# Génie Logiciel Avancé
# Cours 1 : Introduction

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

29 janvier 2010

## Qu'est-ce que le génie logiciel ?

- Le **génie logiciel** est un domaine des sciences de l'ingénieur dont l'objet d'étude est la conception, la fabrication et la maintenance des systèmes informatiques complexes.

## Qu'est-ce qu'un système ?

- Un système est un ensemble d'éléments intéragissant entre eux suivant un certains nombres de principes et de règles dans le but de réaliser un objectif.
- La frontière d'un système est le critère d'appartenance au système.
- L'environnement est la partie du monde extérieure au système.
- Un système est souvent hiérarchisé à l'aide de sous-systèmes.
- Un système complexe se caractérise par :
  - sa dimension qui nécessite la collaboration de plusieurs personnes ;
  - son évolutivité.
- Exemples : une fourmilière, l'économie mondiale, le noyau linux, ...

## Qu'est-ce qu'un logiciel ?

- Un logiciel est un ensemble d'entités nécessaires au fonctionnement d'un processus de traitement automatique de l'information.
- Parmi ces entités, on trouve par exemple :
  - des programmes exécutables ;
  - des documentations d'utilisation ;
  - des informations de configuration.

## Qu'est-ce qu'un logiciel ?

- Un logiciel est en général un sous-système d'un système englobant.
- Il peut interagir avec des clients, qui peuvent être :
  - des opérateurs humains (des utilisateurs, des administrateurs, ...) ;
  - d'autres logiciels ;
  - des contrôleurs matériels.
- Il réalise une spécification : son comportement vérifie un ensemble de critères qui régissent ses interactions avec son environnement.

⇒ Le génie logiciel vise à garantir que :

1. la spécification répond aux besoins réels de ses clients ;
2. le logiciel respecte sa spécification ;
3. les coûts alloués pour sa réalisation sont respectés ;
4. les délais de réalisation sont respectés.

## Comment spécifier un logiciel ?

Que doit faire le logiciel ?

- La spécification d'un logiciel peut prendre de nombreuses formes.
- La complexité et les dimensions de la spécification peuvent varier énormément en fonction de l'environnement d'utilisation du logiciel et des objectifs auxquels il répond.

## Quelques exemples de spécifications

$$\forall I, \mathit{Precondition}(I) \implies \exists O, \mathit{Postcondition}(I, O)$$

- Un algorithme de tri :
  - *Entrée* : un tableau $t$.
  - *Précondition* : il existe une relation d'ordre sur les éléments du tableau.
  - *Sortie* : un tableau $u$.
  - *Postcondition* : $u$ est trié et contient exactement les mêmes éléments que $t$.
- La partie arrière d'un compilateur :
  - *Entrée* : un arbre de syntaxe abstraite $P$.
  - *Précondition* : le programme est bien typé.
  - *Sortie* : un fichier exécutable $E$.
  - *Postcondition* : la sémantique de $E$ est la même que celle de $P$.

⇒ Ce sont des spécifications simples dont la conformité aux objectifs de leurs clients ne fait aucun doute. (Cela ne rend pas aisée pour autant leur réalisation.)

## Quelques exemples de spécifications plus complexes

- Une interface graphique : Le modèle d'interaction avec le client est non déterministe. Doit-on spécifier toutes les traces d'exécution possibles ?
- Un traducteur automatique : Qu'est-ce qu'un texte anglais « bien écrit » ?
- Un logiciel « boursicoteur » (effectuant des achats et des ventes en bourse) : Comment établir une spécification sans y inclure un modèle du système financier ?
- Un jeu vidéo : Comment spécifier ce qui est amusant ?

## Comment concevoir un logiciel de qualité ?

- En plus du respect (essentiel) de sa spécification, la qualité d'un logiciel dépend des 4 critères suivants :
  1. Maintenabilité : Peut-on faire évoluer le logiciel ? [1]
  2. Robustesse : Le logiciel est-il sujet à des dysfonctionnements ?
  3. Efficacité : Le logiciel fait-il bon usage de ses ressources ?
  4. Utilisabilité : Est-il facile à utiliser ?

[1] Un logiciel ne s'use pas. La correction d'une erreur n'est pas évolution mais un échec du concepteur.

## Comment fabriquer un logiciel de qualité ?

- Historiquement, il y a eu une prise de conscience dans les années 70, appelée la crise du logiciel, dû à un tournant décisif : c'est à cette époque que le coût de construction du logiciel est devenu plus important que celui de la construction du matériel.

