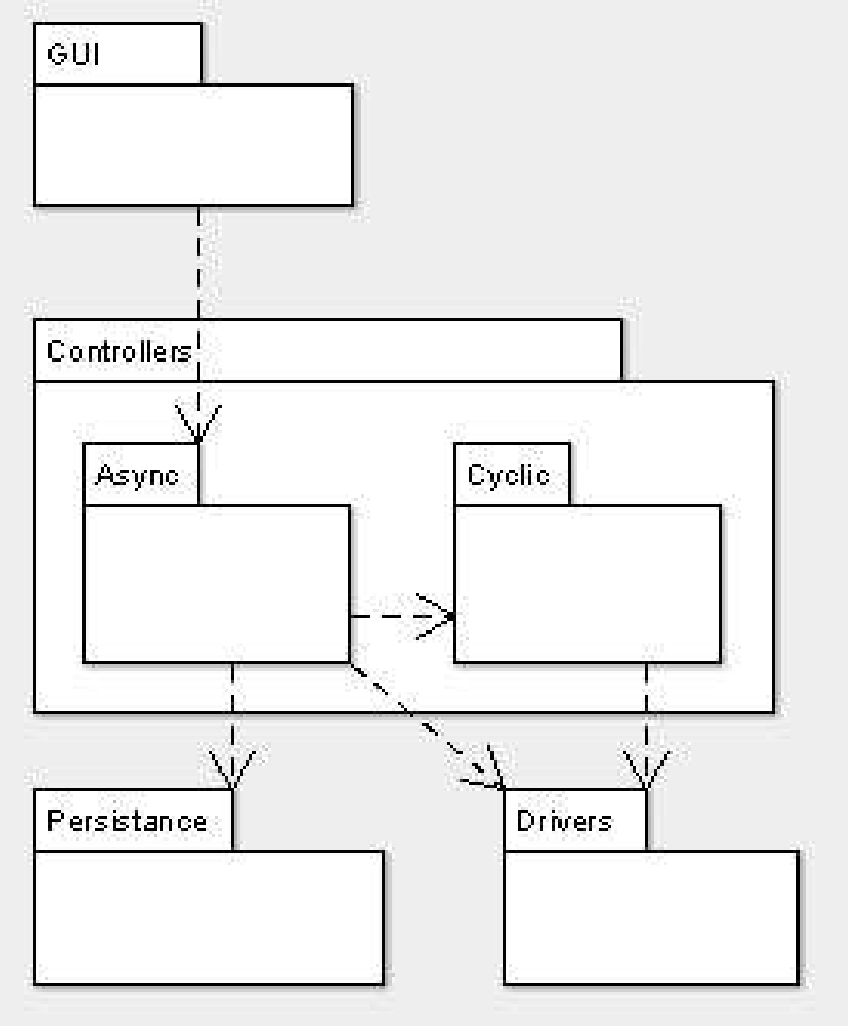


Activités: *concevoir*

Activité	**Concevoir**
Description	**Organiser** le logiciel afin qu'il puisse satisfaire les exigences de la spécification. Faire les principaux **choix techniques** pour satisfaire les exigences de la spécification.
Entrées	Une spécification.
Sorties	Une description des décisions de conception. Des procédures de tests qui permettent de vérifier que les décisions de conception sont correctement implémentées en code source et qu'elles contribuent à satisfaire les exigences de la spécification.

Une conception peut prendre de nombreux formalismes: description textuelle des décisions de l'architecture, modèles UML, code source …

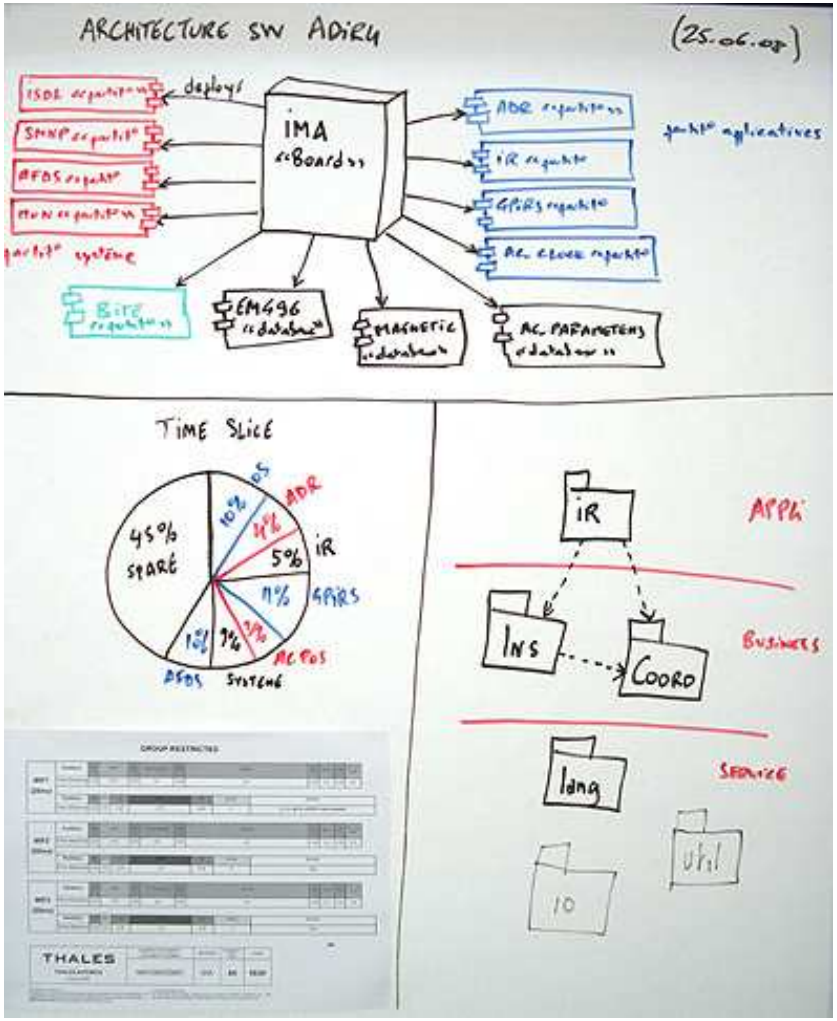
Concevoir: *exemple*



Idée d'architecture candidate:

- **GUI**: *composant responsable de l'interface utilisateur.*
- **Controllers::Async**: *composant contenant la logique de contrôle asynchrone (traitement des commandes utilisateur) et pilotage du composant DigitalRadio.*
- **Controllers::Cyclic**: *composant contenant la logique de contrôle cyclique de surveillance et pilotage du composant DigitalRadio.*
- **Persistance**: composant contenant la logique de gestion des données persistantes (stations mémorisées).
- **Drivers**: *composant contenant les moyens de pilotage et surveillance du composant DigitalRadio.*

Concevoir = *le comment*



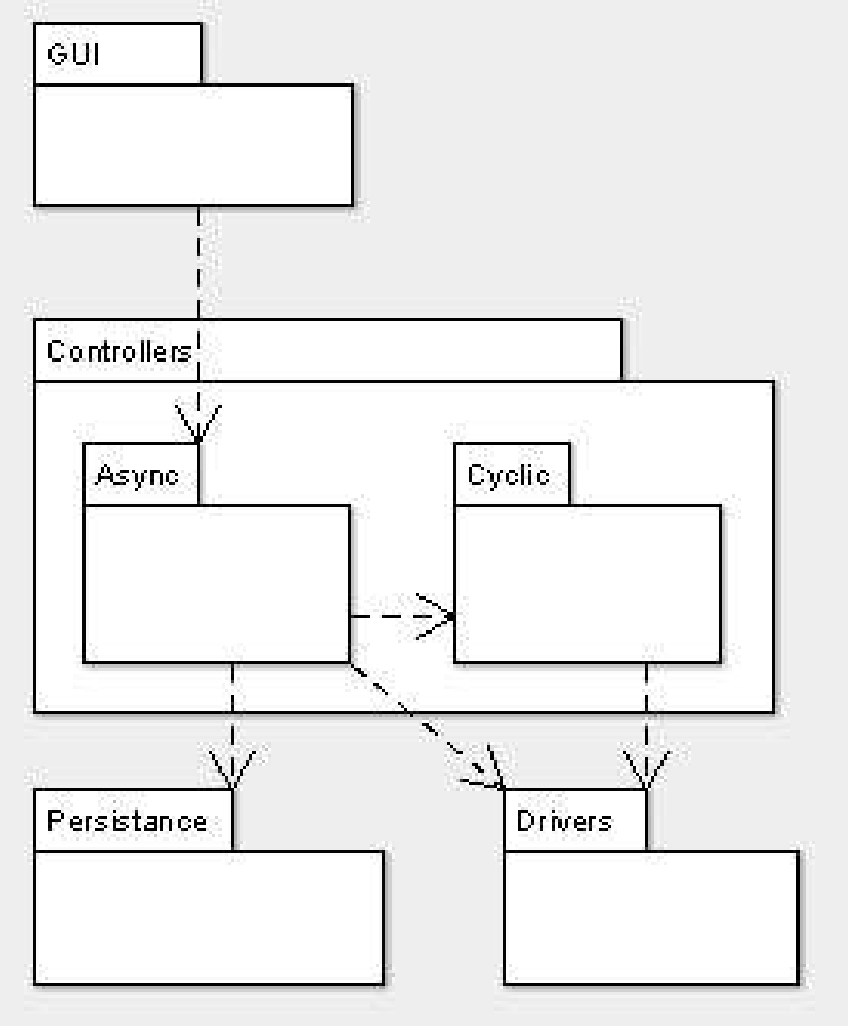
Activités: *coder*

Activité	**Coder et tester**
Description	Écrire le code source du logiciel. Tester le comportement du code source afin de vérifier qu'il réalise les responsabilités qui lui sont allouées.
Entrées	Spécification, conception.
Sorties	Code source. Tests unitaires. Documentation électronique navigable?

Activités: *intégrer*

Activité	**Intégrer**
Description	Assembler le code source du logiciel (éventuellement partiellement) et dérouler les tests d'intégration.
Entrées	Conception, code source, tests d'intégration (tests).
Sorties	Un rapport de test d'intégration.

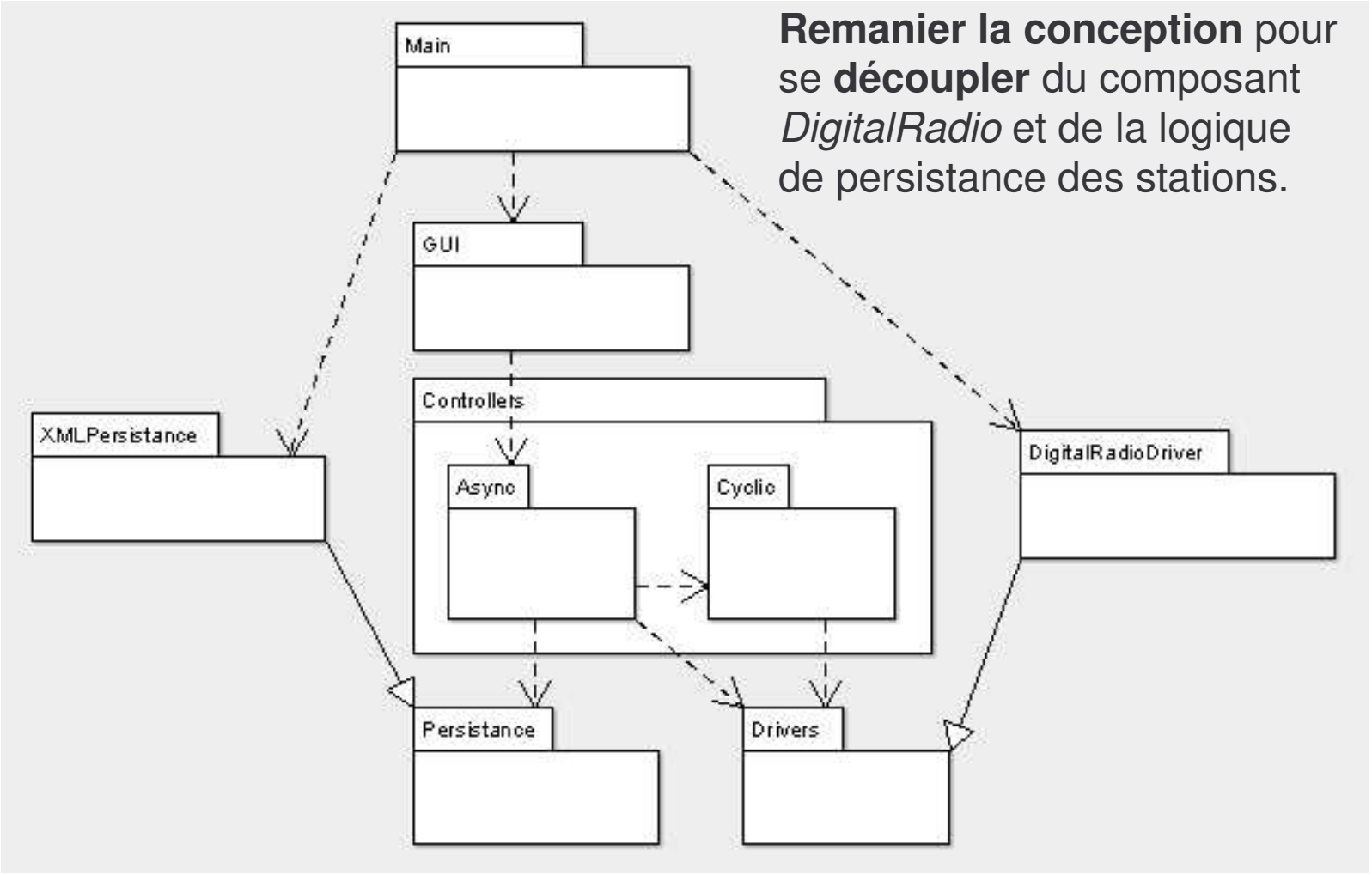
Intégrer: *exemple*



Conception difficile à tester par morceaux!

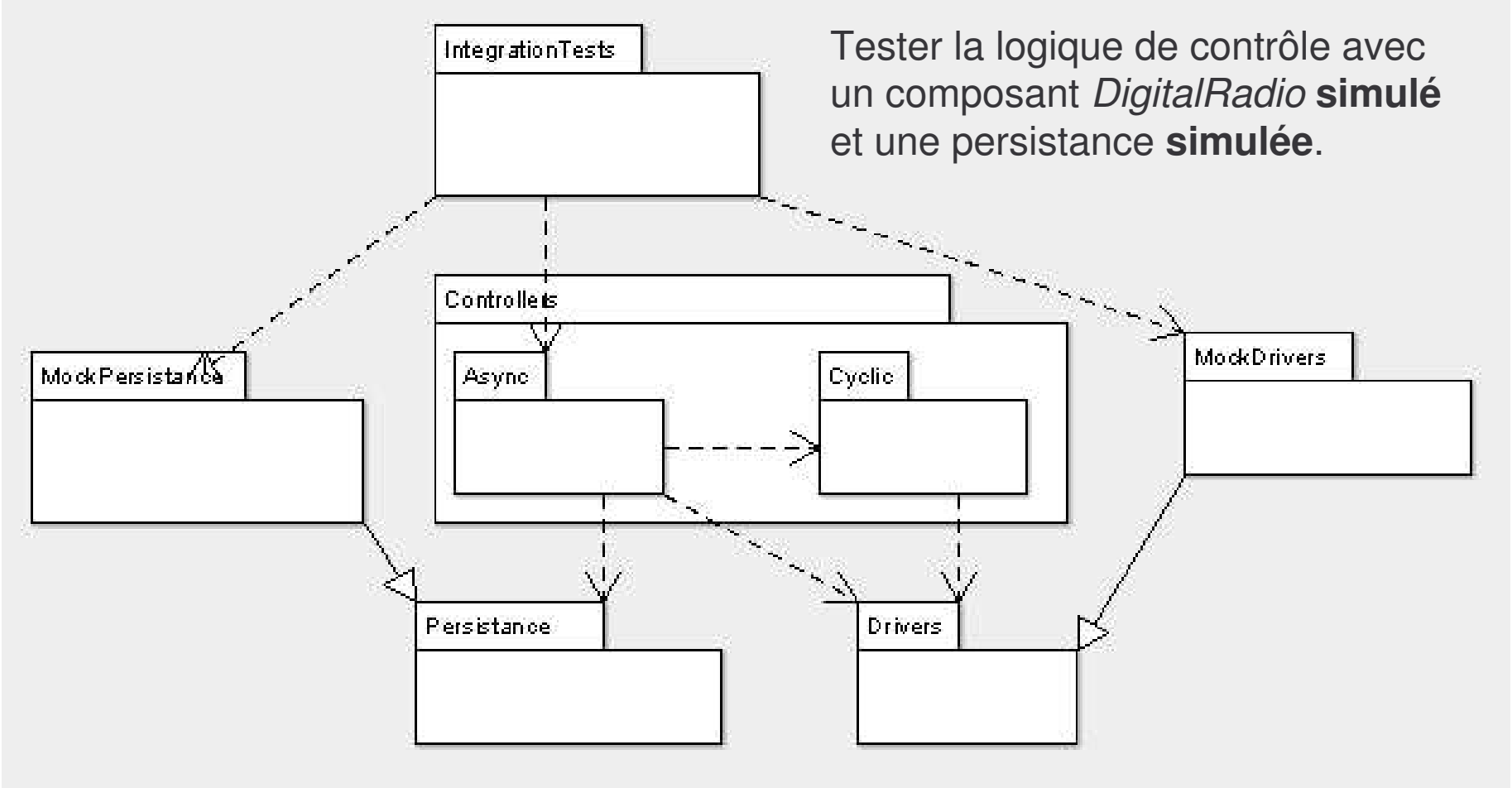
Difficile de tester une application partielle?

Intégrer: *exemple*



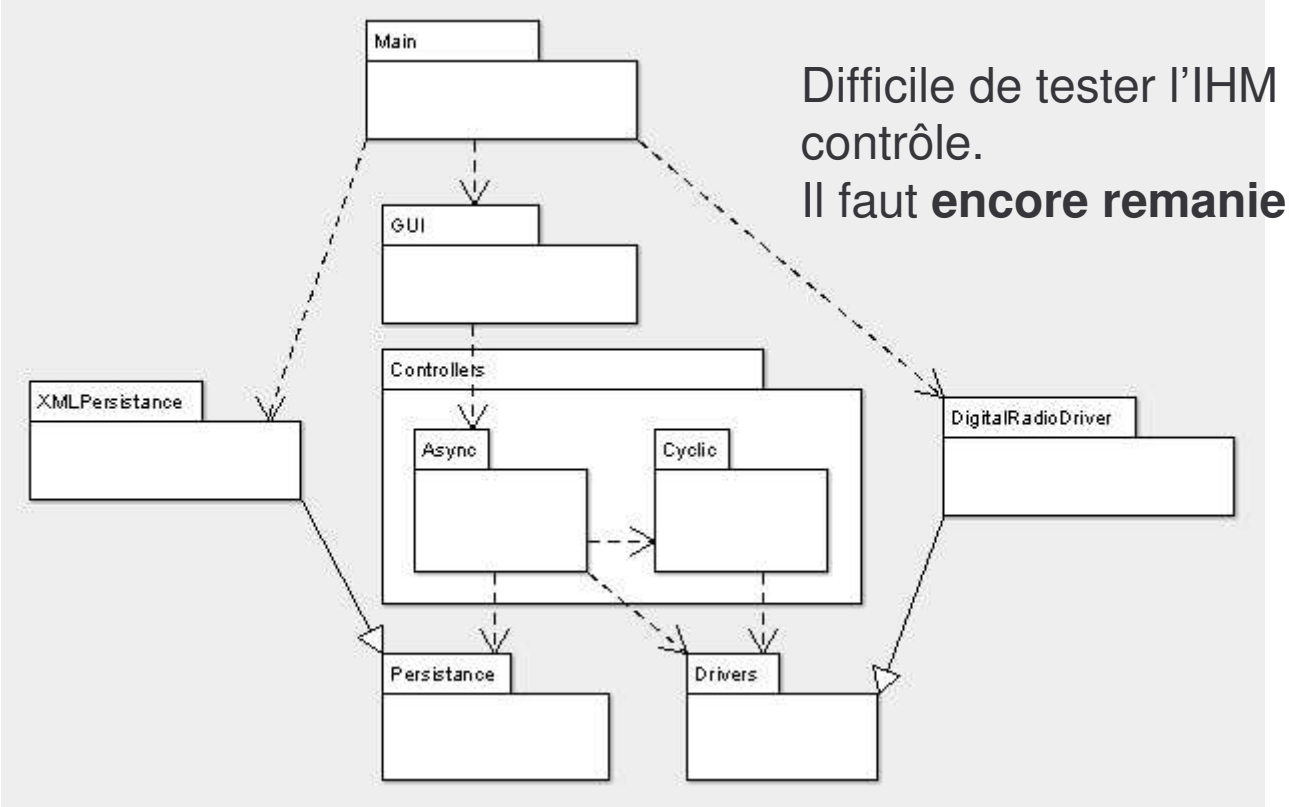
Remanier la conception pour se **découpler** du composant *DigitalRadio* et de la logique de persistance des stations.

