Génie logiciel pour (l'entreprise) la modélisation : I: Programmation extrême et défensive

Jean-Baptiste Apoung Kamga

Cours M2 IM
Université Paris-sud XI

Plan

But

Permettre une meilleure intégration et un meilleur comportement dans une équipe de développement où les projets traités sont de grande envergure.
Il est question de connaître les exigences et les comportements attendues dans ces circonstances.

1. Quelles sont les exigences humaines comment s'y préparer?
   1. Pourquoi ces exigences ? (vous travaillez pour un client) et donc question
   2. Quelles est ma position? relation par rapport à l'autre?

   (Réponse à travers une cartographie d'une équipe de développement selon les standards approuvés.)
2. Quelles sont les compétences attendues et en suis-je préparé
   (A travers des TP)
   1. Conception (Méthodologie, Pratiques conseillées)
   2. Applications(Outils à disposition: optim,debug,générateurs, versionage)
3. Mise en oeuvre
   1. Partir d'un code (C++)
   2. Applications des techniques
      (documentations, structuration, tests, validation)

Outline

Définition de Génie logiciel

Ce n'est pas seulement

la programmation

C'est

un ensemble des méthodes, des techniques et outils concourant à la production d'un logiciel

Son but visé

respecter les délais, maîtriser le budget et donner satisfaction au client.

Vocabulaire rattaché

gestion de projet.

Outline

Cycle de vie d'un logiciel

C'est l' ensemble des étapes

du développement d'un logiciel, de sa conception à sa disparition.

Objectif du découpage :

Définition des jalons intermédiaires nécessaires à la validation du développement.
(conformité aux besoins exprimés et adéquation des méthodes mises en oeuvres)

Origine du découpage

- constat d'un coût élevé des erreurs détectées tardivement.
- constat que des erreurs détectées au plus tôt induit
  la maîtrise de la qualité du logiciel, des délais et les coûts associés.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Activités de cycle de vie du logiciel

Le cycle de vie du logiciel comprend généralement a minima les activités suivantes :

- **Définition des objectifs**, consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité**, c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale**. Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée**, consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation), soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.
- **Tests unitaires**, permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration**, dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** (ou recette), c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation**, visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production**,
- **Maintenance**, comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

La séquence et la présence de chacune de ces activités dans le cycle de vie dépend du choix d'un modèle de cycle de vie entre le client et l'équipe de développement.