> *The major cause of the software crisis is that the machines have become several orders of magnitude more powerful ! To put it quite bluntly : as long as there were no machines, programming was no problem at all ; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Edsger Dijkstra, The Humble Programmer (EWD340)

`http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html`

## Comment fabriquer un logiciel de qualité ?

- Pour répondre à cette crise, on a essayé d'appliquer les méthodes connues de l'ingénieur au domaine du logiciel, pour établir des méthodes fiables sur lesquelles construire une industrie du logiciel.
- Il s'agit de se donner un cadre rigoureux pour :
  - Guider le développement du logiciel, de sa conception à sa livraison.
  - Contrôler les coûts, évaluer les risques et respecter les délais.
  - Établir des critères d'évaluation de la qualité d'un logiciel.

## Les spécificités du logiciel

- Cependant, la construction d'un logiciel diffère de celle d'un pont car :
  - une modification infime peut avoir des conséquences critiques ;
  - les progrès technologiques très rapides peuvent rendre un logiciel caduque ;
  - il est difficile de raisonner sur des programmes ;
  - les domaines des entrées des logiciels sont trop grands pour le test exhaustif ;
  - les défaillances des programmes sont en général dues à des erreurs humaines ;
  - on ne sait pas très bien réutiliser les programmes existants ;
  - chaque logiciel a son organisation et sa logique propre ;
  - ...

## Comment fabriquer un logiciel de qualité ?

- Le génie logiciel est un domaine en pleine évolution qui offre une grande palette d'outils et de méthodes pour parvenir à construire du logiciel de qualité.
- Aucune de ses méthodes ne s'est imposée à ce jour : il faut donc prendre du recul sur les concepts et les conseils qu'elles préconisent et utiliser son **bon sens** pour les adapter à chaque situation.
- Ces méthodes se distinguent principalement par :
  - leur degré de formalisme ;
  - leur champ d'application ;
  - les contraintes de qualité qu'elles ambitionnent.

## Comment fabriquer un logiciel de qualité ?

- Les approches formelles utilisent des outils mathématiques et des méthodes de preuve pour construire un **logiciel correct par construction** dont la vérification est automatisée ou assistée.
- Méthodes : méthode B, *Model-checking*, Logique de Hoare ...
- Outils et notations : Coq, Z, VHDL, Atelier B, CASL, Why, Frama-C, ...
- Ces méthodes sont utilisées pour développer des logiciels critiques.
- Elles correspondent au niveau le plus élevé de certification.

## Comment fabriquer un logiciel de qualité ?

- Les approches semi-formelles visent à introduire un langage normalisé pour décrire le logiciel et sa spécification.
- Cependant, la **sémantique du langage de spécification n'est pas formalisée**.
- Bien que ces approches précisent le discours du concepteur si on le compare à celui décrit à l'aide du langage naturel, elles contiennent certaines ambiguïtés et n'offrent aucune garantie sur la qualité des résultats.
- Méthodes : Rationale Unified Process, Merise, ...
- Outils et notations : UML, AnalyseSI, ...
- Ces méthodes sont utilisées aujourd'hui par l'industrie du logiciel.

## Comment fabriquer un logiciel de qualité ?

- Les approches empiriques mettent en avant un ensemble de "bonnes pratiques" qui ont fait leur preuve par l'expérience.
- Méthodes : relecture de code, *extreme programming*, programmation défensive, ...
- Outils : gestionnaire de versions, outil de documentation automatique, ...

## Les grands principes du génie logiciel

- Un certain nombre de grands principes (de bon sens) se retrouvent dans toutes ces méthodes. En voici une liste proposée par C. Ghezzi :
  1. La rigueur.
  2. La décomposition des problèmes en sous-problèmes indépendants.
  3. La modularité.
  4. L'abstraction.
  5. L'anticipation des évolutions.
  6. La généricité.
  7. La construction incrémentale.

## La rigueur

- Les principales sources de défaillances d'un logiciel sont d'origine humaine.
- À tout moment, il faut se questionner sur la validité de son action.
- Des outils de vérification accompagnant le développement peuvent aider à réduire les erreurs. Cette famille d'outils s'appelle CASE (*Computer Aided Software Engineering*).
- Exemples : typeurs, assistants de preuves, générateurs de code, générateurs de tests...

## La décomposition des problèmes en sous-problèmes

- « *Separation of concerns* » en anglais.
- Il s'agit de :
  - Décorréler les problèmes pour n'en traiter qu'un seul à la fois.
  - Simplifier les problèmes (temporairement) pour aborder leur complexité progressivement.