Intégrer: *exemple*



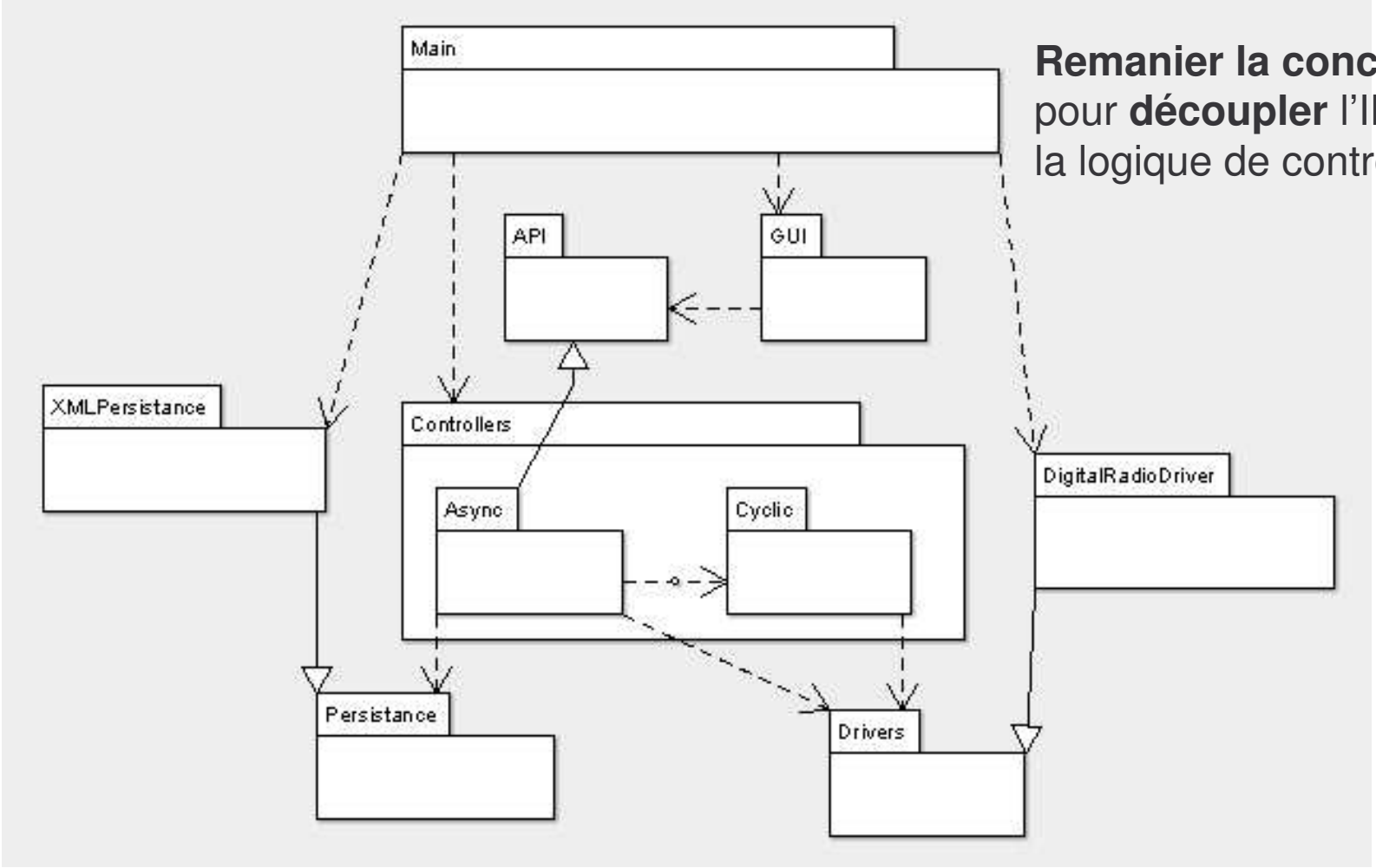
Tester la logique de contrôle avec un composant *DigitalRadio* **simulé** et une persistance **simulée**.

Intégrer: *exemple*



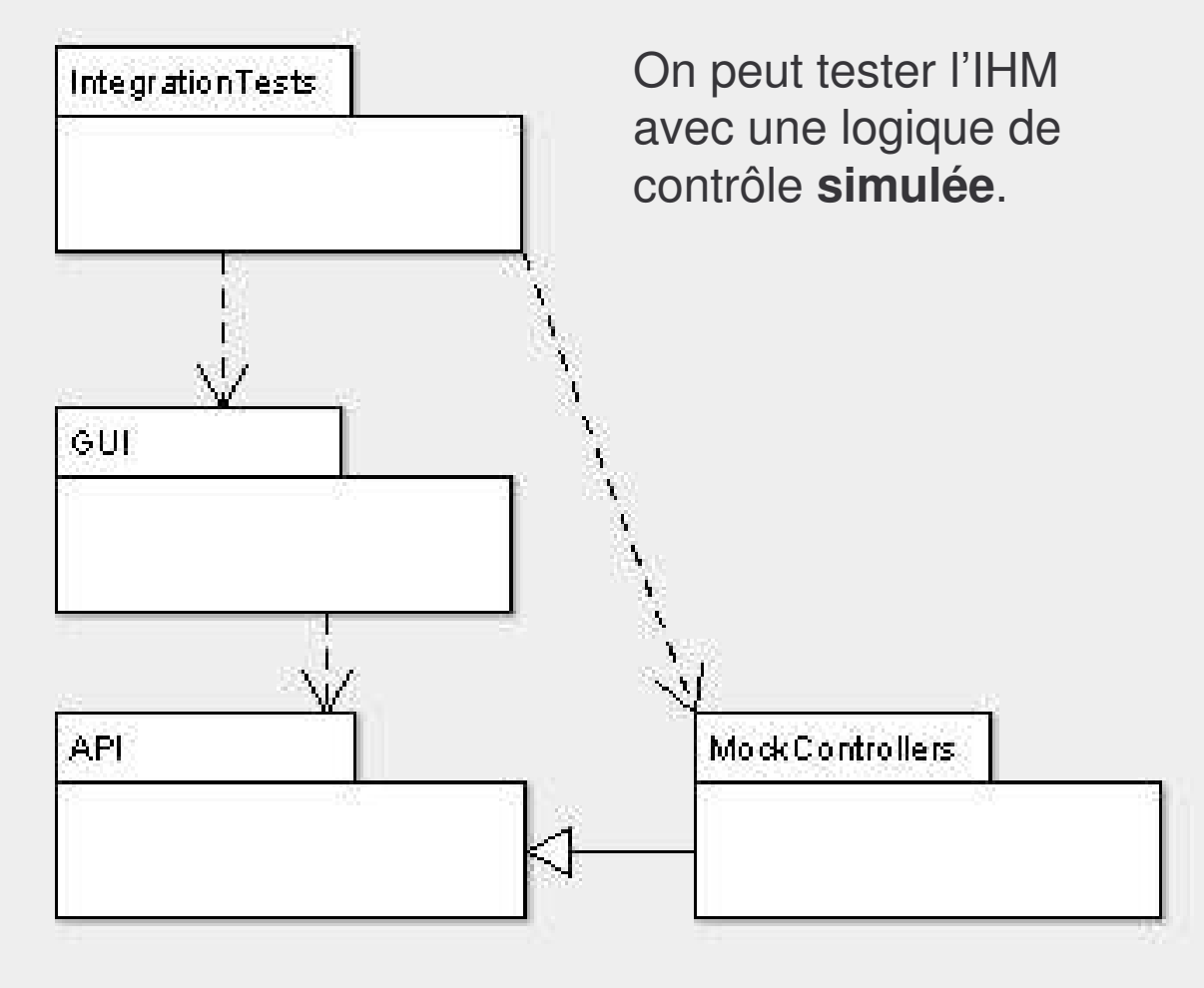
Difficile de tester l'IHM sans la logique de contrôle.
Il faut **encore remanier la conception!**

Intégrer: *exemple*



Remanier la conception pour **découpler** l'IHM de la logique de contrôle.

Intégrer: *exemple*



Intégrer: *exemple*

Conclusion de l'exemple:

- Envisager une stratégie d'intégration du logiciel nous à contraint à **remanier** la conception pour la rendre testable.

- De manière générale, il faut **toujours penser à la testabilité de ce que l'on développe**.

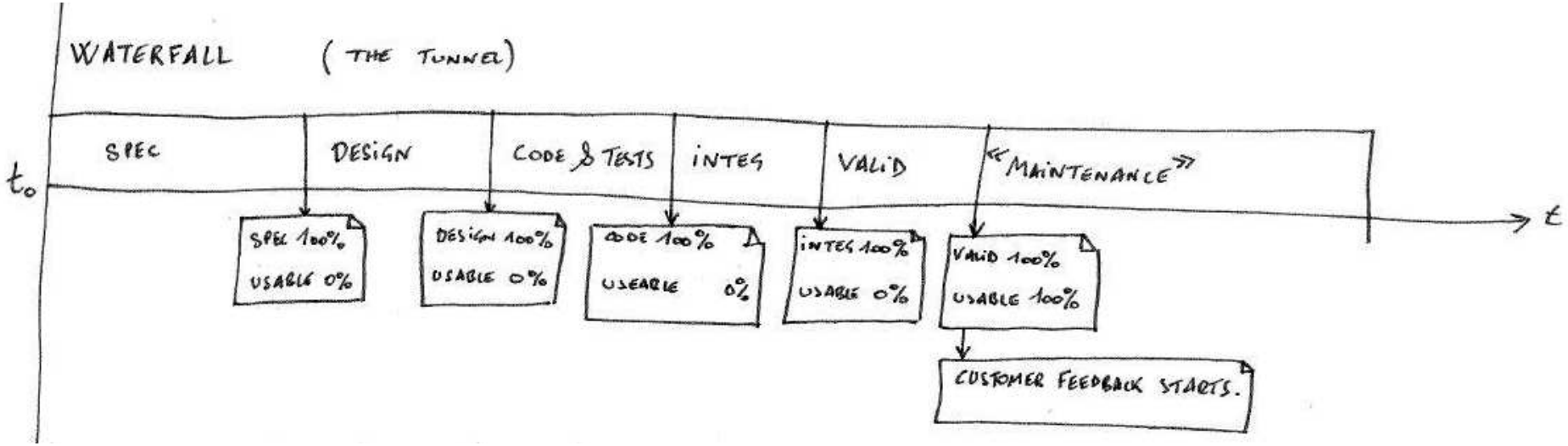
Activités: *valider*

Activité	**Valider**
Description	Construire le logiciel complet exécutable. Dérouler les tests de validation sur le logiciel complet exécutable.
Entrées	Logiciel complet exécutable à valider. Tests de validation.
Sorties	Rapport de tests de validation.

Cycle en V*: efficace?*

Le cycle en V est-il adapté pour:

- **Bien** construire le logiciel?
- Construire le **bon** logiciel?



Cycle en V*: faiblesses*

Construit-on bien le logiciel?

Les choix techniques (spec, conception, codage) sont **validés très (trop) tard par test**.

- Beaucoup de choix techniques sont pris **sans disposer d'information concrète** (prise de risques).
- Il faut attendre longtemps pour savoir si on a **bien construit le logiciel**.

Cycle en V*: faiblesses*

Construit-on le bon logiciel?

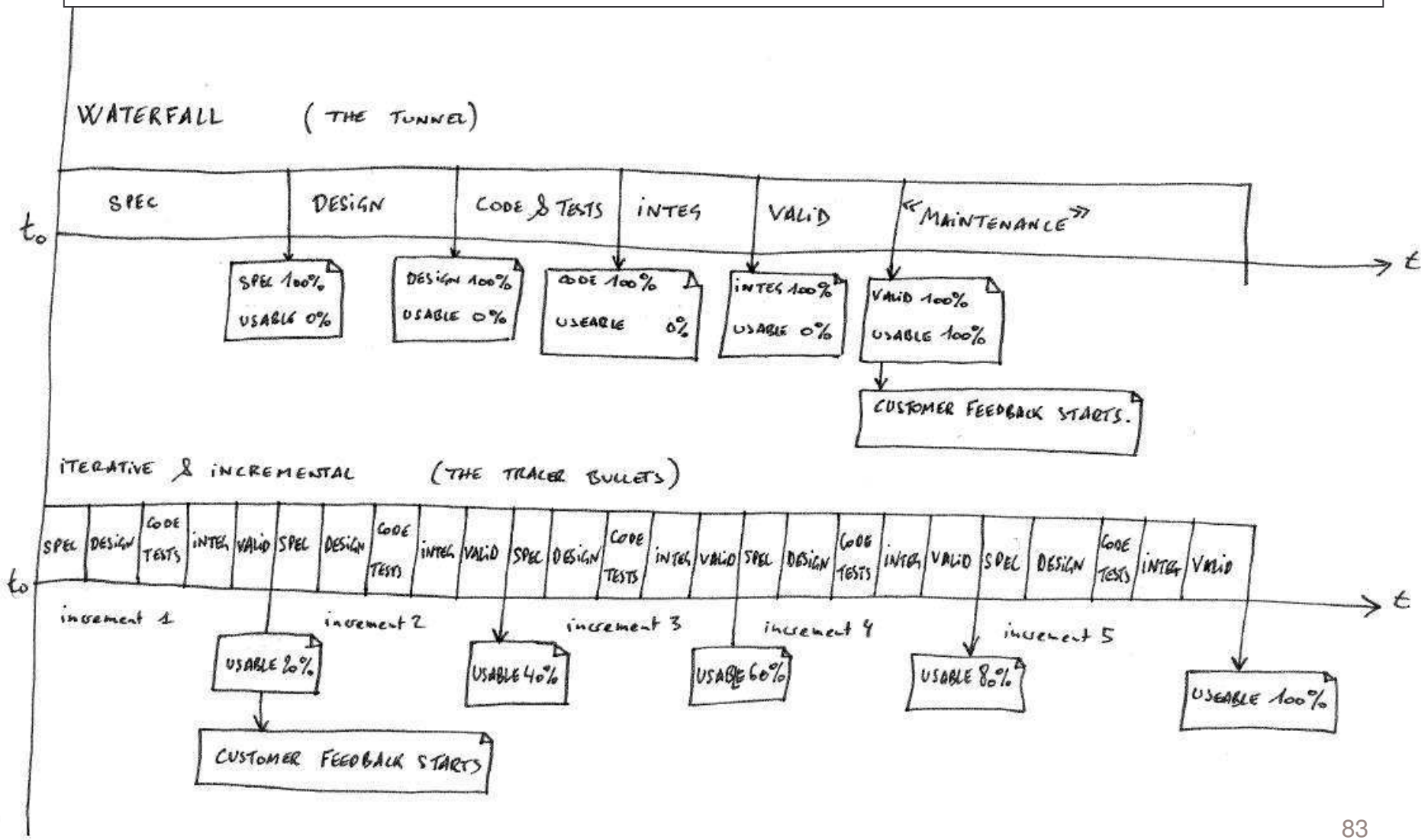
Le logiciel est **utilisé très (trop) tard**.

- Il faut attendre longtemps pour savoir si on a **construit le bon logiciel**.
- Difficile d'impliquer les utilisateurs lorsque qu'un logiciel utilisable n'est disponible qu'à la dernière phase …

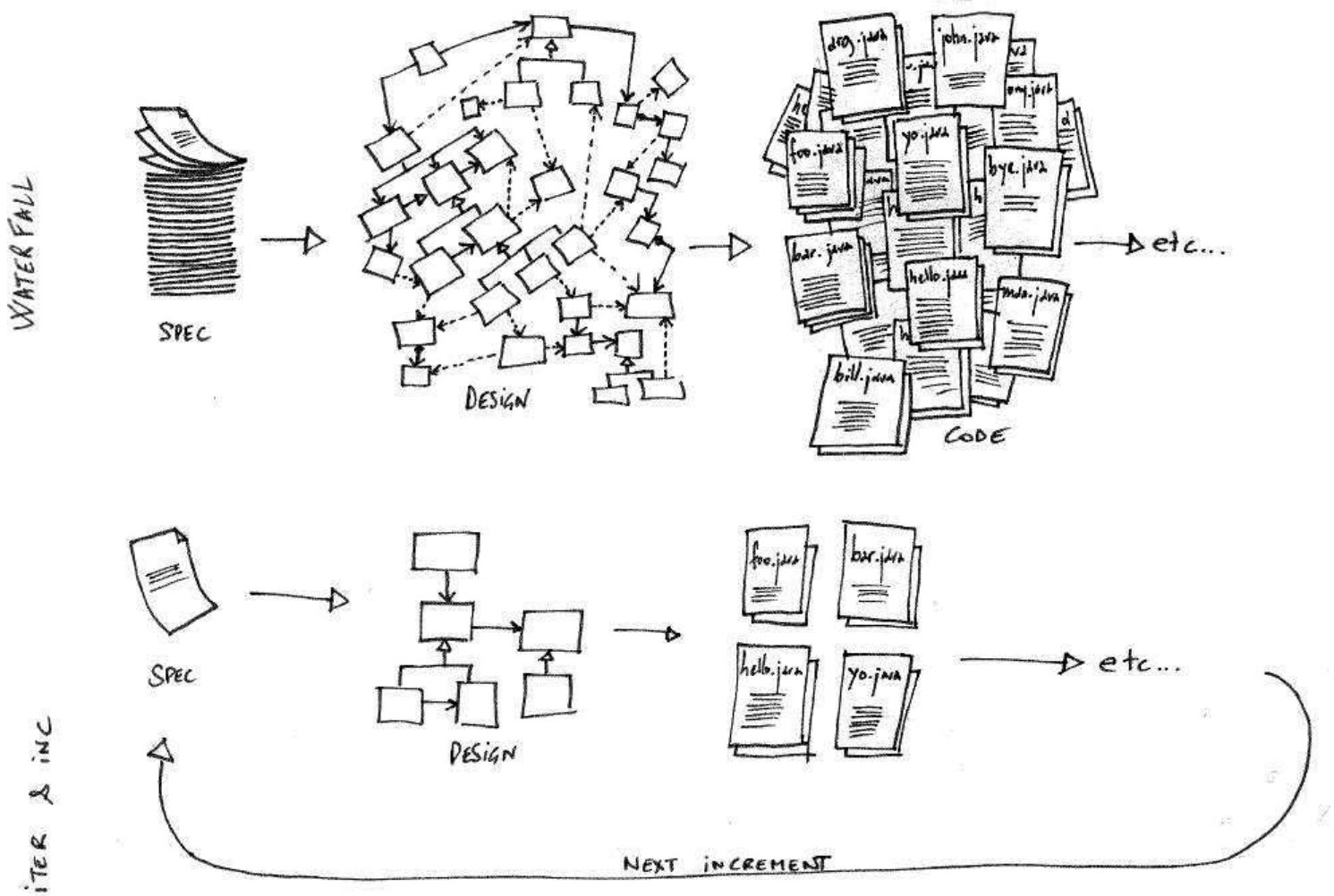
Cycle itératif et incrémental

Le cycle itératif et incrémental est né **en réaction aux faiblesses du cycle en V**.