Modèles de cycles de vie

Origine

permettre une méthodologie commune entre

- le client
- la société de service réalisant le développement logiciel

Fonctionnalité au quotidien

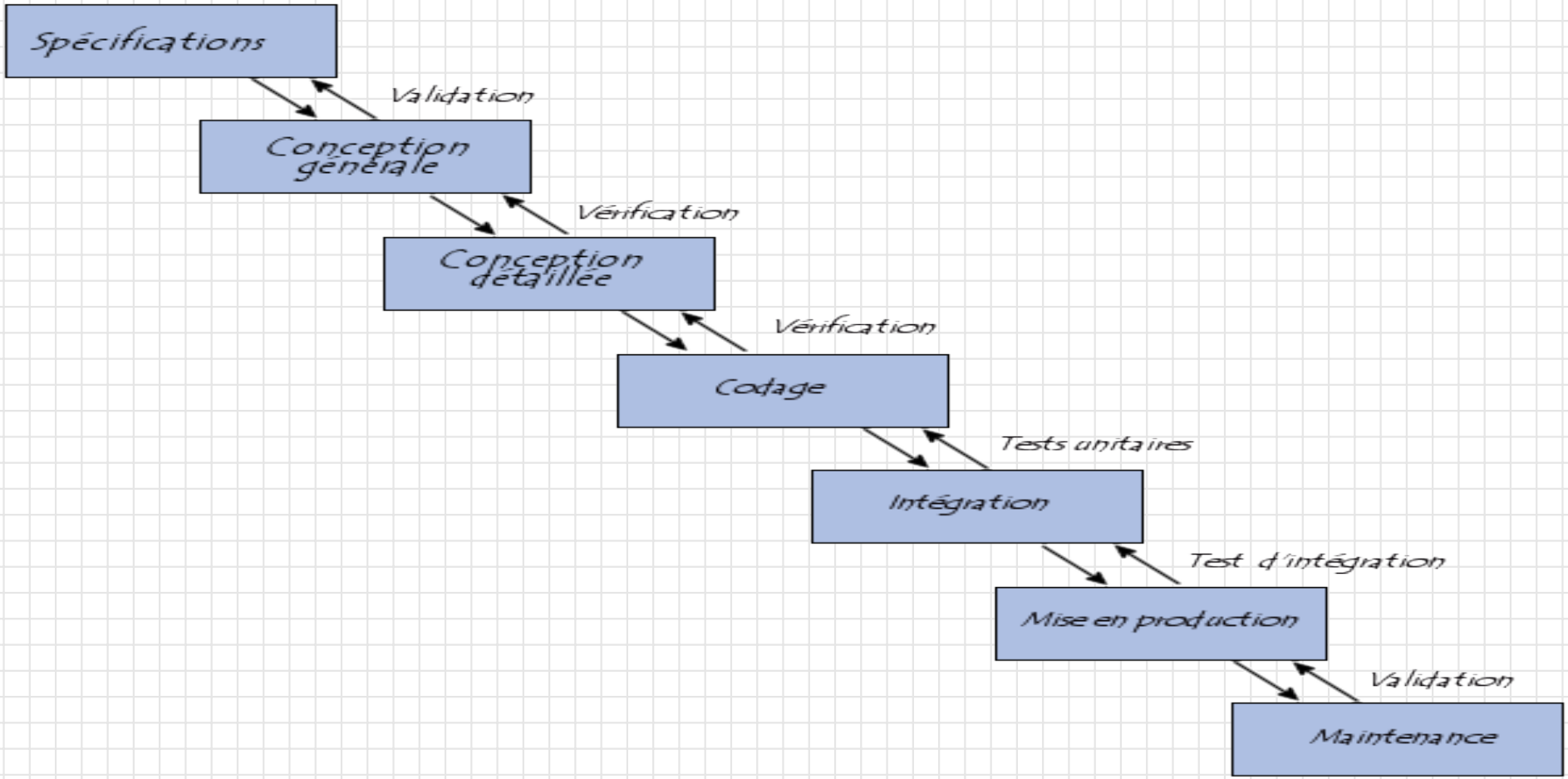
définition des

- étapes du développement
- documents à produire permettant la validation de chacune des étapes avant le passage à la suivante.

A la fin de chaque phase, des revues sont organisées

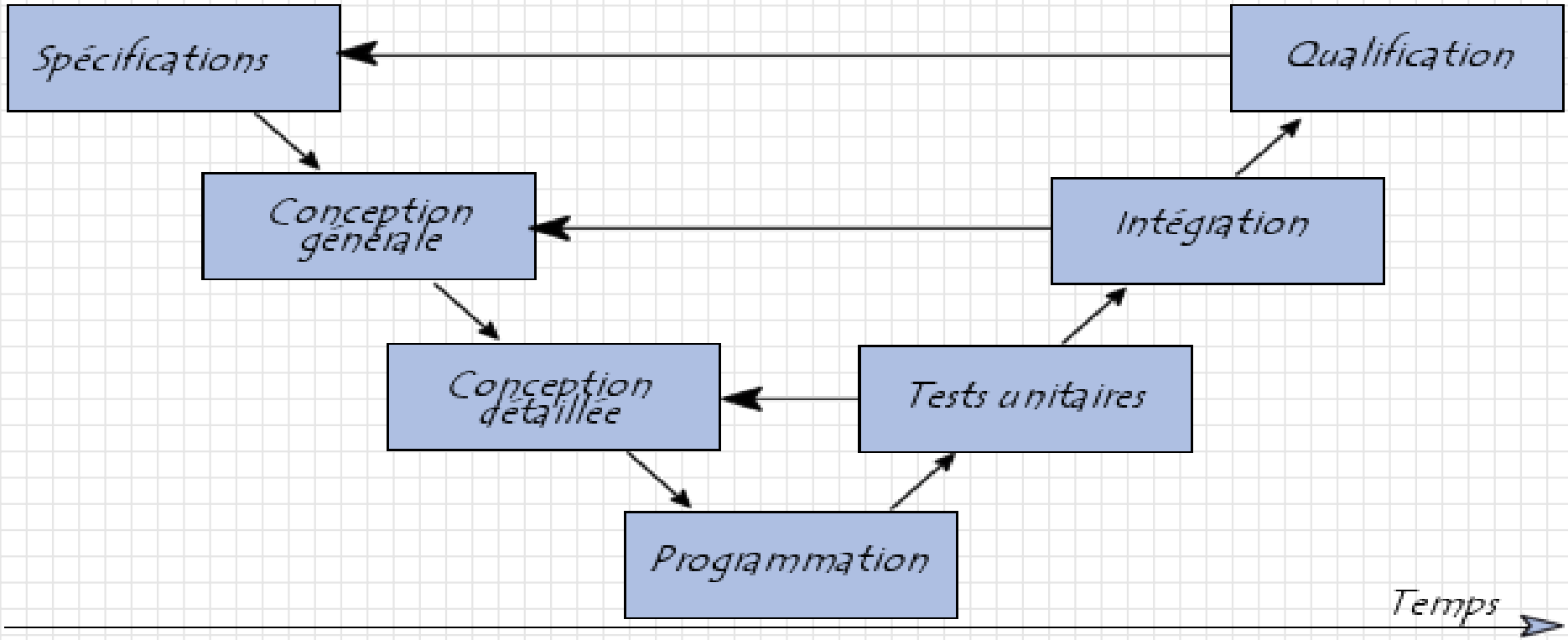
Modèle en cascade

Origine: mis au point dès 1966, puis formalisé aux alentours de 1970.
Caractéristiques: définition des phases séquentielles



Modèle en V

Ce modèle part du principe que les procédures de vérification de la conformité du logiciel aux spécifications doivent être élaborées dès les phases de conception.