## La décomposition des problèmes en sous-problèmes

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only ; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so : why, the program is desirable. But nothing is gained –on the contrary !– by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect" : it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Edsger Dijkstra, *On the role of scientific thought*

## La décomposition des problèmes en sous-problèmes

- Exemple 1 :
  *Comment acheminer un email de façon sûr à travers un réseau ?*

⇒ Décomposition en couches utilisée sur Internet :

- STMP : protocole de la couche application qui suppose une couche de transport de paquet sûr.
- TCP : protocole de la couche transport permettant de s'assurer que tous les paquets arrivent, même si le réseau peut perdre des paquets.

- Exemple 2 :
  *Comment créer dynamiquement une page internet pour visualiser et modifier le contenu d'une base donnée sans la corrompre ?*

⇒ Décomposition en trois composants :

- Modèle : son rôle est gérer le stockage des données.
- Vue : son rôle est formatter les données.
- Contrôleur : son rôle est de n'autoriser que les modifications correctes.

## La modularité

- C'est une instance cruciale du principe de décomposition des problèmes.
- Il s'agit de partitionner le logiciel en modules qui :
  - ont une cohérence interne (des invariants) ;
  - possèdent une interface ne divulgant sur le contenu du module que ce qui est strictement nécessaire aux modules clients.
- L'évolution de l'interface est indépendante de celle de l'implémentation du module.
- Les choix d'implémentation sont indépendants de l'utilisation du module.
- Ce mécanisme s'appelle le **camouflage de l'information** (*information hiding*).

## L'abstraction

- C'est encore une instance du principe de décomposition des problèmes.
- Il s'agit d'exhiber des concepts généraux regroupant un certain nombre de cas particuliers et de raisonner sur ces concepts généraux plutôt que sur chacun des cas particuliers.
- Le fait de fixer la bonne granularité de détails permet :
  - de raisonner plus efficacement ;
  - de factoriser le travail en instanciant le raisonnement général sur chaque cas particulier.
- Exemples en programmation : les classes abstraites dans les langages à objets, le polymorphisme de Caml, les fonctions d'ordre supérieur.

## L'anticipation des évolutions

- Un logiciel a un cycle de vie plus complexe que l'habituel cycle « commande-spécification-production-livraison ».
- La maintenance est la gestion des évolutions du logiciel.
- Il est primordial de prévoir les évolutions possibles d'un logiciel pour que la maintenance soit la plus efficace possible. Pour cela, il faut s'assurer que les modifications à effectuer soient le plus locales possibles.
- Ces modifications ne devraient pas être **intrusives** car les modifications du produit existant remettent en cause ses précédentes validations.
- Concevoir un système suffisamment riche pour que l'on puisse le modifier **incrémentalement** est l'idéal.

## La généricité

- Un logiciel réutilisable a beaucoup plus de valeur qu'un composant dédié.
- Un composant est générique lorsqu'il est adaptable.

## La construction incrémentale

- Un développement logiciel a plus de chances d'aboutir si il suit une cheminement incrémental (*baby-steps*).
- Exemple :
  Laquelle de ses deux méthodes de programmation est la plus efficace ?
  1. Écrire l'ensemble du code source d'un programme et compiler.
  2. Écrire le code source d'une fonction, le compiler et passer à la suivante.

## À propos des principes

- Vous devez avoir en tête ces principes : ils se retrouvent dans toutes les méthodes et outils que nous allons aborder.

## Les processus de développement logiciel

## Qu'est-ce qu'un processus ?

- Un processus de développement logiciel est un ensemble (structuré) d'activités que conduisent à la production d'un logiciel.
- Il n'existe pas de processus idéal.
- La plupart des entreprises adapte les processus existants à leurs besoins.
- Ces besoins varient en fonction du domaine, des contraintes de qualité, des personnes impliquées.

⇒ Ce qui est essentiel, c'est de comprendre quel est son rôle dans ce processus et d'en saisir les rouages.

⇒ L'étude et la pratique de processus existants doit vous permettre de vous forger un regard affuté (et même critique) sur ces processus.