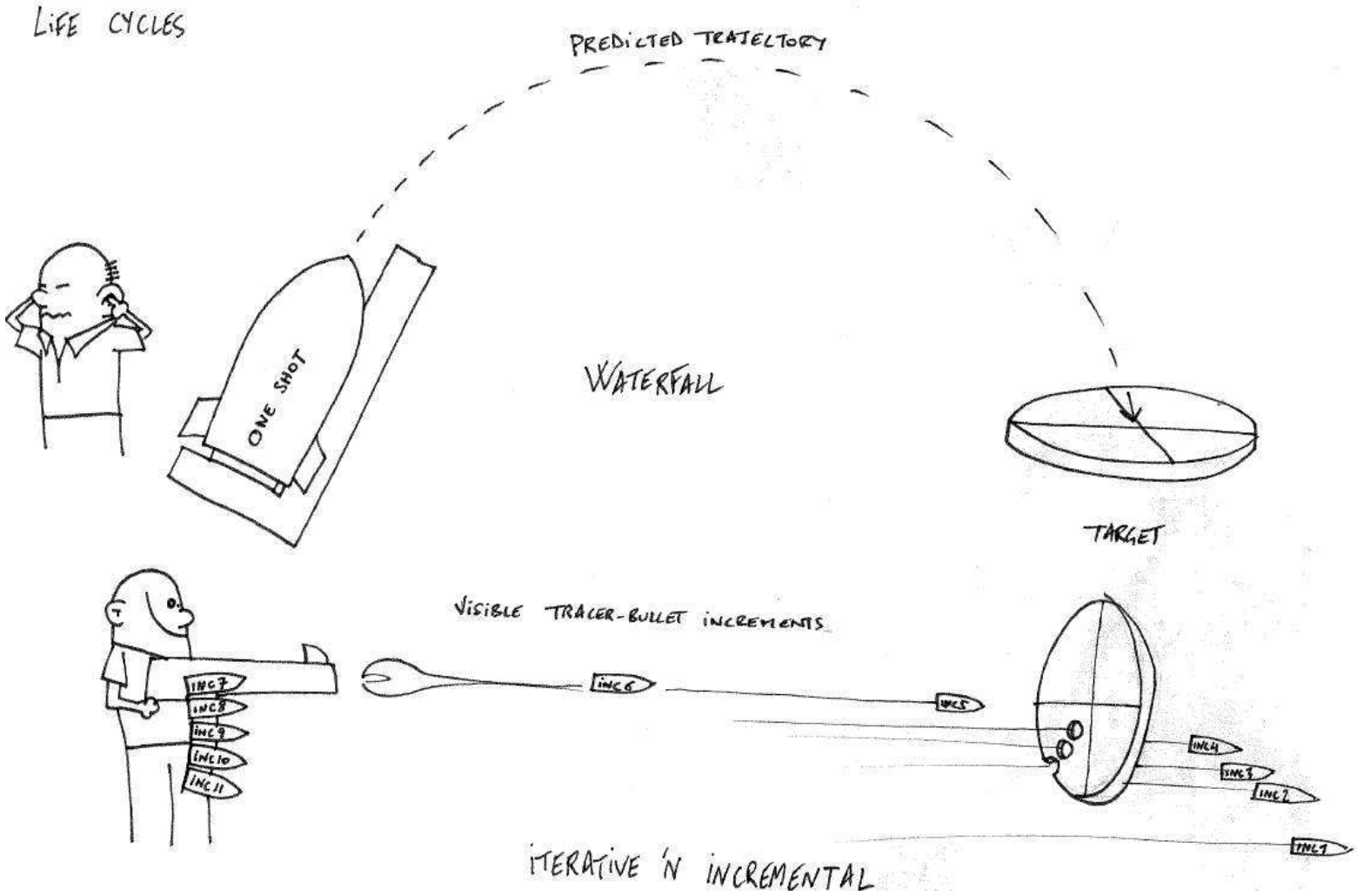
Cycle en V versus iter&inc



Cycle en V versus iter&inc



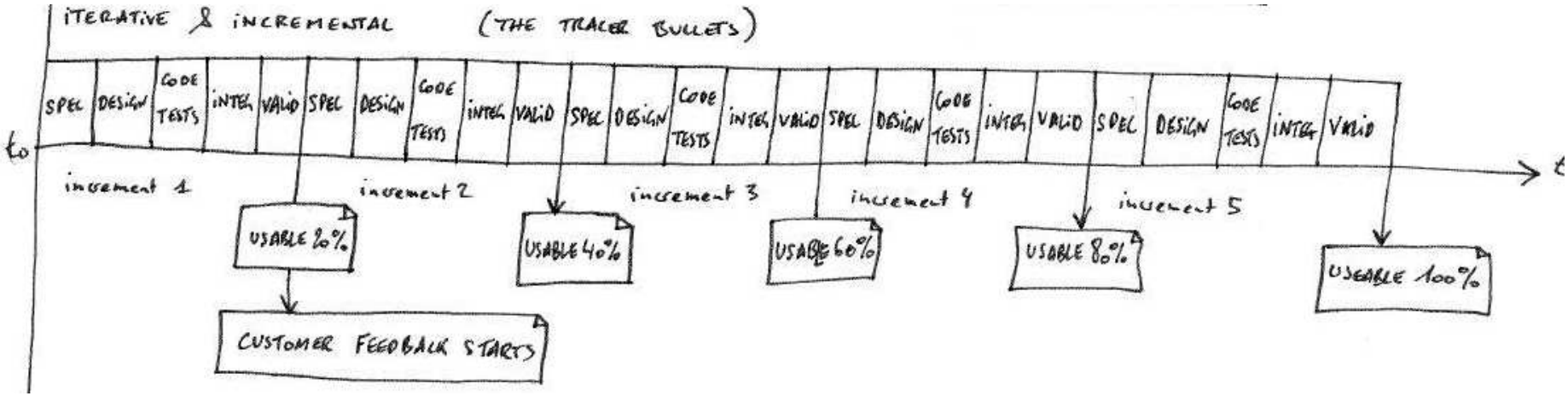
Cycle en V versus iter&inc



Cycle iter&inc: *efficace?*

Le cycle iter&inc est-il adapté pour:

- **Bien** construire le logiciel?
- Construire le **bon** logiciel?



Cycle itér&inc: *avantages*

- Les choix techniques (spécification, conception, codage) sont **validés très tôt par test**.

  L'enchainement des itérations permet de **régulièrement** vérifier que l'on **construit bien le logiciel.**

- Le logiciel est **utilisé très tôt**.

  L'enchainement des itérations permet aux utilisateurs de vérifier régulièrement que l'on **construit le bon logiciel**.

Cycle iter&inc

PRATIQUE: **Adoptez un cycle de vie itératif et incrémental**
Développez de petits incréments fonctionnels livrés en courtes itérations.

Analogie: « Un tiens vaut mieux que deux tu l'auras. »

Remplacez les phases par des activités.

Un incrément fonctionnel est un besoin du client.

Cycle iter&inc

« Un système complexe qui fonctionne a toujours évolué à partir d'un système simple qui a fonctionné…

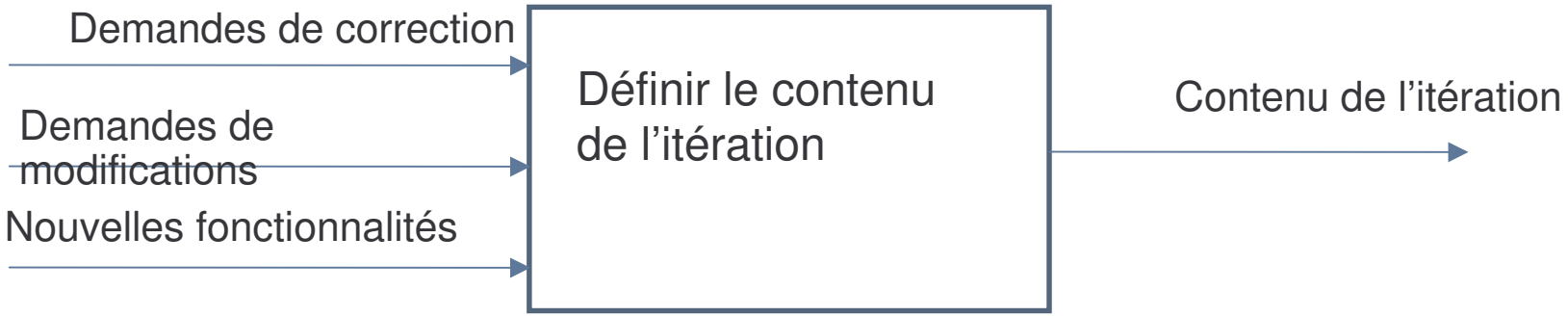
Un système complexe conçu à partir de zéro ne fonctionne jamais et ne peut être rapiécé pour qu'il fonctionne.

Il faut tout recommencer, à partir d'un système simple qui fonctionne. » - John Gall

Cycle iter&inc: *difficultés*

Pour chaque version à développer après la 1ère version livrée, il faut **arbitrer** entre:

- les demandes de **correction**,
- les demandes de **modification** et
- les **nouvelles** fonctionnalités à développer.

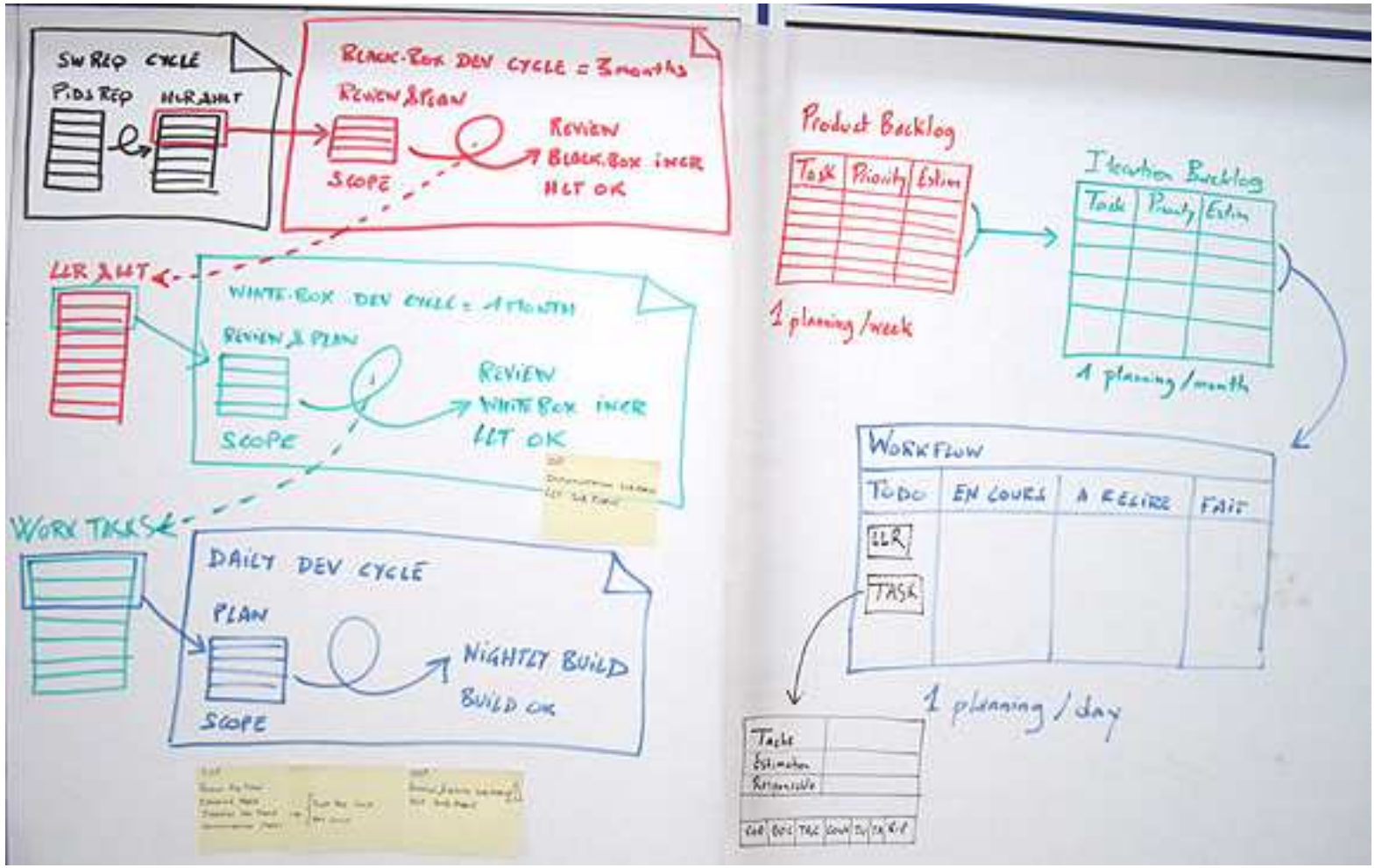


Cycle iter&inc: *difficultés*

Comment arbitrer?

- Il est malsain de vivre avec des défauts identifiés (ou pas …)
- Ajouter des fonctionnalités sur un code que l'utilisateur souhaite modifier entrainera des changements en cascade.
- Laissez le client choisir …

Cycle iter&inc: *exemple*



Que met-on dans un incrément?

PRATIQUE: **Pilotez le développement par les besoins du client**

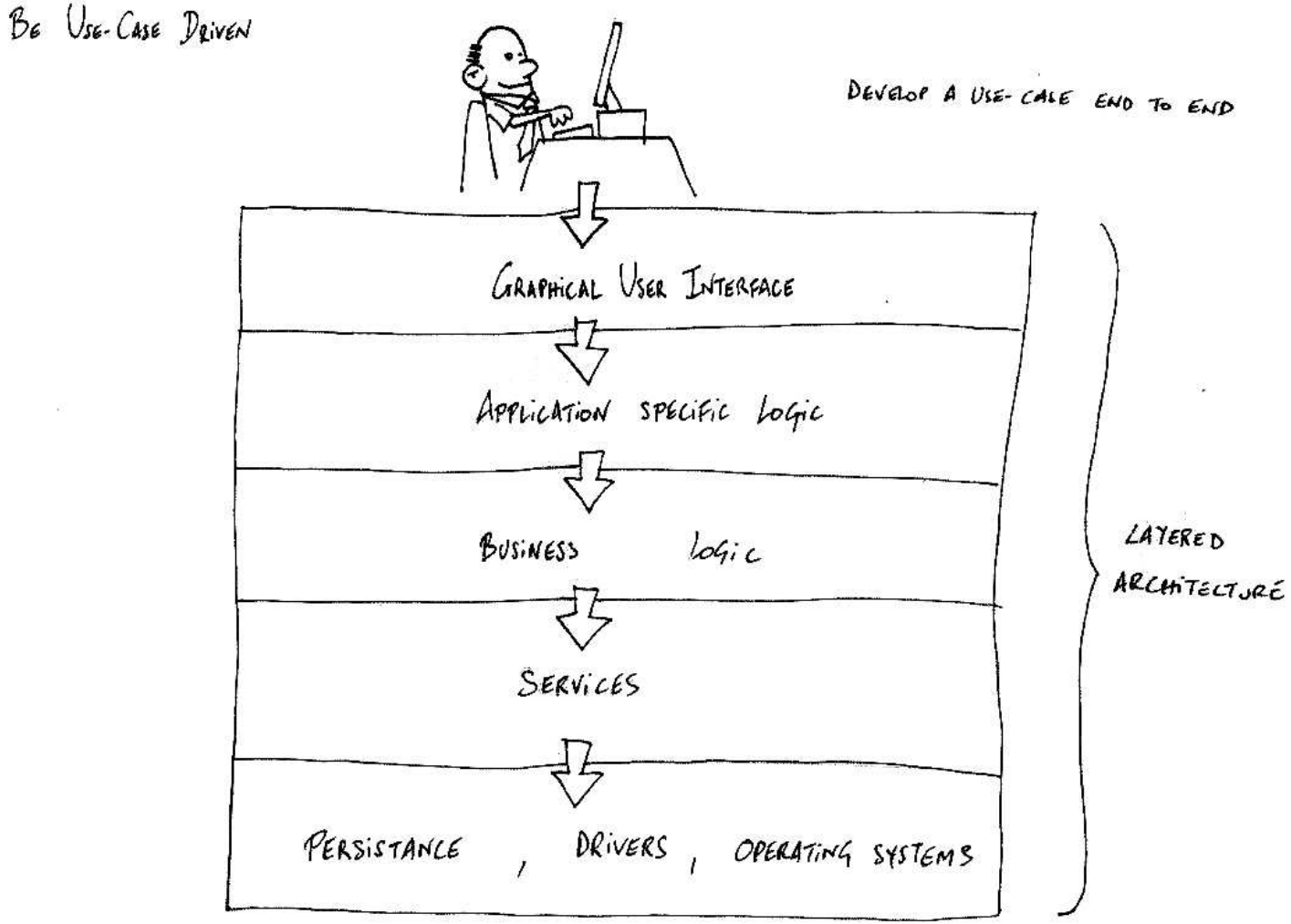
Développez des incréments de fonctionnalités de bout en bout.

Fonctionnalité du produit = Besoin du client.

Les besoins peuvent être exprimés par des exigences, des cas d'utilisation, des scénario, etc …

Laissez le client piloter.

Un besoin de bout en bout



Dans quel ordre faut-il réaliser les itérations?

PRATIQUE: **Pilotez le développement par les priorités**
Commencez par développer les fonctionnalités les plus importantes pour le client et les plus risquées.

Analogie: « Un tiens vaut mieux que deux tu l'auras. »

Pensez à la règle des 80/20!

Comment savoir qu'une itération est terminée?

PRATIQUE: **Pilotez le développement par les tests d'acceptation**
Écrivez des tests automatisés d'acceptation.
Utilisez ces tests pour mesurer l'avancement du développement.

Il n'y a pas de meilleure mesure de l'avancement que les fonctionnalités **s'exécutant** conformément aux besoins du client.

Une fonctionnalité s'exécute conformément au besoin du client si elle **réussit ses tests d'acceptation**.

Les tests d'acceptation sont ceux qui font le plus avec le moins.

Cycle iter&inc*: difficultés*

Un cycle itératif et incrémental implique que l'on **retouche souvent** au même code:

ATTENTION AUX REGRESSIONS!

Heureusement, on peut les éviter grâce aux **tests**.

Comment construire bien et sans régression?

PRATIQUE: **Pilotez le développement par les tests**
Développez en cycles très courts: ajoutez un test qui échoue, puis faites le réussir.
Dès qu'un bug est détecté, commencez par écrire un test qui révèle le défaut.
Écrire les tests avant le code est un outil de conception: cela conduit à une conception plus pragmatique, plus simple et moins couplée.
Assurez vous que l'exécution des tests est automatisée, qu'ils vérifient leurs propres résultats et sont jouées par un outil d'intégration continue sur toutes les plateformes et pour tous les environnements visés.

Les tests sont destinés à éviter les défauts, pas les trouver!