Outline

Méthodes agiles

But

Les méthodes de développement dites « méthodes agiles » (en anglais Agile Modeling, noté AG) visent à réduire le cycle de vie du logiciel (donc accélérer son développement) en développant une version minimale, puis en intégrant les fonctionnalités par un processus itératif basé sur une écoute client et des tests tout au long du cycle de développement.

Origine

- Instabilité de l'environnement technologique
- le client est souvent dans l'incapacité de définir ses besoins de manière exhaustive dès le début du projet.

Le terme « agile » fait ainsi référence à la capacité d'adaptation aux changements de contexte et aux modifications de spécifications intervenant pendant le processus de développement.

En 2001, 17 personnes mirent ainsi au point le manifeste agile dont la traduction est la suivante :

- individus et interactions plutôt que processus et outils
- développement logiciel plutôt que documentation exhaustive
- collaboration avec le client plutôt que négociation contractuelle
- ouverture au changement plutôt que suivi d?un plan rigide

Grâce aux méthodes agiles, le client est pilote à part entière de son projet et obtient très vite une première mise en production de son logiciel. Ainsi, il est possible d'associer les utilisateurs dès le début du projet

RAD - Développement rapide d'applications

Origine

La « **méthode de développement rapide d'applications** » en anglais *Rapid Application Development, notée RAD*,
est définie par James Martin au début des années 80

Caractéristiques

Elle consiste en un cycle de développement court basé sur 3 phases

- Cadrage
- Design
- Construction

dans un délai idéal de 90 jours et de 120 jours au maximum.

DSDM

Origine

La méthode DSDM (*Dynamic Software Development Method*) a été mise au point en s'appuyant sur la méthode RAD afin de combler certaines de ses lacunes, notamment en offrant un canevas prenant en compte l'ensemble du cycle de développement.

Caractéristiques

Les principes fondateurs de la méthode DSDM sont les suivants :

- Une implication des utilisateurs
- Un développement itératif et incrémental
- Une fréquence de livraison élevée
- L'intégration des tests au sein de chaque étape
- L'acceptation des produits livrés dépend directement de la satisfaction des besoins

UP - Unified Process

Origine

La méthode du Processus Unifié (*UP pour Unified Process*) est un processus de développement itératif et incrémental, ce qui signifie que le projet est découpé en phases très courtes à l'issue de chacune desquelles une nouvelle version incrémentée est livrée.

Caractéristiques

Il s'agit d'une démarche s'appuyant sur la modélisation UML pour la description de l'architecture du logiciel (fonctionnelle, logicielle et physique) et

RUP - Rational Unified Process

Origine

RUP (*Rational Unified Process*) est une méthode de développement par itérations promue par la société Rational Software, rachetée par IBM.

Caractéristiques

RUP propose une méthode spécifiant notamment la composition des équipes et le calendrier ainsi qu'un certain nombre de modèles de documents.

XP - eXtreme Programming

Origine

La méthode XP (pour eXtreme Programming) définit un certain nombre de bonnes pratiques permettant de développer un logiciel dans des conditions optimales en plaçant le client au coeur du processus de développement, en relation étroite avec le client.

Caractéristiques

L'eXtreme Programming est notamment basé sur les concepts suivants :

- Les équipes de développement travaille directement avec le client sur des cycles très courts d'une à deux semaines maximum.
- Les livraisons de versions du logiciel interviennent très tôt et à une fréquence élevée pour maximiser l'impact des retours utilisateurs.
- L'équipe de développement travaille en collaboration totale sur la base de binômes.
- Le code est testé et nettoyé tout au long du processus de développement.
- Des indicateurs permettent de mesure l'avancement du projet afin de permettre de mettre à jour le plan de développement.

Outline

Design Patterns

Ce que c'est

Les Design Patterns (en français Patrons de conception, Modèles de conception ou encore Motifs de conception sont un recueil de bonnes pratiques de conception pour un certain nombre de problèmes récurrents en programmation orientée objet.