## Activités du développement logiciel

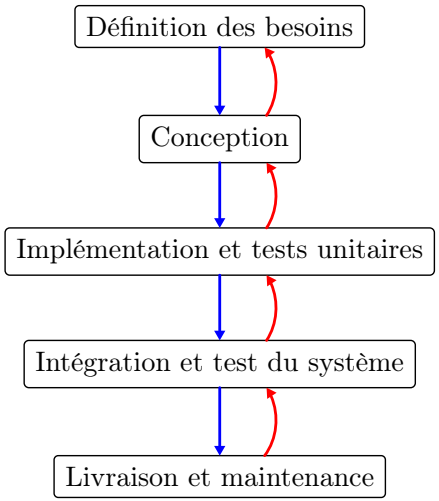
- Les activités des processus de développement logiciels se regroupent en 4 grandes catégories :

1. La spécification du logiciel définit ses fonctionnalités et leurs contraintes.
2. La conception et l'implémentation sont chargées de réaliser le logiciel, en conformité avec sa spécification.
3. La validation s'assure effectivement du respect de la spécification par le logiciel produit.
4. L'évolution adapte le logiciel aux besoins futurs de ses clients.

## Schéma général d'un processus de développement

- Il est très rare d'appliquer un processus comme une unique séquence des 4 activités précédentes.
- En effet, ce serait à l'encontre du principe d'incrémentalité.
- En général, un logiciel complet est le fruit de plusieurs itérations.
- Chaque itération contient les 4 activités de spécification, conception, validation et évolution.
- Il existe différents modèles de processus qui organisent de façon différentes ces activités : le modèle en cascade, le modèle de développement évolutif et le modèle de développement par composants.

## Modèle en cascade




- Chaque phase doit se terminer pour commencer la suivante.
- Des documents sont produits pour concrétiser la réalisation de chaque phase.

## Le modèle en cascade

- Le modèle en cascade est hérité des méthodes classiques d'ingénierie.

⇒ Il s'adapte donc bien dans un contexte où le logiciel fait partie d'un système complexe englobant.

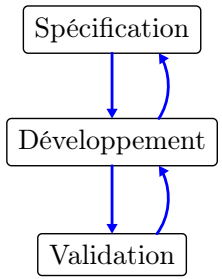
- La production de documents entre chaque phase améliore le suivi du projet.
- Lorsqu'une erreur a été commise dans une phase et qu'elle est détectée dans une phase suivante, il faut faire remonter cette information dans la phase incriminée et recommencer le processus à partir de celle-ci. On doit alors reproduire de nouveaux documents ...
- Ce modèle de processus impose donc une importante réflexion sur les choix faits en amont car le coût de la correction d'une erreur est important.

⇒ Typique d'un développement industriel pour lequel les coûts de la construction du produit sont trop importants pour se permettre une erreur de choix de conception.

## Critique du modèle en cascade

- Le modèle en cascade rend coûteux le développement itératif puisque la rédaction des documents de validation de chaque phase demande beaucoup de travail.
- Ce modèle est inadapté au développement de systèmes dont la spécification est difficile à formuler *a priori*.

## Modèle de développement évolutif




- Ces trois activités sont entrelacées.
- Un prototype est écrit rapidement et est confronté à l'utilisateur.
- En fonction du résultat, on raffine la spécification.
- On reprend le prototype ou on le réécrit jusqu'à l'obtention du système final.

## Le modèle de développement évolutif

- Ce modèle augmente les chances de répondre aux besoins de l'utilisateur car il permet de les comprendre plus rapidement. (*Are we building the right product ?*)
- Il remplit le critère d'incrémentalité.
- *Ce modèle ne dispense d'écrire la spécification* du système car il faut s'assurer que l'implémentation est correct. (*Are we building the product right ?*)
- C'est un processus particulièrement adapté aux projets de taille moyenne ou importante (inférieur à 100 000 lignes de code) comme par exemple les applications WEB ou encore les solutions intégrées pour les petites entreprises.

## Critique du modèle de développement évolutif

- Il est plus difficile de gérer un projet utilisant ce modèle car la visibilité de l'avancement du développement est peu clair.

⇒ Dans ce cadre, encore plus que dans un autre, un chef de projet doit aussi être un bon programmeur puisqu'il doit être capable de se faire une idée de l'état du système en observant le développement (possiblement chaotique) des prototypes.