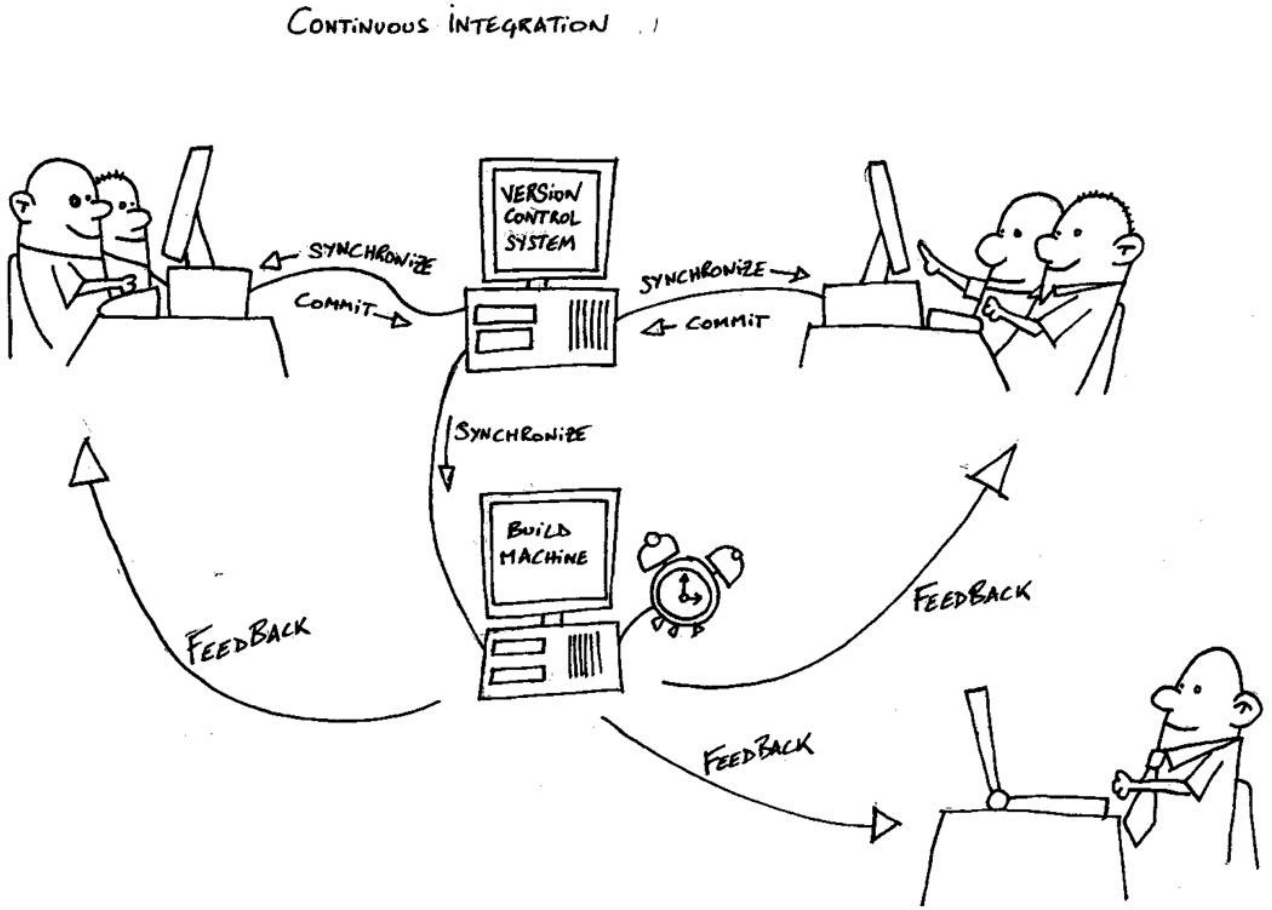
L’intégration est risquée *1/5*

PRATIQUE: **Intégrez continuellement**

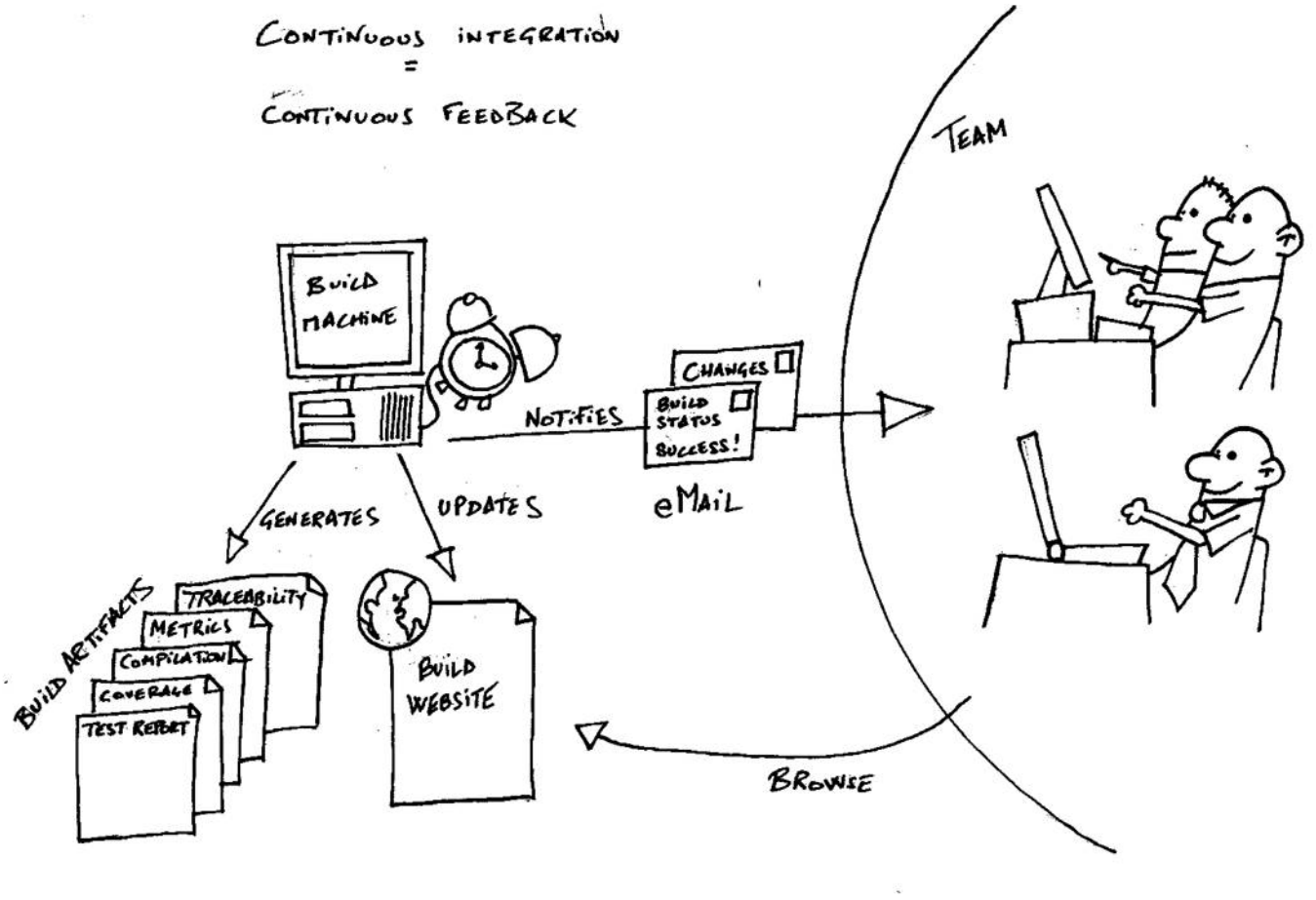
Entretenez tôt et toujours une version du logiciel qui soit compilable, testable, livrable, installable et comprenant les tous derniers développements.

Évitez les stocks, préférez le flux tendu.

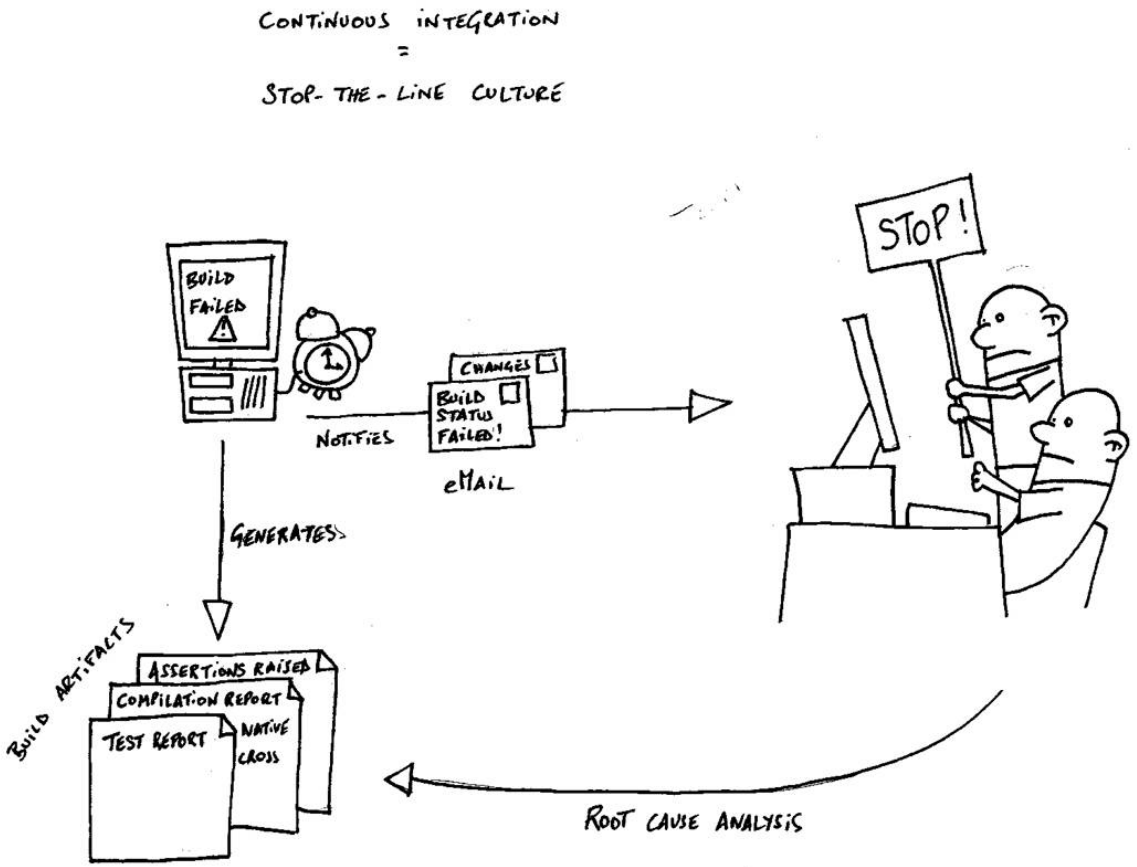
Intégration continue *2/5*



Intégration continue *3/5*



Intégration continue *4/5*



Intégration continue *5/5*

CruiseControl.rb
Continuous Integration for Ruby

adiru — Build Now

build 1615 (10:37) took 39 seconds
1613 (10:15)
1611 (9:46)
1610 (9:44) FAILED
1609 (9:08)

adiru-nightly — Build Now

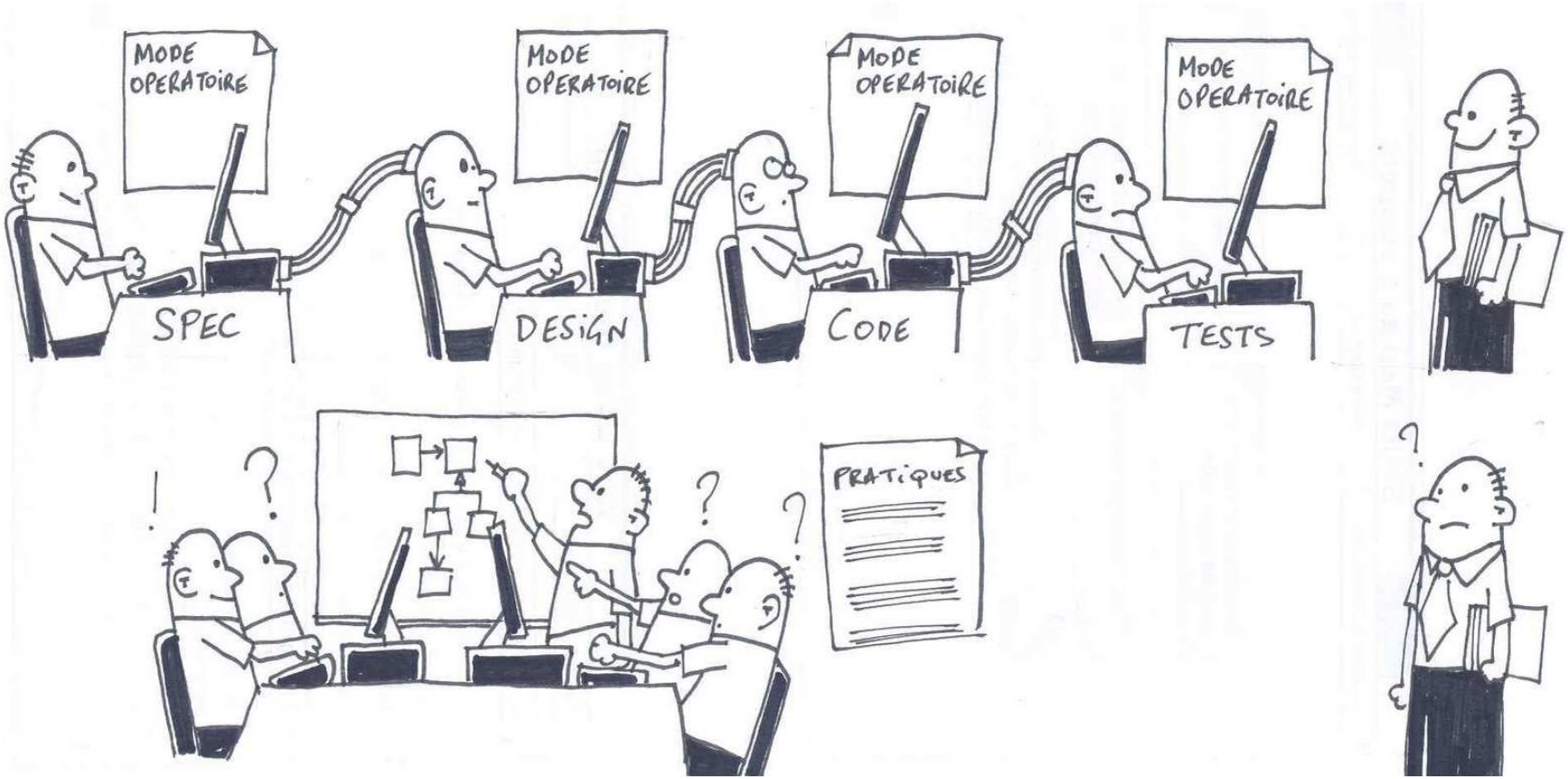
build 1606 (6:11) took 11 minutes
1599 (8 Jul)
1593 (8 Jul)
1499 (8 Jul)
1497.1 (8 Jul)

Et hop! Sans les mains ;o)

PRATIQUE: **N'utilisez pas de procédure manuelle**
Automatisez *les procédures de production, de test, de livraison, d'installation afin de gagner en productivité et d'éviter les erreurs humaines.*

Mais n'automatisez que si la procédure sera déroulée plus d'une fois …

Penser et/ou faire?



Gestion de configuration *1/5*

PRATIQUE: **Utilisez un outil de gestion de configuration**
Gérez en configuration tout ce qui sert à produire le logiciel (code, tests, outils, documentation, procédures automatisées).
Développez dans un espace privé.
Ne soumettez à la base commune que des éléments qui ne la font pas régresser.

La machine à remonter le temps. La mémoire du projet.

« L'encre la plus pâle est meilleure que la meilleure mémoire. »
Proverbe chinois.

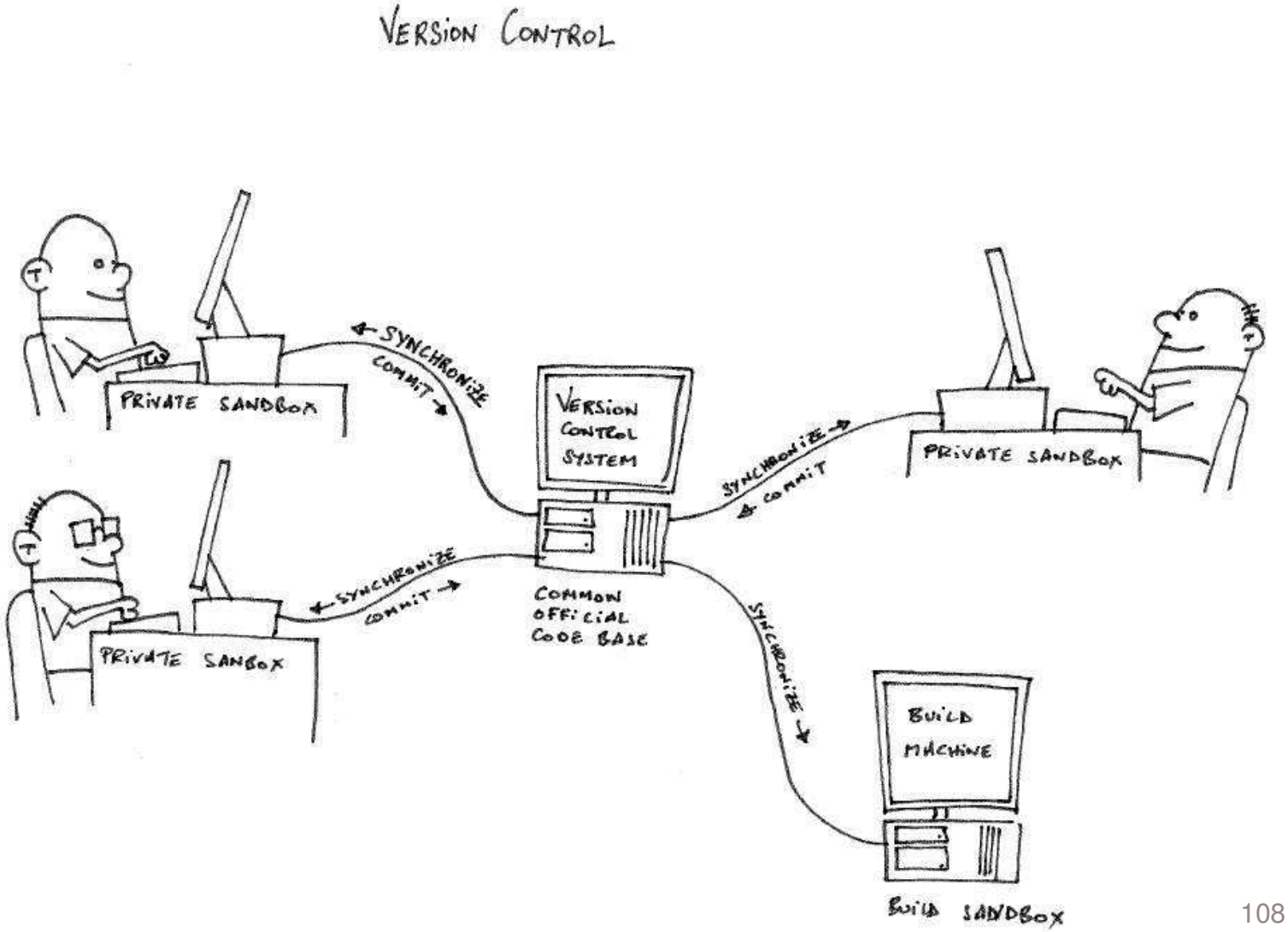
Une discipline nécessaire pour développer en équipe.

Gestion de configuration *2/5*

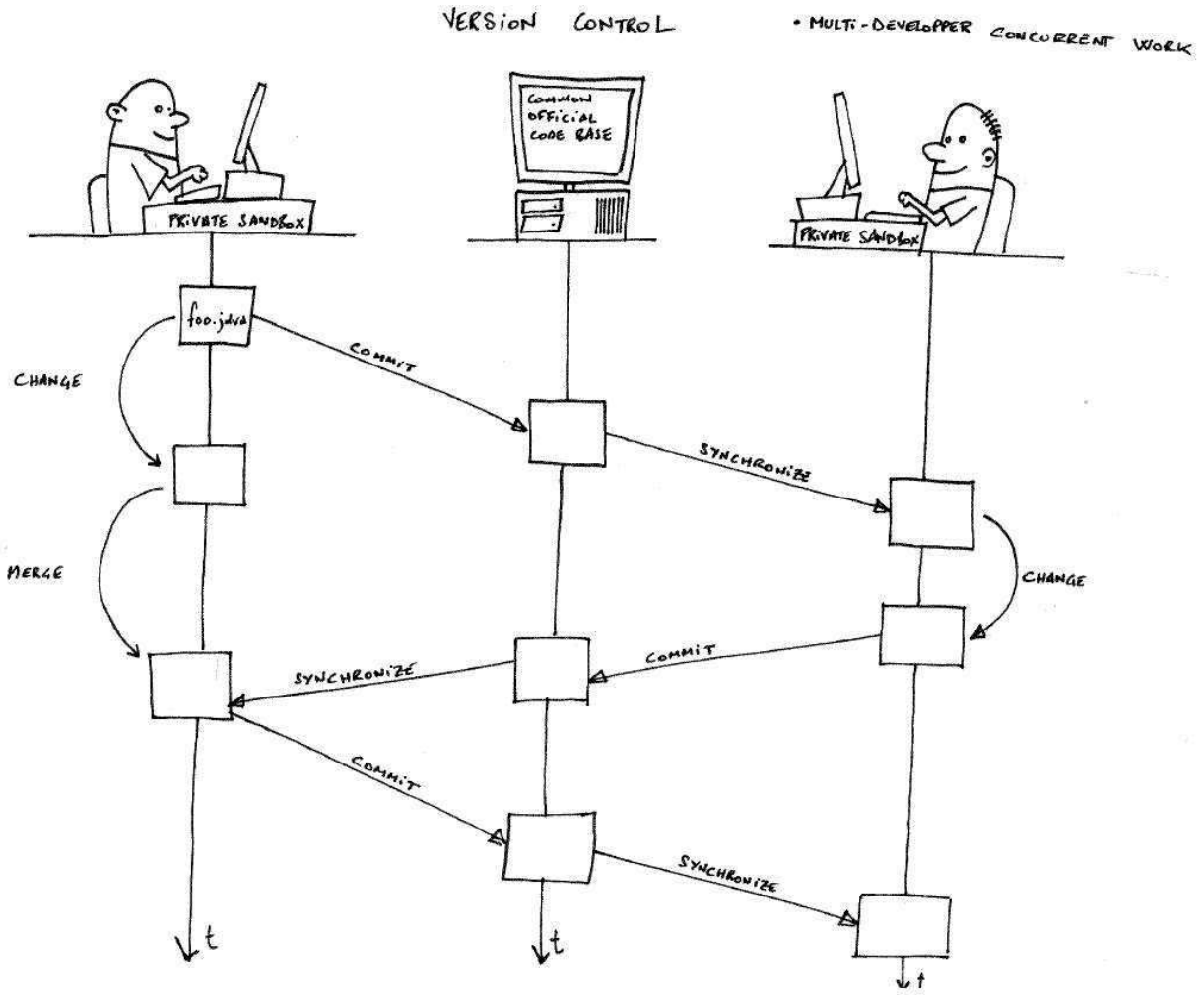
Un outil de gestion de configuration permet de:

- Identifier et archiver les éléments de la configuration
  - code, tests, documentation, scripts de production …
- Tracer et archiver les changements dans la configuration
- Identifier et archiver les versions de la configuration
- Gérer le travail concurrent à plusieurs développeurs sur les éléments de la configuration

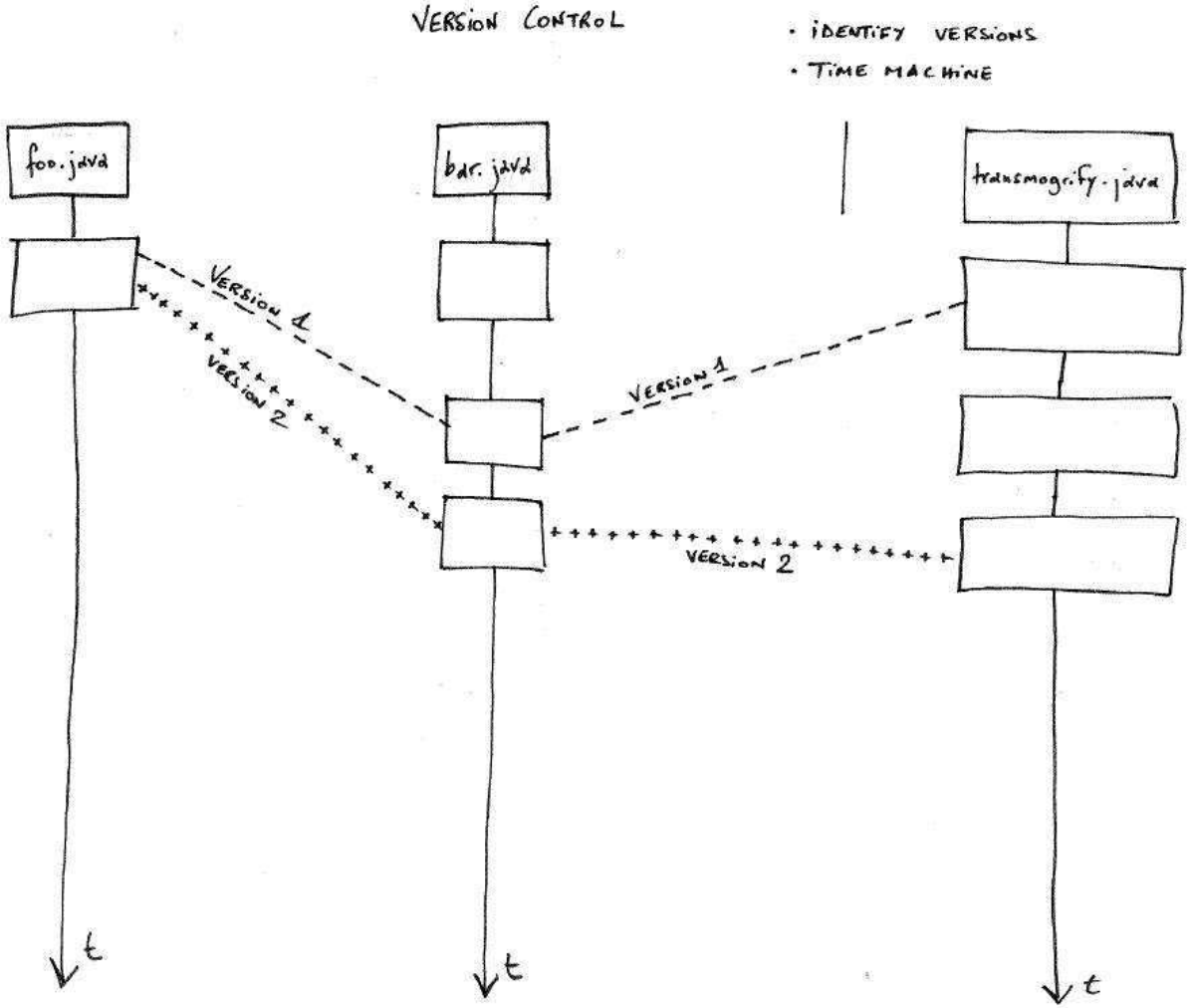
Gestion de configuration *3/5*



Gestion de configuration *4/5*



Gestion de configuration *5/5*



Tout doux

PRATIQUE: **Avancez à petits pas**

Travaillez en petits cycles:
[synchroniser / coder / compiler / tester / livrer].

- Pour avancer en limitant les risques.
- Pour intégrer en douceur les travaux en parallèle.
- Avec une gestion de configuration, pour revenir en arrière si besoin est.
- Démarche itérative et incrémentale appliquée à la journée du développeur.

Gestion des faits techniques *1/4*

PRATIQUE: **Utilisez un outil de gestion des faits techniques**
Évitez l'amnésie collective.
Maintenez une liste de problèmes et de solutions.

« L'encre la plus pâle est meilleure que la meilleure mémoire. »
Proverbe chinois.

« Savoir, c'est se souvenir. » Aristote.

Gestion des faits techniques *2/4*

Un fait technique peut être:

- Un défaut (bug)
- Une demande d'évolution
  - Changement de spécification
- Une décision importante
  - Choix technique déterminant

Gestion des faits techniques *3/4*

Un outil de gestion des faits techniques permet de:

- Identifier les faits techniques
  - Identifiant, description, type (défaut, changement, décision …)
- Tracer et archiver le traitement des faits techniques
  - État courant (nouveau, en cours de traitement, clos), décisions prises pour traiter le fait technique, impact sur le logiciel …

Gestion des faits techniques *4/4*

Gestion de configuration et **gestion des faits techniques** sont 2 **pratiques très couplées**.

En utilisant des outils couplés, on peut:

- Identifier la version concernée par le fait technique;
- Identifier les changements dans la configuration liés au traitement du fait technique;
- Identifier la version pour laquelle le fait technique est clos.

DRY: D*on't* **R***epeat* **Y***ourself*

PRATIQUE: **Ne vous répétez pas**

*Éliminez **toute** duplication de code et d'information.*

Code, valeurs (constantes), …

Une question de vocabulaire

PRATIQUE: **Soignez votre vocabulaire**
*Construisez et maintenez un **glossaire** du vocabulaire du produit.*
*Concevez, codez et testez en utilisant le **vocabulaire des utilisateurs**.*

« Retirer de l'argent » n'est pas la même chose que « Débiter quantité de devise du compte principal client ».

Un travail d'équipe *1/20*

Constats:

- L'équipe est au cœur d'un projet de développement de logiciel.
- Pas de réussite sans travail d'équipe!
- Équipe /= groupe
- Travail(équipe) > somme(travail individuel)

Problème:

- Comment construire une équipe efficace?

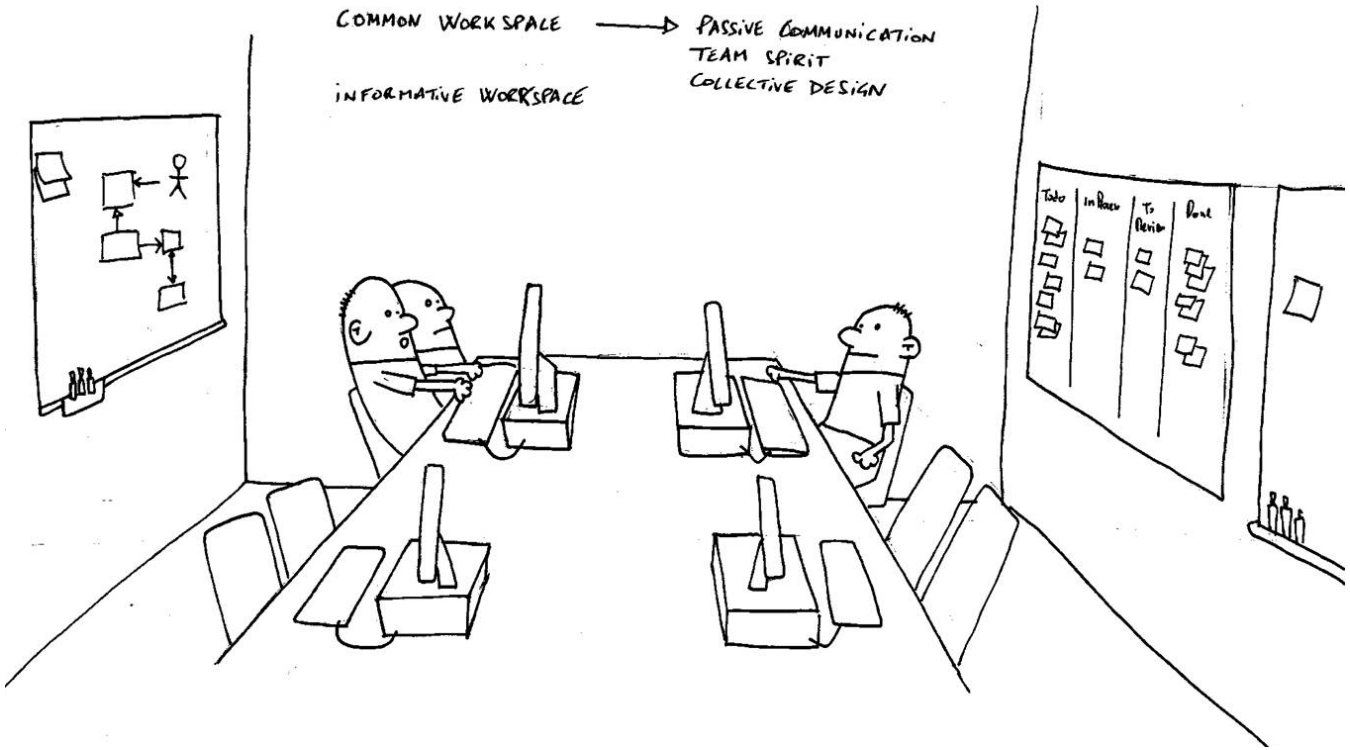
Un travail d'équipe *2/20*

Pour construire une équipe efficace:

- Pratiques d'équipe;
- Espace de travail adapté aux pratiques d'équipe;
- Management adapté aux pratiques d'équipe;
- Des équipiers!

Un travail d'équipe *3/20*

Partagez un espace commun au projet



Un travail d'équipe *4/20*

Partagez un espace commun au projet

Un travail d'équipe *5/20*

Un travail d'équipe *6/20*

Un travail d'équipe *7/20*

Ce qui est important est visible.

Un travail d'équipe *8/20*

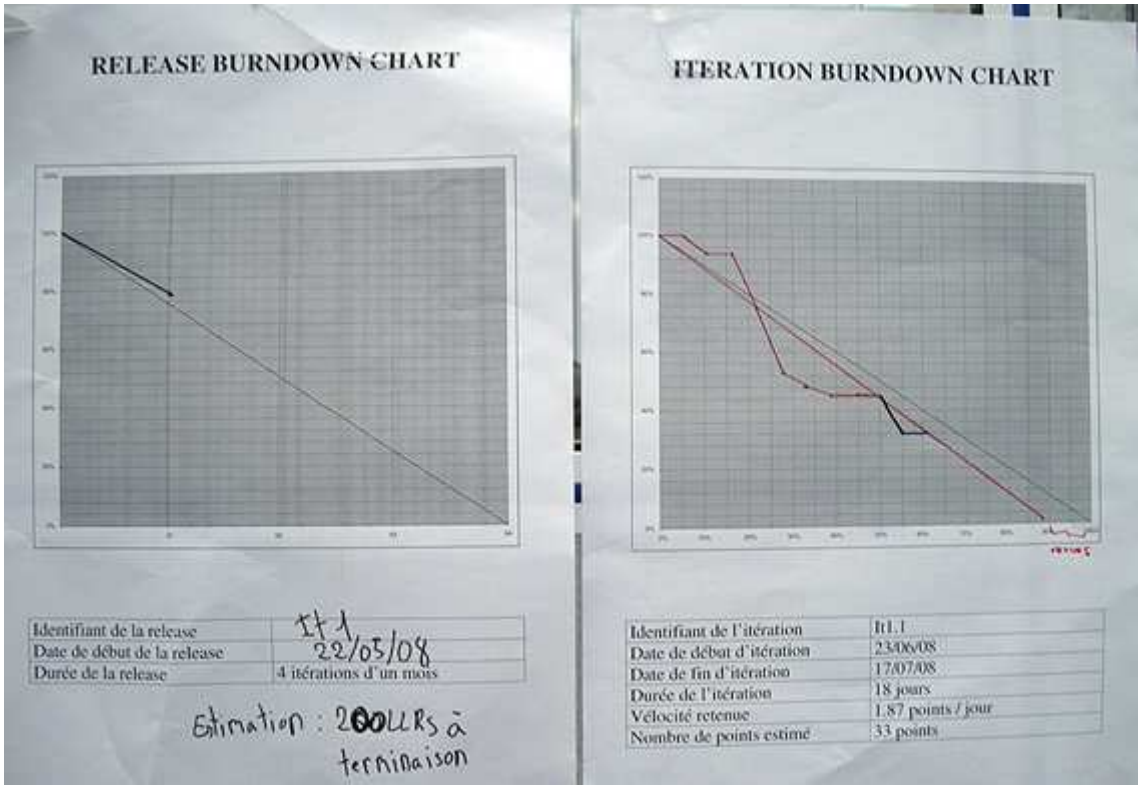
L'équipe se pilote elle-même: le flux-tendu par Kanban

Un travail d'équipe *9/20*

L'équipe reste synchronisée

Un travail d'équipe *10/20*

L'équipe sait où en est le projet



RELEASE BURNDOWN CHART

Identifiant de la release	It 1
Date de début de la release	22/05/08
Durée de la release	4 itérations d'un mois

Estimation : 200LLRs à terminaison

ITERATION BURNDOWN CHART

Identifiant de l'itération	It1.1
Date de débat d'itération	23/06/08
Date de fin d'itération	17/07/08
Durée de l'itération	18 jours
Vélocité retenue	1.87 points / jour
Nombre de points estimé	33 points

Un travail d'équipe *11/20*

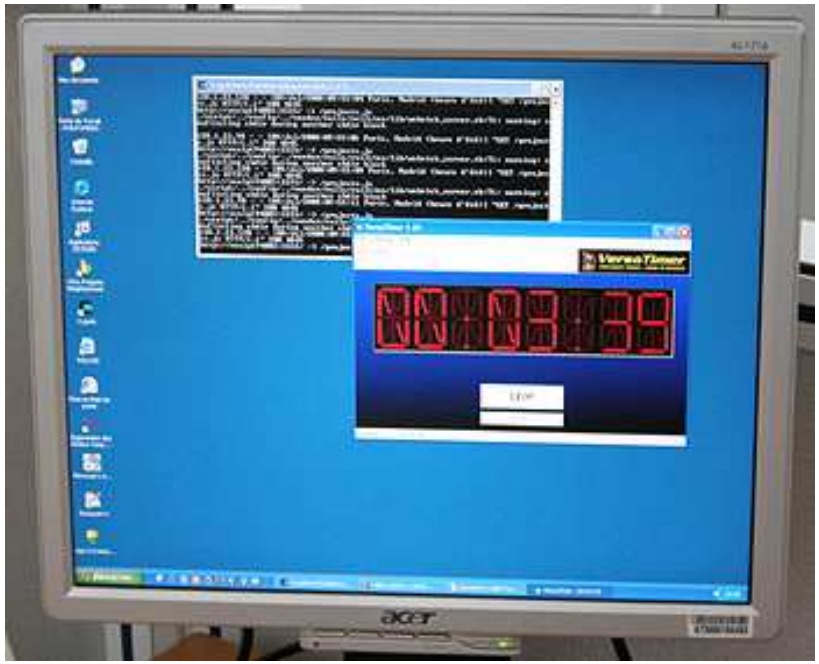
L'équipe fait prend soin d'elle

Un travail d'équipe *12/20*

L'équipe a du rythme - Gizmo!

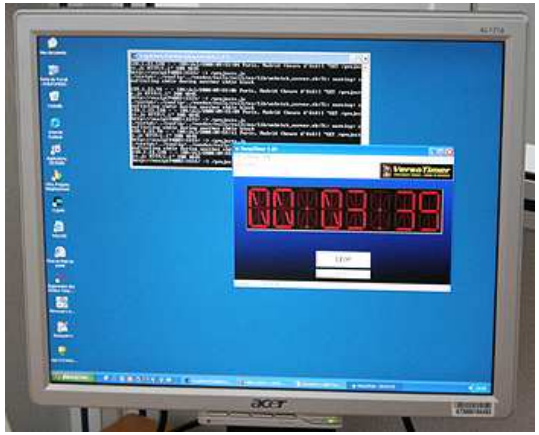
Un travail d'équipe *13/20*

L'équipe a du rythme - Timebox

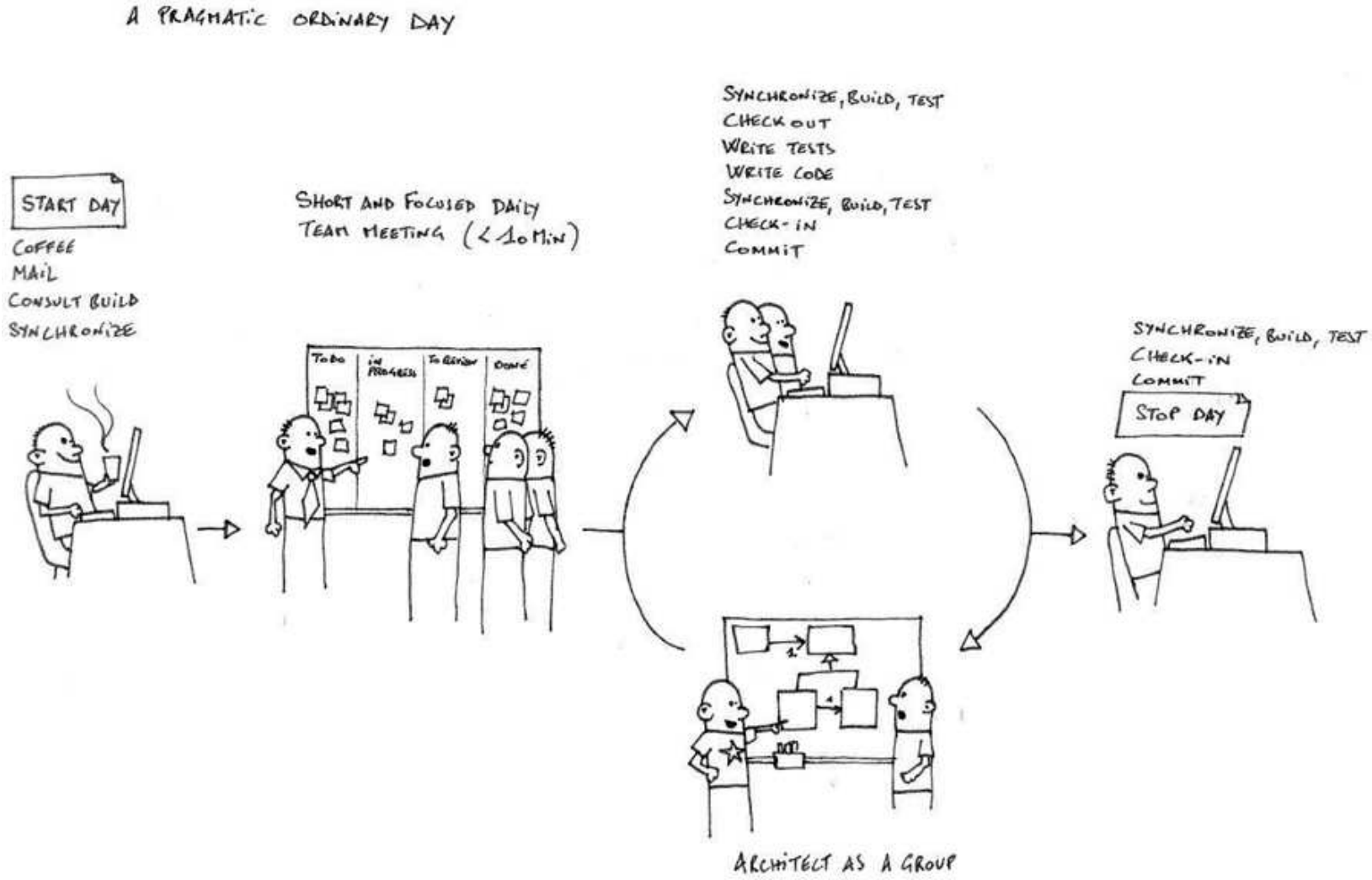


Un travail d'équipe *14/20*

L'équipe a du rythme : le groove



Un travail d'équipe *15/20*



Un travail d'équipe *16/20*

PRATIQUE: **Développez une propriété collective du code**
N'importe quel développeur de l'équipe peut (et se doit d'être prêt à) modifier n'importe quel code de l'application.
Permutez les développeurs sur les zones du code, les fonctionnalités du produit et les activités à réaliser.

Objectif: les développeurs sont **polyvalents** sur leur projet.