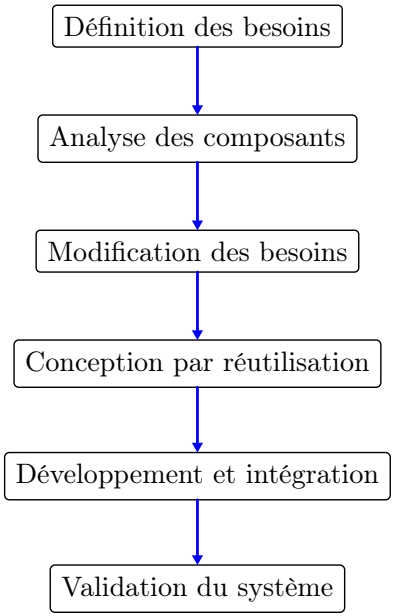
- Il est difficile de structurer correctement le logiciel (définir de bonnes abstractions, modulariser efficacement) car les prototypes sont par définition des produits "bricolés".
- Le coût en termes de tests et de validation du produit final peuvent être très importants.

⇒ Des approches mixtes intégrant modèle de développement évolutif pour produire un premier prototype validé et un modèle en cascade pour reconstruire correctement un produit final constituent en général de bons compromis.

## Modèle de développement à livraison incrémentale

- Une approche à mi-chemin entre le modèle en cascade et le modèle de développement évolutif s'appuie sur une livraison incrémentale du produit.
- On hiérarchise les besoins du client en termes de priorité.
- Chaque itération du modèle vise à obtenir un ensemble de fonctionnalités par ordre de priorité.
- Traiter les parties les plus critiques du système en premier permet de minimiser les risques d'inadéquation avec le produit final.
- Cependant, il se peut que les choix pris en amont, trop focalisés sur ce noyau de fonctionnalités, compromettent le développement des fonctionnalités secondaires.

## Modèle de développement par composants




## Modèle de développement par composants

- Ce modèle vise à développer un logiciel en grande partie à l'aide d'une base de composants génériques pré-existants.
- L'élaboration de la spécification est dirigée par cette base : une fonctionnalité est proposée à l'utilisateur en fonction de sa facilité à l'obtenir à l'aide d'un composant existant.

⇒ Situation typique chez les sociétés de services (hébergement de serveurs, déploiement automatique de site web, etc . . . ).

- Ce modèle permet d'obtenir rapidement des produits de bonne qualité puisqu'ils sont construits à partir de composants qui ont fait leur preuve.
- Le travail d'intégration peut s'appuyer sur des outils dirigés par des descriptions de haut-niveau du système qui génèrent le code de "glue" par exemple.

## Critique du modèle de développement par composants

- Le principal défaut de ce modèle est de ne pas construire un produit adapté aux besoins du client.

⇒ Un travail complexe de configuration et d'adaptation peut être nécessaire.

La gestion de projet

## Quel est le rôle d'un chef de projet ?

- Les activités de gestion d'un projet informatique sont très similaires à celles des autres domaines :
  - Écriture de proposition de projet
  - Plannification du projet
  - Évaluation des coûts
  - Surveillance du projet et écriture de rapport d'étapes
  - Sélection du personnel et évaluation
  - Écriture de rapport et de présentation

## Écrire une proposition de projet

- À partir d'un appel d'offre, un chef de projet doit écrire une proposition de projet décrivant les objectifs du projet (en général, ses délivrables) et les grandes lignes de sa réalisation.
- Une proposition doit aussi contenir une évaluation des risques et des coûts.
- La plupart du temps, cette proposition doit servir d'argumentaire pour justifier la mise en route du projet.

⇒ C'est une activité qui requiert une importante expérience et compréhension du domaine d'activité. Le chef de projet engage sa responsabilité.

## Plannifier un projet

- Le chef de projet doit établir un jalonnement, c'est-à-dire une répartition des activités dans le temps en fonction de leurs dépendances et des ressources disponibles et d'une évaluation des risques liés à leur réalisation.

⇒ Il s'agit d'un travail d'ordonnancement qui nécessite encore une connaissance très précise du domaine, des équipes de développement, etc ...

## Veiller sur un projet

- De façon continue, le chef de projet s'assure du progrès des tâches et du respect des délais.
- En cas de retard, il doit réévaluer la plannification et éventuellement renégocier les ressources et les contraintes du projet.

⇒ La visibilité de la progression des activités est ici essentielle. Un chef de projet doit donc savoir se doter d'indicateurs révélateurs sur l'état du développement.

## Sélectionner le personnel

- En cohérence avec la politique de gestion du personnel (projet de carrière, formation continue, sous-traitement, ...), le chef de projet doit affecter des activités et des rôles aux différentes personnes impliquées dans le projet.

⇒ Des qualités relationnelles semblent donc utiles...

## Écrire un rapport

- Le chef de projet doit pouvoir communiquer une vue synthétique du projet à différents publics (autres chefs de projet, clients, responsables, etc ...).