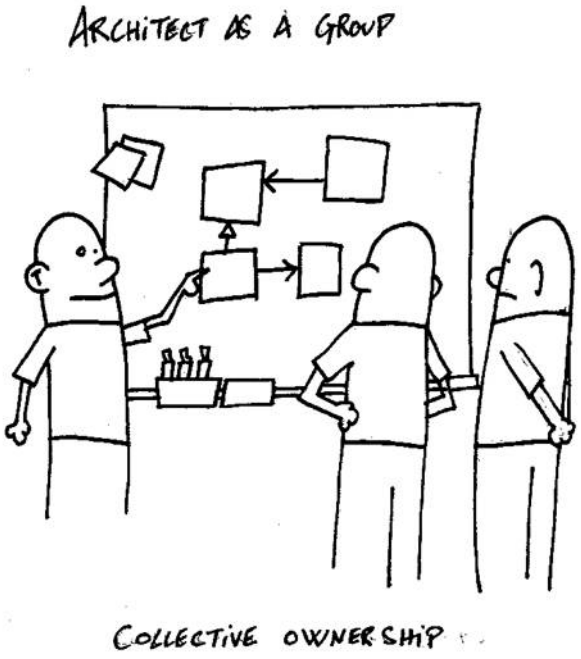
Mais ne veut pas dire que personne n'est responsable!

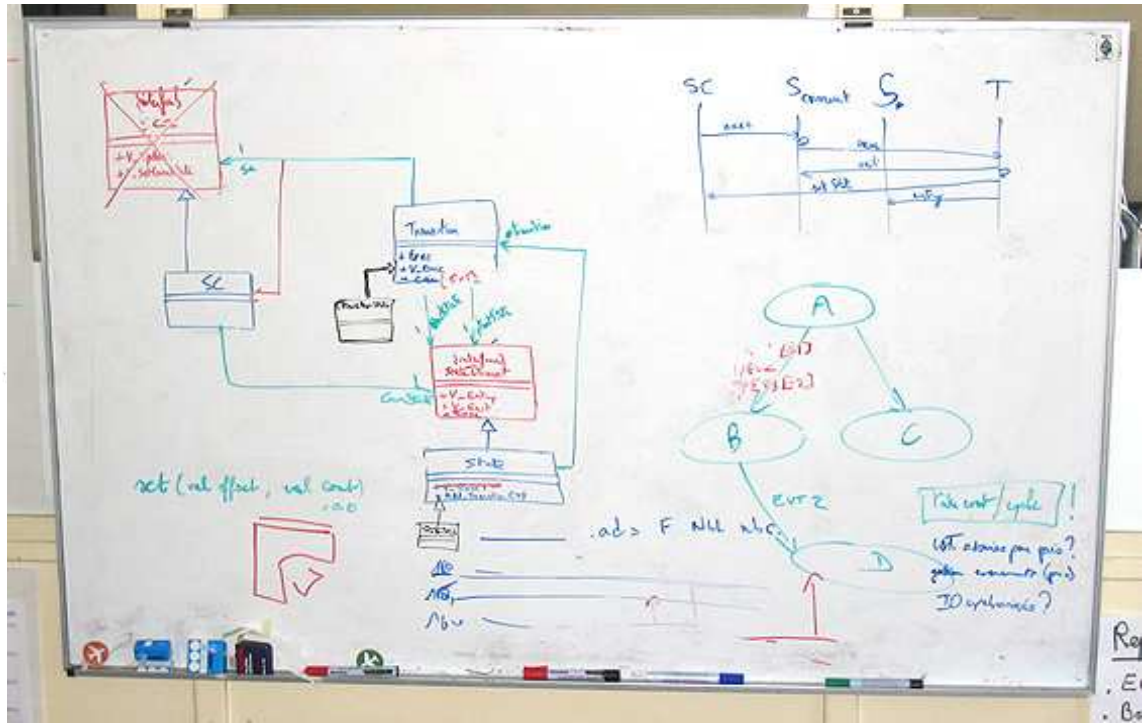
Un travail d'équipe *17/20*

PRATIQUE: **Concevez en équipe**
Prenez les décisions importantes de conception à plusieurs.

« Je ne fais pas qu'utiliser tous les neurones que j'ai, mais aussi tous ceux que je peux emprunter. » Woodrow Wilson (Président des Etats-Unis).

Un travail d'équipe *18/20*





Utilisez un langage formel pour communiquez vos décisions de conception *(ex: UML)*

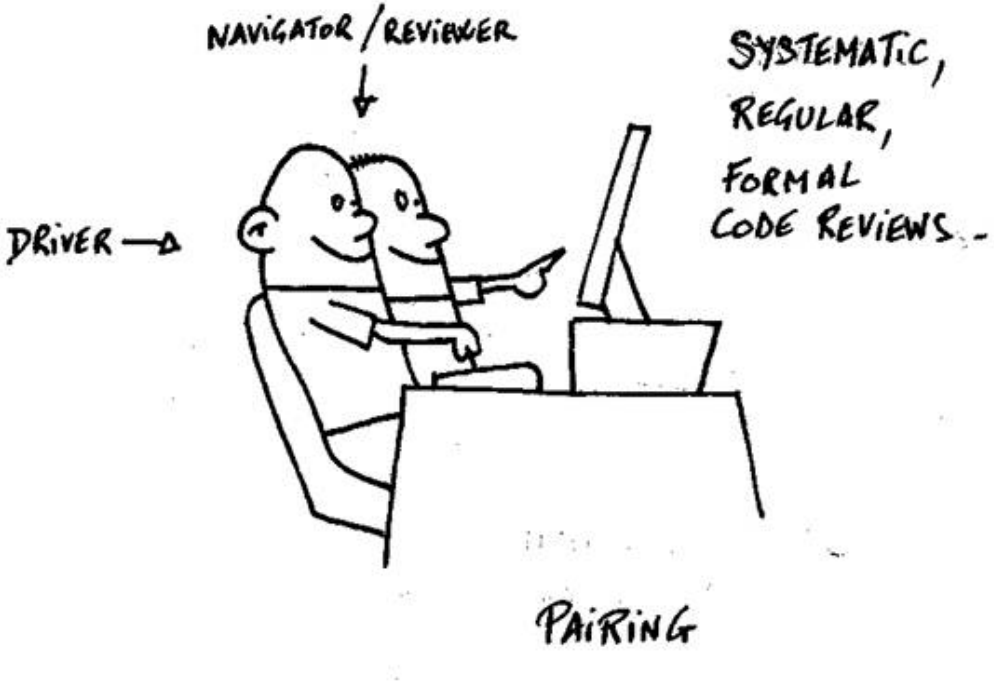
Un travail d'équipe *19/20*

PRATIQUE: **Relisez-vous les uns les autres**
*Relisez **tout**.*
*Travaillez en **binômes** en permutant les paires.*

« Deux paires d'yeux valent mieux qu'une. »

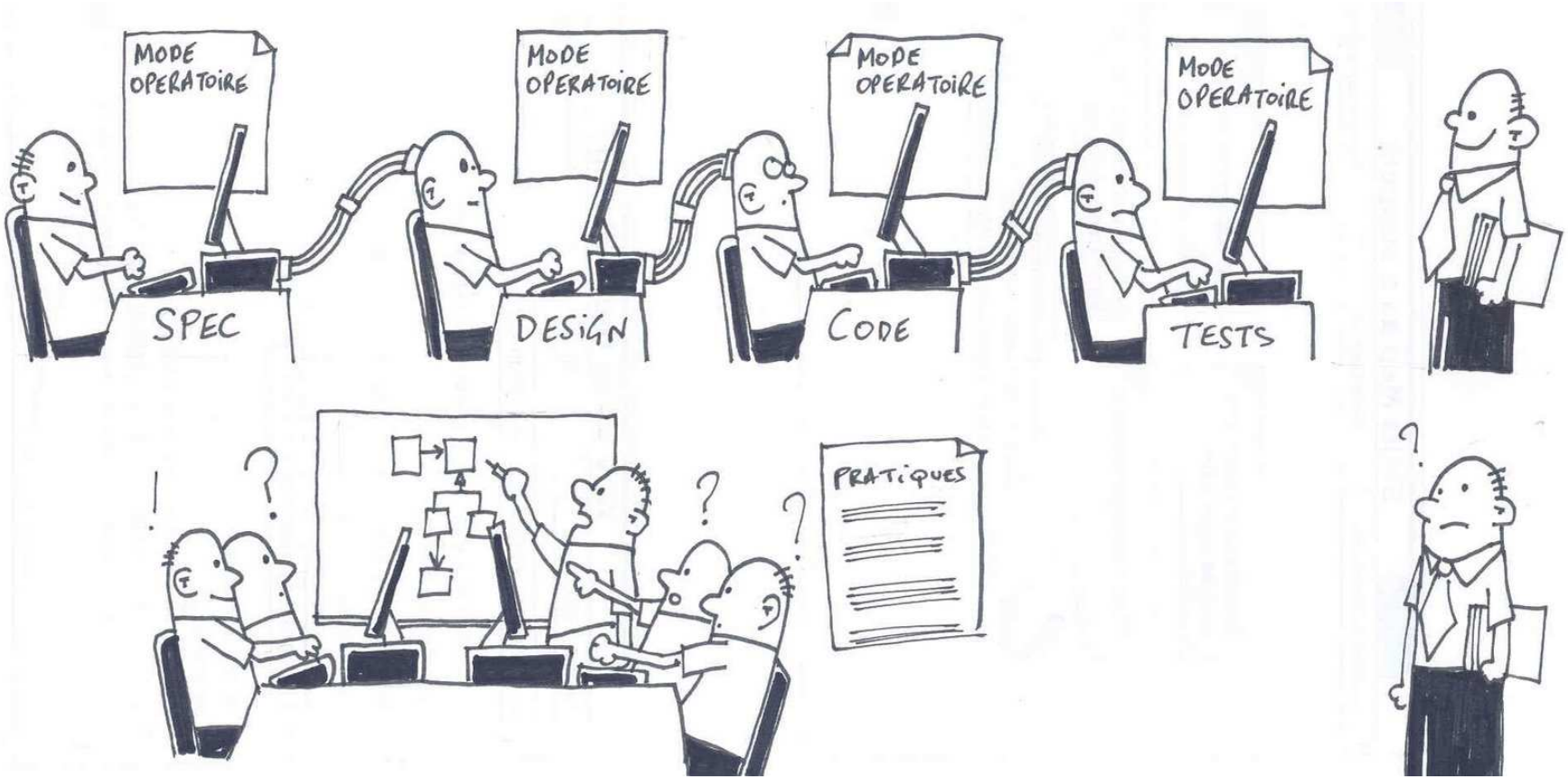
Travail en binôme: relecture continue, relire sans stock

Un travail d'équipe *20/20*



2 styles d'équipe

Penser et/ou faire? *(bis)*



Génie Logiciel

PRATIQUE: **Gare aux outils et aux technologies**
Les outils couteux ne font pas de meilleurs logiciels.
Ne mettez pas de technologie non-maitrisée sur le chemin critique.

Il vaut mieux faire briller son CV en citant des projets réussis qu'en citant des technologies et des outils « maitrisés ».

KISS: K*eep* **I***t* **S***imple* **S***oftware*

PRATIQUE: **Optez pour la simplicité**
*Développez la solution la plus simple qui soit **viable**.*
L'équipe entretien une conception strictement adaptée aux fonctionnalités actuelles du produit. *Le logiciel passe avec succès tous ses tests, ne contient aucune duplication, exprime clairement l'intention des développeurs et contient aussi peu de code que possible.*

« La perfection est atteinte pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer ». St Exupéry.

« Il y a deux manière de créer une conception de logiciel. Une manière est de la rendre si simple qu'il n'y a évidemment aucun défaut. Et l'autre est de la rendre si complexe qu'il n'y a aucun défaut évident. » C.A.R. Hoare.

Mais simple n'est pas simpliste …

C’est clair!

PRATIQUE: **Soyez clairs**
Écrivez un code clair, pas un code rusé.
Communiquez en code source et réservez les commentaires pour justifier des décisions complexes.
Écrivez un code qui puisse être compris par un humain, puis par un compilateur ou un interpréteur.

Chaque ligne de code sera plus souvent lue que écrite.

« Ce que l’on conçoit bien s’énonce clairement, et les mots pour le dire arrivent aisément». Boileau.

« La première qualité du style, c’est la clarté. » Aristote.

Ça passe …

PRATIQUE: **Gare à l'optimisation**
*N'optimisez pas **prématurément** votre code.*
Il est plus facile de rendre un code correcte rapide que de rendre un code rapide correcte.

« Ne cravachez pas un cheval consentant. » Proverbe latin.

Mais ayez conscience des ressources que vous consommez.

La cause, pas de symptôme *(1/2)*

PRATIQUE: **Attaquez les problèmes à la racine**
Cherchez à comprendre pourquoi.
Ne faites pas de correction rapide. Cherchez l'origine du problème et corrigez le proprement à la racine.

« De verrue en verrue, on produit un chou-fleur. »

« Oui, mais pourquoi ce pointeur est-il nul? »

Les 5 pourquoi.

La cause, pas de symptôme *(2/2)*

On surprend un ouvrier en train de épandre de la sciure de bois dans le couloir entre deux machines:

Q: Pourquoi répandez-vous de la sciure? R: Parce que le plancher est glissant et peut présenter un danger. Q: Pourquoi le plancher est-il glissant et peut-il présenter un danger? R: Parce qu'il y a de l'huile sur le plancher. Q: Pourquoi y a-t-il de l'huile sur le plancher?	R: Parce qu'elle fuit le la machine. Q: Pourquoi fuit-elle de la machine? R: Parce que l'huile s'échappe du cardan à huile. Q: Pourquoi fuit-elle du cardan? R: Parce que le revêtement en caoutchouc à l'intérieur du cardan est usé.

Standardisez

PRATIQUE: **Faites votre standard**

Tout le code semble avoir été écrit par une seule, même et très compétente personne.

Mais ne standardisez pas les goûts et les couleurs …

Ne pas figer les plans

PRATIQUE: **Laissez la conception guider, pas dicter**
La conception est un plan qui doit évoluer.

« Aucun plan ne survit au contact avec l'ennemi. » Helmut Von Moltke.

« En préparant une bataille, j'ai toujours trouvé que les plans sont inutiles, mais la planification est indispensable. » Dwight Eisenhower.

La bataille de la Somme: suivre un plan coûte que coûte peut coûter très cher …

Refactoring

PRATIQUE: **Remaniez votre logiciel**
Corrigez les mauvaises conceptions, les mauvaises décisions et le mauvais code dès que vous les détectez.
Lorsque vous avez à ajouter une fonctionnalité à un logiciel et que le code n'est pas organisé de manière à l'accueillir, commencez par remanier le logiciel pour que la fonctionnalité devienne facile à ajouter. *Ensuite, ajoutez la fonctionnalité.*

Remanier: **changer** le code **sans changer** les fonctionnalités et le comportement du logiciel.

Détrompez-vous *1/6*

PRATIQUE: **Détrompez votre code**
*Concevez par **contrats**: Utilisez des contrats pour documenter le logiciel et vérifier que le code ne fait ni plus ni moins que ce qu'il annonce.*
*Utilisez des **assertions** pour valider vos postulats et contrats.*
Utilisez le niveau d'avertissement le plus strict du compilateur. *Exigez des constructions sans avertissement.* *Éliminez les avertissements en changeant le code, non en réduisant la sévérité du compilateur.*
Écrivez vos tests au plus tard en même temps que le code et rejouez les aussi souvent que possible.

Détrompez par assertion *2/6*

Une **assertion** représente un énoncé considéré ou présenté comme vrai.

```
double altitude = getAltitude();
assert altitude >= 0.0 : "Altitude is positive";
double res = Math.sqrt(altitude);
```

En programmation, une assertion erronée interrompt l'exécution du programme:

```
run:
java.lang.AssertionError: Altitude is positive
at cours.Main.main(Main.java:30)
```

« Une violation d'assertion à l'exécution est la manifestation d'un bogue dans le logiciel. » Bertrand Meyer

Détrompez par contrat *3/6*

Soit une routine **r**.

Le **contrat** de **r** est sa **pré-condition** et sa **post-condition**.

*« Si vous promettez d'appeler **r** avec une pré-condition vérifiée, en retour, je promets de délivrer un état final dans lequel la post-condition est vérifié. » Bertrand Meyer.*

```
static double sqrt(double num) {
        assert num >= 0.0 : "precond num must be positive";
        final double res = java.lang.Math.sqrt(num);
        assert (res * res) == num : "postcond";
        return res;
    }
```

Détrompez par contrat *4/6*

« Une violation de pré-condition est la manifestation d'un bogue chez le client. » Bertrand Meyer.

```
res = Math.sqrt(4.0);
res = Math.sqrt(-4.0);
run:
java.lang.AssertionError: precond num must be positive!
        at cours.Math.sqrt(Math.java:19)↰
        at cours.Main.main(Main.java:34)↰
Exception in thread "main"
Java Result: 1
```

Le bug se trouve chez l'appelant de cours.Math.sqrt().

Détrompez par contrat *5/6*

« Une violation de post-condition est la manifestation d'un bogue chez le fournisseur. » Bertrand Meyer.

```
run:
java.lang.AssertionError: postcond!
        at cours.Math.sqrt(Math.java:22)↰
        at cours.Main.main(Main.java:33)↰
Exception in thread "main"
Java Result: 1
```

Le bug se trouve dans cours.Math.sqrt().

Détrompez par test *6/6*

```
public void testSqrt() {
        System.out.println("sqrt");

        assertEquals(Math.sqrt(0.0), 0.0);
        assertEquals(Math.sqrt(4.0), 2.0);
}
```

Si ce test est écrit avant l'implémentation de Math.sqrt(), alors l'implémentation sera correcte.

Si ce test est rejoué aussi souvent que possible, alors on détectera toute régression de Math.sqrt().

Le Manifeste Agile

Les 4 Valeurs

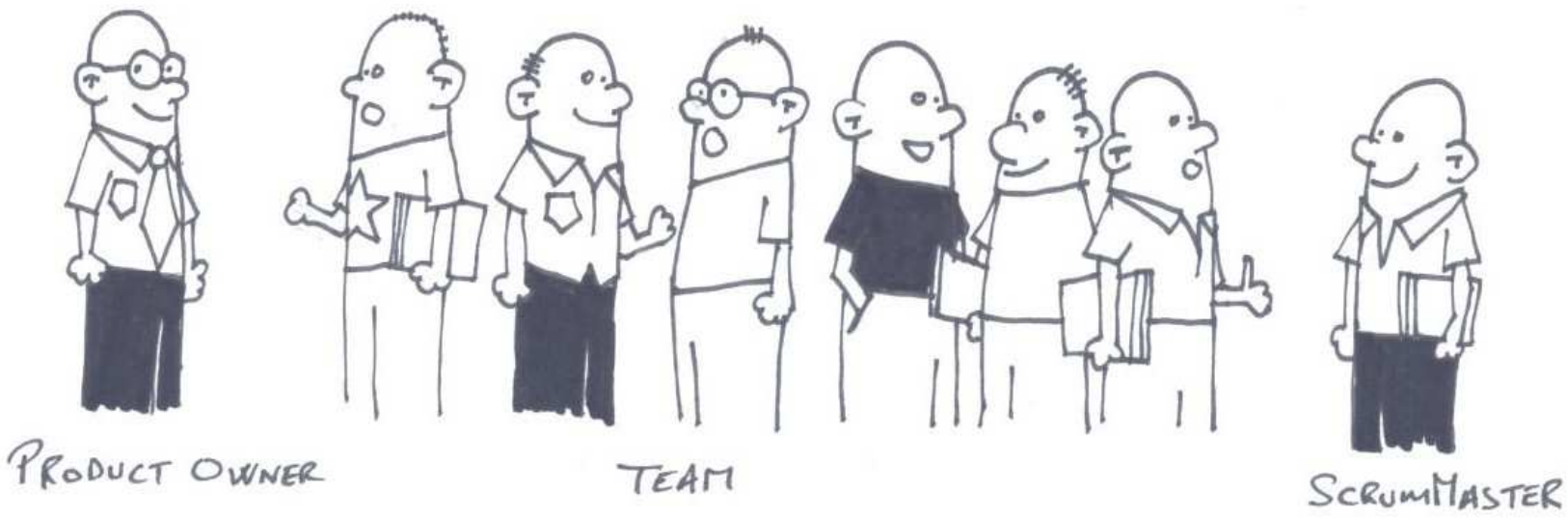
- **Les personnes et les interactions** *plutôt que les processus et les outils.*
- **Un logiciel opérationnel** *plutôt qu'une documentation exhaustive.*
- **La collaboration avec le client** *plutôt que la négociation du contrat.*
- **Réagir au changement** *plutôt que le suivi d'un plan.*

Le Manifeste Agile

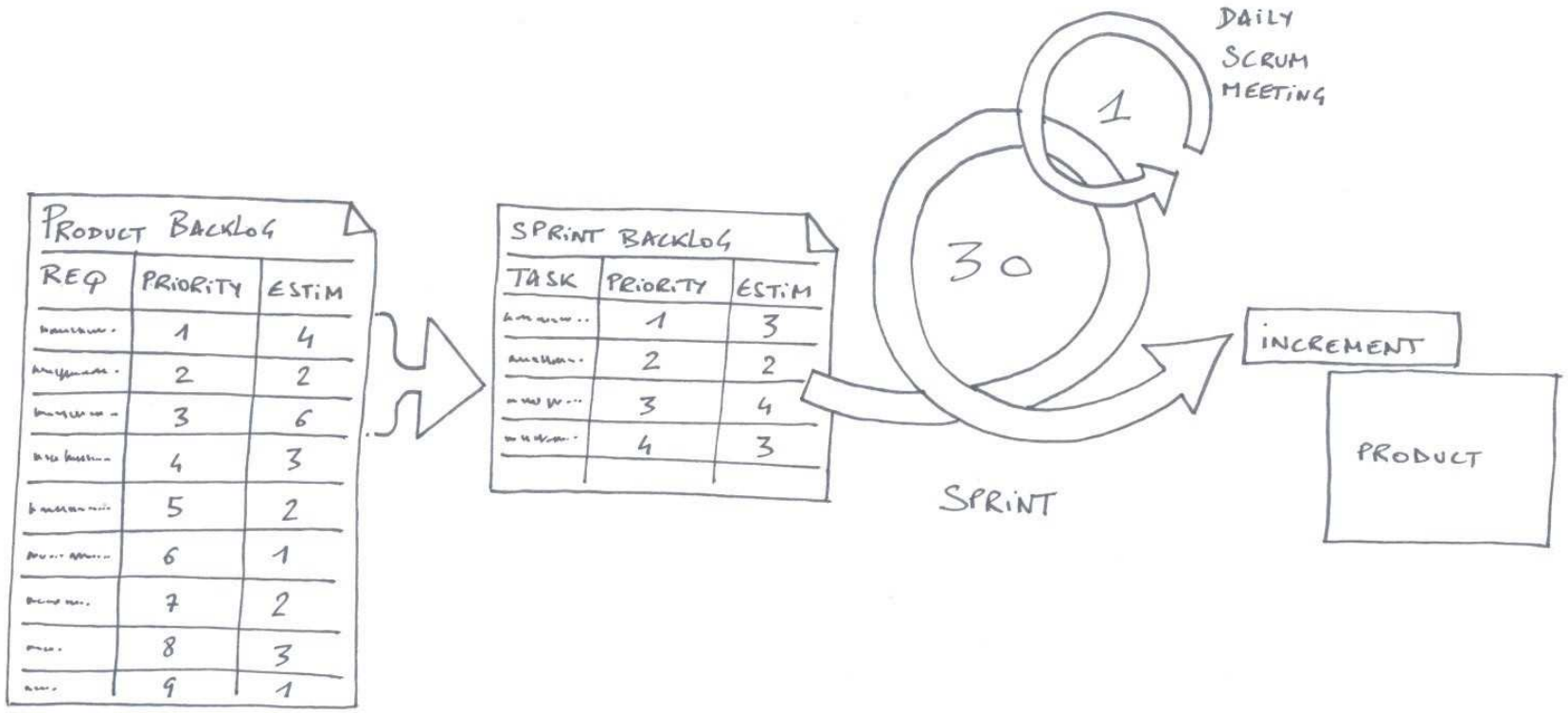
Les 12 principes

- Notre priorité est de satisfaire le client par des livraisons rapides et continues de logiciel utile.
- Intégrer les changements aux exigences même s'ils arrivent tard dans le processus de développement. Les méthodes Agiles intègrent rapidement les changements de façon à offrir un avantage compétitif au client.
- Livrer fréquemment du logiciel opérationnel, soit à toutes les quelques semaines ou quelques mois en visant de courts délais.
- Les clients et les développeurs doivent travailler main dans la main quotidiennement tout au long du projet.
- Élaborer des projets autour d'individus motivés. Leur procurer l'environnement et le support nécessaire et leur faire confiance pour réaliser le travail.
- La façon la plus efficace de transmettre l'information à une équipe et entre les membres est par des conversations en face à face. Le logiciel opérationnel est la principale mesure de progrès
- Agile favorise le développement à rythme "normal".
- Les gestionnaires, développeurs et utilisateurs devraient être en mesure de maintenir un rythme constant et ce, indéfiniment.
- Porter une attention continue à l'excellence technique et à un bon design améliore l'agilité.
- La simplicité - l'art de maximiser la quantité de travail non fait - est essentielle.
- Les meilleures architectures, exigences et designs prennent naissance dans des équipes qui se gèrent elles-mêmes.
- Régulièrement, l'équipe fait une réflexion sur les façons de devenir plus efficace, s'ajuste et modifie son comportement en conséquence.

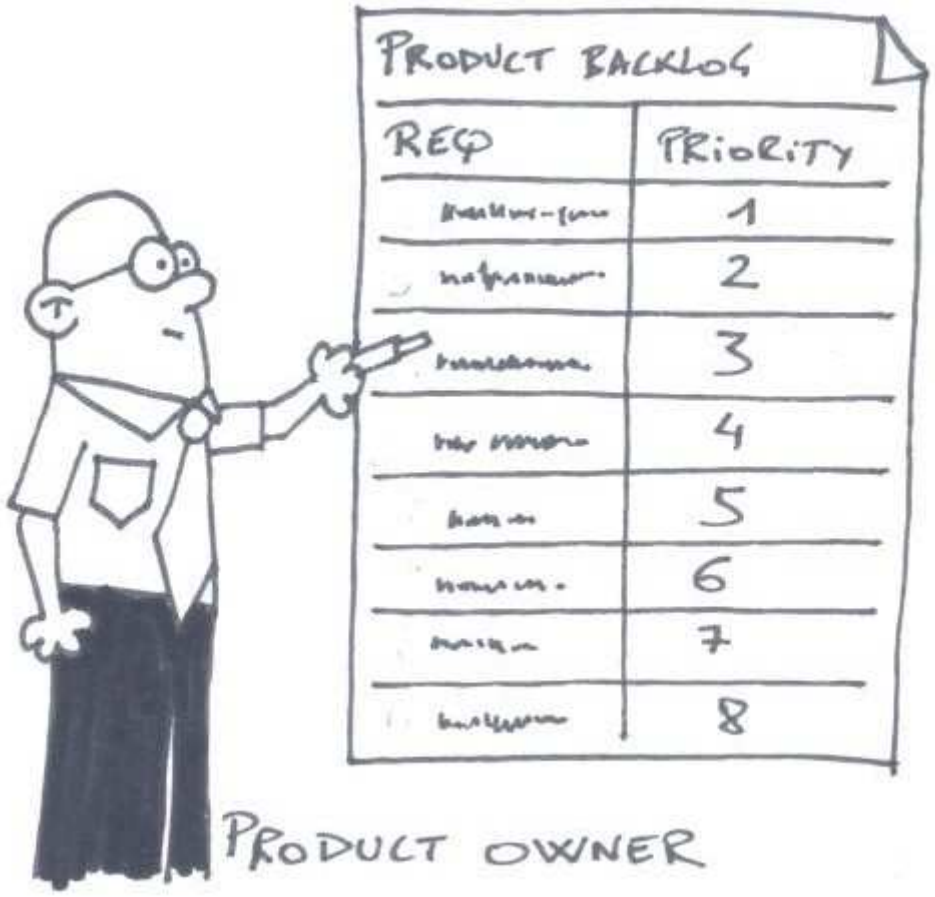
SCRUM: Team, ScrumMaster & Product Owner



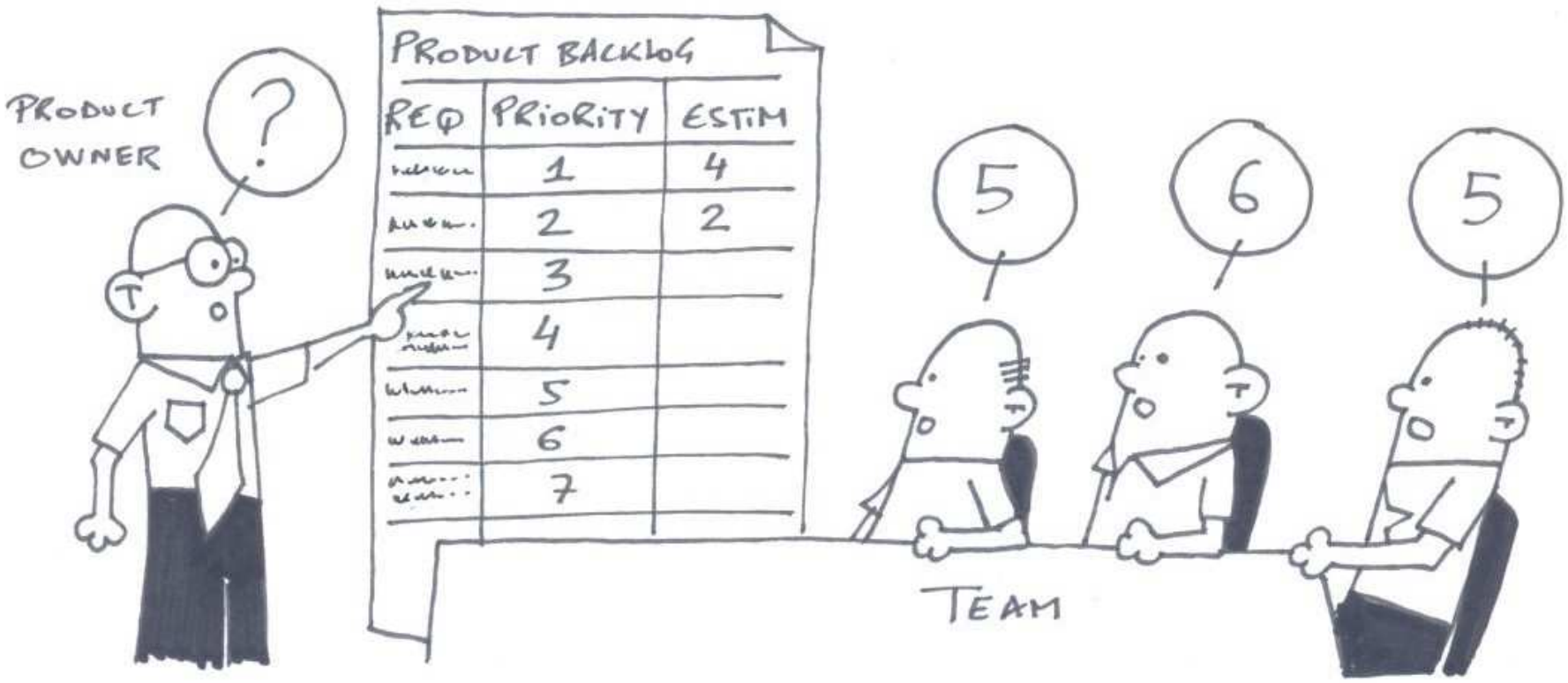
SCRUM: Pratiques



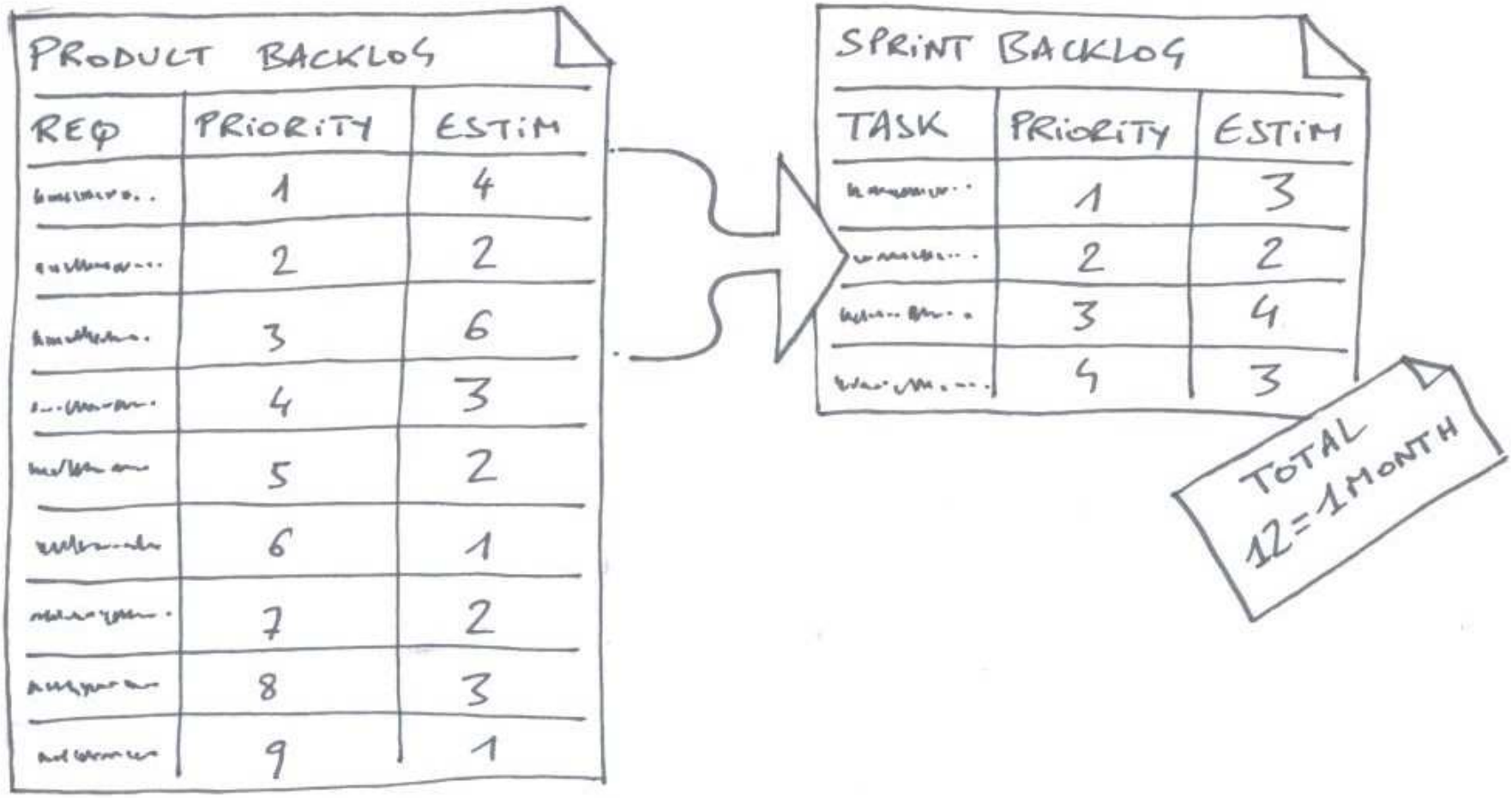
SCRUM: Product backlog



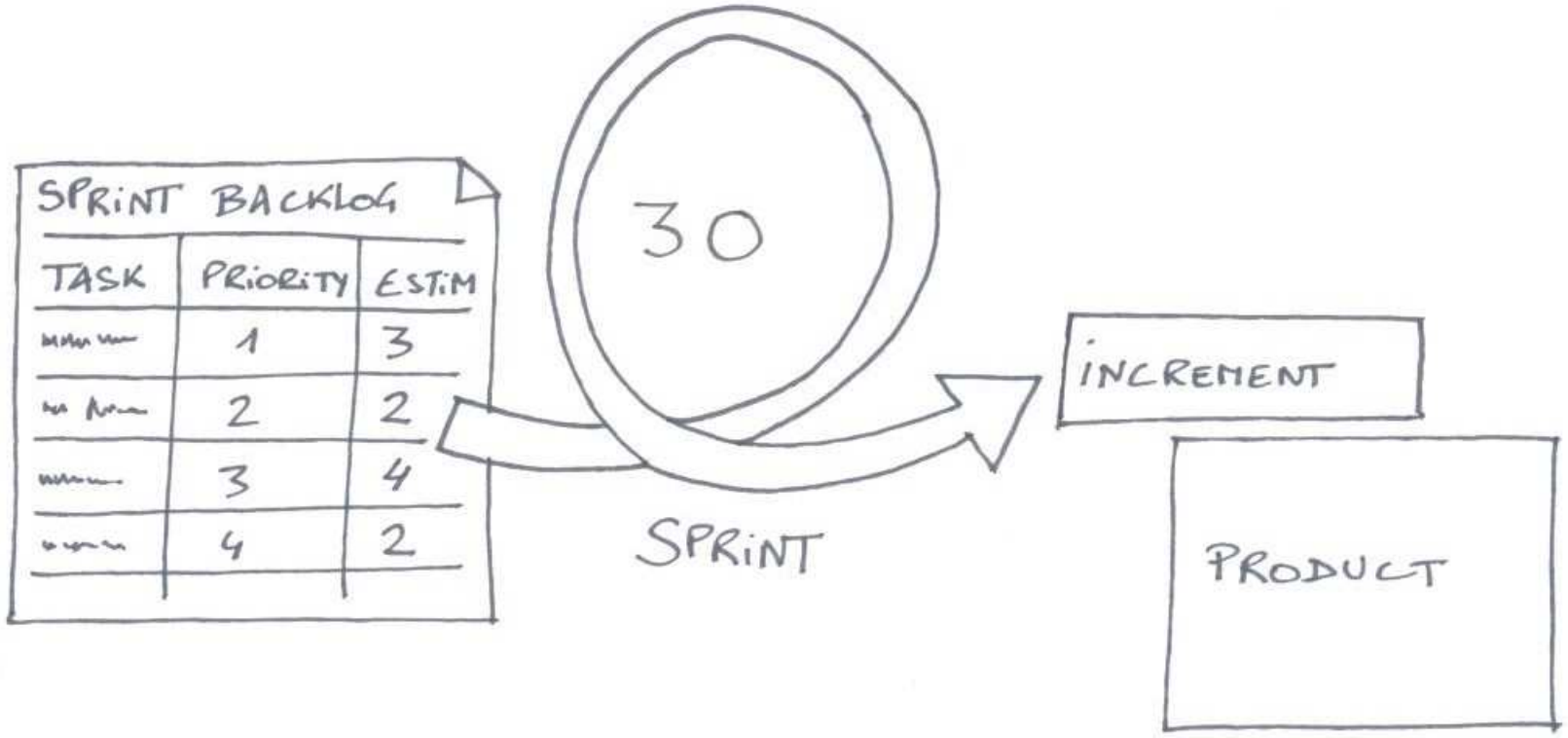
SCRUM: Sprint planning meeting



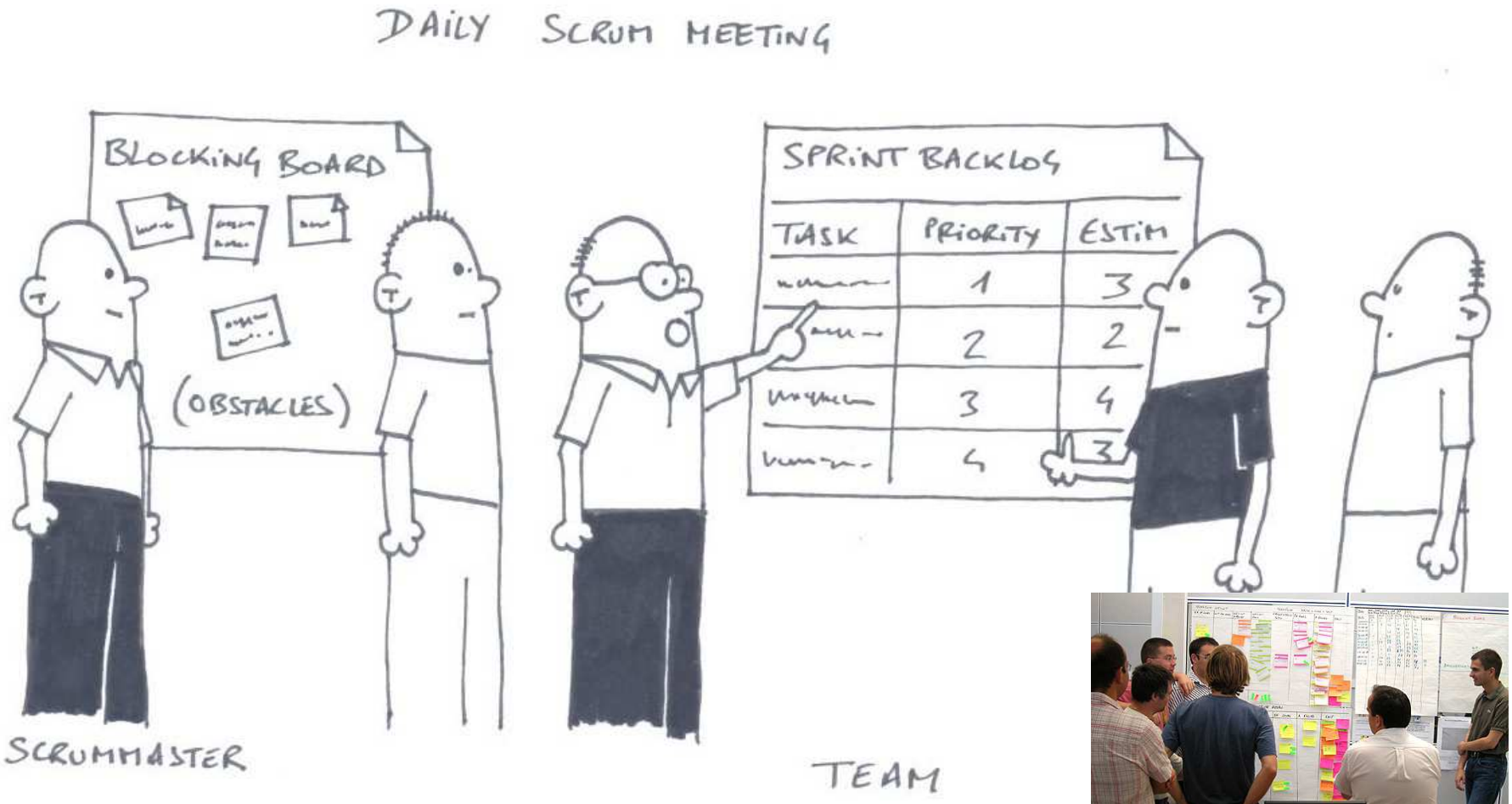
SCRUM: Sprint planning meeting & sprint backlog