Synthèse de cette introduction

## Objectifs du cours

- Ce cours a pour but de vous familiariser avec les futures structures de votre vie professionnelle et de vous donner les outils de vous adapter à la situation, nécessairement singulière, dans laquelle vous serez acteurs.
- Il a aussi pour objectif de développer vos capacités d'analyse de problèmes de conception logiciel.

## Organisation du cours

- Le cours se déroule le vendredi en amphi 6C de 8h30 à 10h30 et sera présenté par Mihaela Sighireanu et Yann Régis-Gianas.
- Les travaux dirigés se déroulent le vendredi de 10h30 à 12h30 en salle 473F et sont encadrés par Constantin Enea et Mihaela Sighireanu.
- La page du cours : `http://www.pps.jussieu.fr/~yrg/gl/index.php`
- Inscrivez-vous sur la *mailing-list* ! (voir le site).

## Validation

- Le cours est validé par un projet et par un examen.
- Le projet consiste à développer un logiciel en équipe, en utilisant les méthodes et outils de génie logiciel que nous découvrirons.

## Bibliographie


- *Software Engineering* – 8$^{\text{ème}}$ édition
  Sommerville – Addison Wesley
- *Génie logiciel*
  J. Longchamp – Cours de CNAM


## QCM : Question 1

Le génie logiciel fournit des outils et des méthodes pour :

- ☐ analyser les besoins d'un client.
- ☐ créer des besoins chez un potentiel client.
- ☐ s'assurer que les contraintes budgétaires d'un projet sont respectées.
- ☐ réaliser correctement une spécification.
- ☐ construire des composants logiciels réutilisables.

## QCM : Question 2

Le produit appelé « logiciel » peut être composé :

- ☐ de programmes exécutables.
- ☐ de tests.
- ☐ de manuels d'utilisation.
- ☐ de scripts de configuration automatique.

## QCM : Question 3

La spécification d'un logiciel peut :

- ☐ être définie après son implémentation.
- ☐ être issue de l'étape de validation.
- ☐ ne pas exister.
- ☐ être inappropriée.
- ☐ être incohérente.

## QCM : Question 4

La robustesse d'un programme est :

- ☐ caractérisée par sa résistance aux chocs.
- ☐ proportionnelle à sa stabilité.
- ☐ une conséquence de sa correction vis-à-vis de sa spécification.
- ☐ une condition nécessaire à sa correction vis-à-vis de sa spécification.

## QCM : Question 5

La « crise du logiciel » était causée par :

- ☐ une crise de l'investissement dans le domaine informatique ;
- ☐ un inversement du rapport entre les coûts du logiciel et du matériel ;
- ☐ un déficit en informaticiens sur le marché du travail.

## QCM : Question 6

Une méthode de développement formelle :

- ☐ prouve mathématiquement la correction d'un logiciel vis-à-vis de sa spécification.
- ☐ n'est pas très coûteuse.
- ☐ rend inutile les phases de tests.
- ☐ est toujours applicable.
- ☐ peut s'appuyer sur le langage UML.

## QCM : Question 7

Quels sont les « bons principes » de développement dans la liste suivante :

- ☐ la modularité ;
- ☐ le code « spaghetti » ;
- ☐ la réinvention de la roue ;
- ☐ le code est la spécification ;
- ☐ la décomposition des problèmes.

(C.f. les « *anti-patterns* »)

## QCM : Question 8

Cacher les détails d'implémentation :

- ☐ est une erreur de conception puisqu'il faut que le client d'un module ait un maximum d'information sur ce module pour l'utiliser au mieux.
- ☐ permet de faire rendre indépendant l'implémentation d'un module de ses utilisations.
- ☐ introduit une forme d'abstraction.
- ☐ est impossible lorsque l'on programme vraiment.

## QCM : Question 9

Un processus de développement :

- ☐ fixe un cadre rigoureux pour le développement de projets de taille importante.
- ☐ est une perte de temps !
- ☐ doit s'appliquer à la lettre.
- ☐ peut être itéré.
- ☐ peut s'appuyer sur plusieurs modèles de processus.

## QCM : Question 10

C'est le rôle d'un chef de projet :

- ☐ de programmer les composants d'un logiciel.
- ☐ de vérifier le bon déroulement des tâches.
- ☐ d'organiser l'enchaînement des tâches.
- ☐ de fournir une visibilité globale sur un projet.
- ☐ d'écrire la spécification du logiciel.