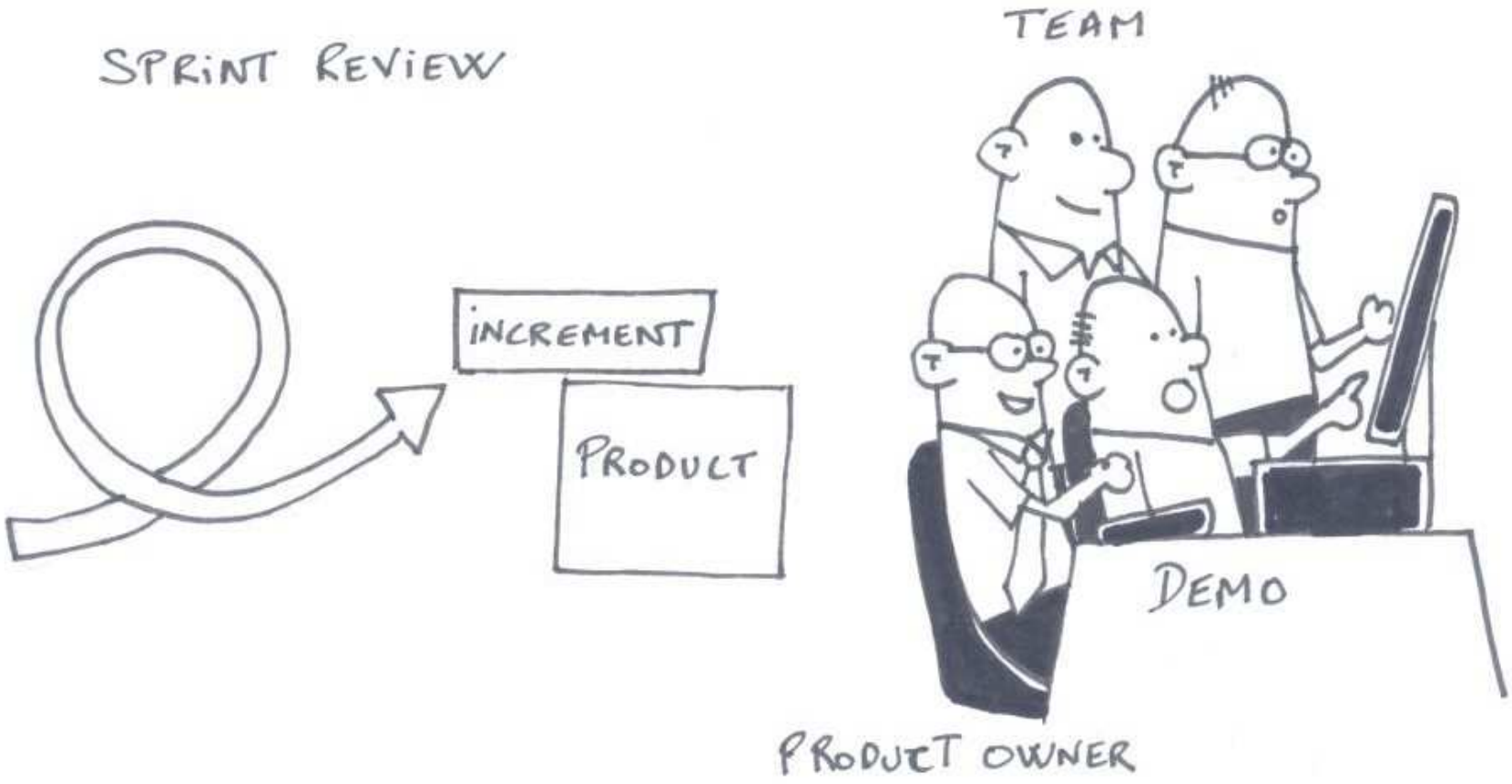
SCRUM: Sprint



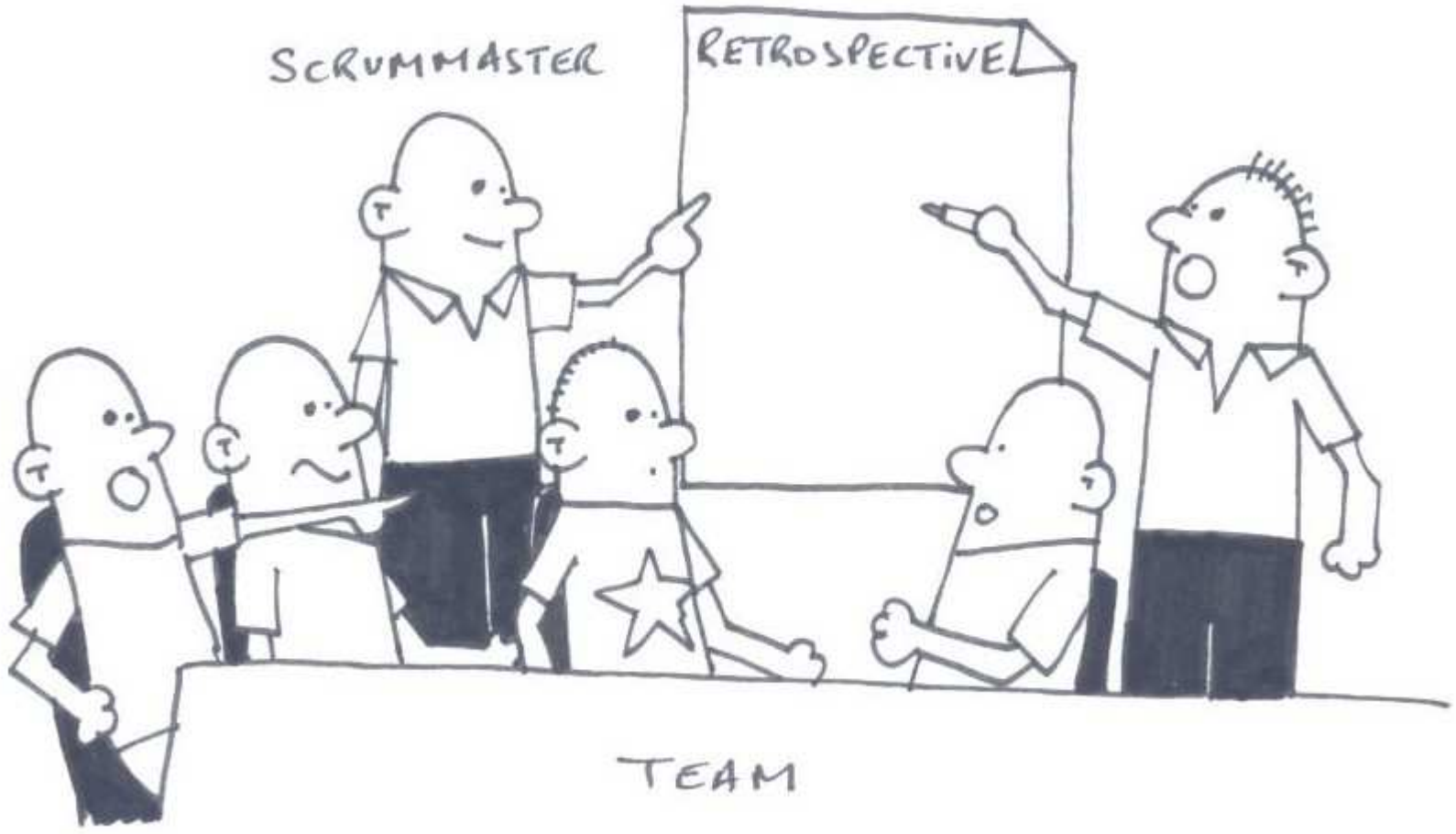
SCRUM: Daily scrum meeting



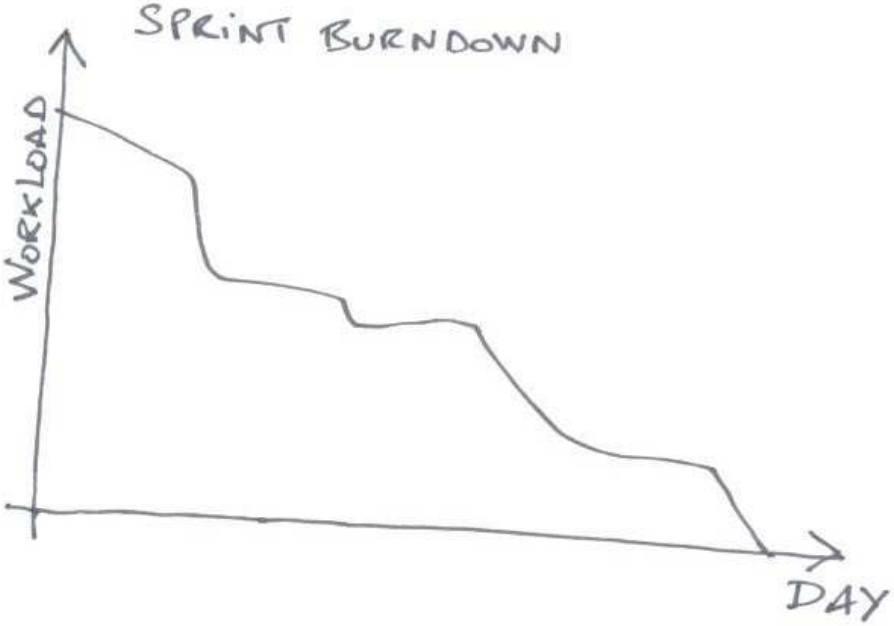
SCRUM: Sprint review meeting

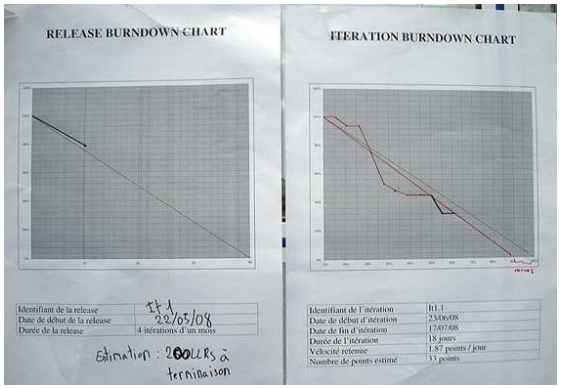


SCRUM: Sprint retrospective meeting



SCRUM: Burndown





eXtreme-Programming (XP)

- Méthode de développement logiciel
- Recherche l'efficacité maximale en concentrant l'effort de travail sur l'objectif de développer vite et juste.
- La démarche est légère, pragmatique, disciplinée, empirique et adaptative.

Les valeurs:

- Communication
- Simplicité
- Retour d'information
- Courage
- Respect

eXtreme-Programming (XP)

Pratiques:

- Jeu du Planning
- Intégration continue
- Petites livraisons
- Rythme soutenable
- Tests de recette
- Client sur site
- Tests unitaires
- Conception simple
- Remaniement
- Appropriation collective du code
- Convention de nommage
- Programmation en binôme

Scrum ou XP?

- **Scrum: démarche de gestion de projet (générique);**

- **XP : démarche technique de développement logiciel, qui prend en compte la gestion de projet;**

Scrum + XP !

Glossaire *1/7*

Application concurrente	Une application est concurrente lorsqu'elle a plusieurs flux d'exécution en parallèle. Les applications distribuées et multithread sont 2 exemples d'applications concurrentes.
Application multithread	Application dont le flux d'exécution se divise en flux parallèles appelés processus légers ou threads dans l'espace d'adressage d'un processus unique.
Architecture	L'**architecture logicielle** décrit d'une manière symbolique et schématique les différents composants d'un ou de plusieurs programmes informatiques, leurs interrelations et leurs interactions. L'architecture logicielle, produite lors de la phase de conception, ne décrit pas ce que doit réaliser un programme mais plutôt comment il doit être conçu de manière à répondre aux spécifications. L'analyse décrit le « quoi faire » alors que l'architecture décrit le « comment le faire ».
Assertion	Une **assertion** représente un énoncé considéré ou présenté comme vrai.
Asynchrone	L'asynchronisme qualifie deux processus qui ne se déroulent pas au même moment. Au sein d'un programme, les appels de méthodes sont faits de façon synchrone, c'est-à-dire l'appellent attend le retour de l'appelé pour continuer son exécution. En asynchrone, l'appelant continue immédiatement son exécution après un appel. L'appelé réalise son traitement de façon indépendante.
Boîte noire	Envisager un produit uniquement en termes d'entrées et sorties.

Glossaire *2/7*

Classe	En programmation orientée objet une **classe** déclare des propriétés communes à un ensemble d'objets. La classe déclare des attributs représentant l'état des objets et des méthodes représentant leur comportement. Une classe représente donc une catégorie d'objets. Il apparaît aussi comme un *moule* ou une *usine* à partir de laquelle il est possible de créer des objets. On parle alors d'un objet en tant qu'instance d'une classe (création d'un objet ayant les propriétés de la classe).
Conception	Voir Architecture.
Cohésion	Cohésion entre classes dans un package, entre services dans une classe : « qui se ressemble s'assemble »
Compatibilité	Facteur de qualité. La compatibilité est la facilité avec laquelle des éléments logiciels peuvent être combinés à d'autres.
Composant	Brique logicielle. Un composant est un élément d'un système, d'une application ayant des fonction prédéfinies et est capable, théoriquement , de communiquer avec d'autres composants. On parle aussi de programmation orientée composant, c'est-à-dire une programmation ayant une approche modulaire de l'architecture applicative.
Correction	Facteur de qualité. La correction est la capacité que possède un produit logiciel de mener à bien sa tâche, telle qu'elle a été définie par sa spécification.
Couplage	Une entité (fonction, module, classe, package, composant) est couplée à une autre si elle dépend d'elle.

Glossaire *3/7*

Couplage faible	Par opposition au couplage fort. Aussi nommé couplage lâche ou léger. Désigne une relation faible entre plusieurs entités (classes, composants), permettant une grande souplesse de programmation, de mise à jour. Ainsi, chaque entité peut être modifiée en limitant l'impact du changement au reste de l'application. Le couplage fort, au contraire, tisse un lien puissant qui rend l'application plus rigide à toute modification de code.
Couverture de code	Consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester.
Déploiement	Action permettant la diffusion d'une application. On peut utiliser un installeur ou simplement un téléchargement.
Design-pattern	Motif de conception ou patron de conception. Concept aidant à résoudre un problème récurrent. Il décrit une solution au problème que l'on rencontre. Il s'agit de procédés de conception, des bonnes pratiques.
Efficacité	Facteur de qualité. L'efficacité est la capacité d'un système logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l'espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication
Encapsulation	L'encapsulation permet de cacher quelque chose dans une autre chose. En programmation, l'encapsulation de données est l'idée de cacher l'information.
Extensibilité	Facteur de qualité. L'extensibilité est la facilité d'adaptation des produits logiciels aux changements de spécifications.

Glossaire *4/7*

Facilité d'utilisation	Facteur de qualité. La facilité d'utilisation est la facilité avec laquelle des personnes présentant des formations et des compétences différentes peuvent apprendre à utiliser les produits logiciels et s'en servir pour résoudre des problèmes. Elle recouvre également la facilité d'installation, d'opération et de contrôle
Fonctionnalité	La fonctionnalité est l'étendue des possibilités offertes par un système.
Génie logiciel	Le terme génie logiciel désigne l'ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel, au-delà de la seule activité de programmation
Gestion de configuration	La **gestion de configuration** consiste à gérer la description technique d'un système (et de ses divers composants), ainsi qu'à gérer l'ensemble des modifications apportées au cours de l'évolution du système. La gestion de configuration est utilisée pour la gestion de systèmes complexes . La **gestion de configuration** est avant tout un *ensemble de pratiques*. Ces pratiques sont au nombre de quatre. **- Identification des articles de la configuration** **- Contrôle des changements** **- Enregistrement des états de la configuration** **- Audit et revue**
Instance	Instance d'une classe = objet. Voir les définitions de classe et objet.

Glossaire *5/7*

Intégration	L'intégration a pour but de valider le fait que toutes les parties développées indépendamment fonctionnent bien ensemble. L'**intégration** est une activité d'un projet durant laquelle on vérifie le produit par des tests d'intégration
Intégration continue	L'intégration continue est une technique utilisée en génie logiciel. Elle consiste à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression de l'application en cours de développement. Bien que le concept existait auparavant, l"intégration continue" se réfère généralement a la pratique de l'extreme programming
Itération	Au sens général, une itération représente des passages dans une boucle de répétition. Au sens méthodologique, ce terme est employé pour désigner un passage au travers toutes les activités de développement d'un logiciel.
Objet	Un **objet** est l'instanciation d'une classe au sens de la programmation orientée objet. C'est une variable d'une classe.
Optimisation	En programmation optimiser le code consiste à le modifier pour rendre le logiciel plus efficace, c'est-à dire plus économe en terme de consommation de ressources (temps d'exécution, utilisation de la mémoire, …).

Glossaire *6/7*

Paradigme de programmation	Un **paradigme de programmation** est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation (à comparer à la méthodologie, qui est une manière de résoudre des problèmes spécifiques de génie logiciel.
Persistance	La persistance est un mot employé pour qualifier le fait de permettre à une application d'exploiter ses données même après un arrêt ou un crash: les données persistent même après l'arrêt de l'application.
Polymorphisme	En informatique, le **polymorphisme** est l'idée d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.
Programmation orientée objet	Paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées *objet*; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.
Ponctualité	Facteur de qualité. La ponctualité est la capacité d'un système logiciel à être livré au moment désiré par ses utilisateurs, ou avant.
Portabilité	Facteur de qualité. La portabilité est la facilité avec laquelle des produits logiciels peuvent être transférés d'un environnement logiciel ou matériel à l'autre

Glossaire *7/7*

Processus	Ensemble d'**activités** coordonnées et régulées, en partie ordonnées, dont le but est de créer un produit
Réutilisabilité	Facteur de qualité. La réutilisabilité est la capacité des éléments logiciels à servir à la construction de plusieurs applications différentes
Robustesse	Facteur de qualité. La robustesse est la capacité qu'offrent des systèmes logiciels à réagir de manière appropriés à la présence de conditions anormales
Spécification	C'est l'étape en génie logiciel qui consiste à **décrire** ce que le logiciel doit faire
Test	En informatique, un **test** (anglicisme) désigne une procédure de vérification partielle d'un système informatique : le but est de s'assurer que le système informatique réagit de la façon prévue par ses concepteurs.
Test d'intégration	Test qui vérifie que toutes les parties développées indépendamment fonctionnent bien ensemble.
Test unitaire	Le **test unitaire** est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme. Il s'agit pour le programmeur de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances.

Mythes – affirmations erronées

Les délais et le budget ne sont pas compatibles d'un travail propre.
Les exigences peuvent être assez précises dès le début du projet.
Les exigences sont stables.
La conception peut être terminée avant que ne débute la programmation.
Il faut tester pour trouver les défauts.
Se dépêcher créé du gaspillage.
Il existe une unique meilleure manière de procéder.
Planifier, c'est s'engager.

Les règles d'or

Travailler en **équipe**, confronter les points de vue
Être **pragmatique**
Utiliser **l'expérience** acquise
Rendre les choses **simples**
Faire les choses **bien – et y prendre du plaisir!**