Origine

Le concept de Design Pattern est issu des travaux de 4 personnes (**Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides** connus sous le patronyme de « **Gang of Four** ») dans leur ouvrage «**Design Patterns: Elements of Reusable Object-Oriented Software** » édité en **1995** et proposant 23 motifs de conception.
Un motif de conception peut être vu comme un document formalisant la structure d'une classe permettant de répondre à une situation particulière.

Caractérisqtiques

Les motifs de conception sont classifiés selon trois grandes familles :

- **Motifs de création** : Motif Abstract Factory, Motif Builder, Motif Factory Method, Motif Prototype, Motif Singleton.
- **Motifs de structuration** : Motif Adapter, Motif Bridge, Motif Composite, Motif Decorator, Motif Facade, Motif Flyweight, Motif Proxy.
- **Motifs de comportement** : Motif Chain of Responsibility, Motif Command, Motif Interpreter, Motif Iterator, Motif Mediator, Motif Memento, Motif Observer, Motif State, Motif Strategy, Motif Template Method, Motif Visitor.

Motifs de conception

Voici quelques exemples de motifs de conception :

- Motif **MVC** (Modèle-Vue-Contrôleur) : il part du principe que toute application peut être décomposée en trois couches séparées :
  - **Modèle**, c'est-à-dire les données
  - **Vue**, c'est-à-dire la représentation des données
  - **Controlleur**, c'est-à-dire le traitement sur les données en vue de leur représentation.
- Motif **Proxy** définissant un objet intermédiaire ayant procuration pour effectuer de manière transparente pour l'utilisateur les appels de méthodes à un objet distant.

Outline

Atelier de génie logiciel

Un **atelier de génie logiciel** (noté *AGL* ou en anglais Case, pour *Computer Aided Software Environment*) est un ensemble d'outils logiciels structurés au sein d'une même interface permettant la conception, le développement et le débogage de logiciels.
Un AGL comprend ainsi des outils permettant de modéliser visuellement une application, à produire du code avec des assistants visuels et éventuellement un débogueur permettant de tester le code produit.

Outline

Outline

Principes

1. Codez et concevez avec simplicité
2. Pratiquez inlassablement la restructuration
3. Développez les standards de développement
4. Développez un vocabulaire commun

Codez et concevez avec simplicité

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Pratiquez inlassablement la restructuration

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Développez les standards de développement

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Développez un vocabulaire commun

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Pratiques développeur

Principes

1. Adoptez un développement piloté par les tests
2. Programmer en binôme
3. Adoptez la propriété collective du code
4. Intégrez continuellement

Adoptez un développement piloté par les tests

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Programmer en binôme

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Adoptez la propriété collective du code

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Intégrez continuellement

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Principes

1. Ajoutez un client à l'équipe
2. Jouez le jeu de la planification
3. Livrez régulièrement
4. Travaillez à un rythme raisonnable

Ajoutez un client à l'équipe

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Jouez le jeu de la planification

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Livrez régulièrement

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Travaillez à un rythme raisonnable

Pourquoi ?

1.
2.

Comment?

1.
2.

Conclusion

1.
2.

Développement d'un "Test Suite"

La pratique de développement XP reposant sur les tests, nous nous proposons ici la réalisation d'un outils simple de Tests Unitaires.

La classe Test

Entête du fichier Test.hpp

Entête du Fichier Test.hpp

```
1  #ifndef _TEST_H_
2  #define _TEST_H_
3  #include <string>
4  #include <iostream>
5  #include <cassert>
6  using std::string;
7  using std::ostream;
8  using std::cout;
9
10 #define test_(cond) \
11   do_test(cond, #cond, __FILE__, __LINE__)
12 #define fail_(str) \
13   do_fail(str, __FILE__, __LINE__)
14
15 namespace TestSuite{
```

Ces deux macros (expression C++) permettent de localiser les lignes et fichier où l'erreur s'esr produite.

La classe Test : la classe

La classe Test

```
1  class Test
2  {
3  public:
4  Test(ostream* osptr = &cout) {
5      this->osptr = osptr;
6      nPass = nFail = 0;
7    }
8    virtual ~Test() {}
9    virtual void run() = 0; // fonction virtuelle
10   long getNumPassed() const { return nPass; }
11   long getNumFailed() const { return nFail; }
12   const ostream* getStream() const { return osptr; }
13   void setStream(ostream* osptr) { this->osptr = osptr; }
14   void succeed_() { ++nPass; }
15   long report() const;
16   virtual void reset() { nPass = nFail = 0; }
17
18 protected:
19 void do_test(bool cond, const string& lbl,
20      const char* fname, long lineno);
21   void do_fail(const string& lbl,
22      const char* fname, long lineno);
23 private:
24 ostream* osptr;
25 long nPass;
26 long nFail;
27 // Disallowed:
28 Test(const Test&);
29 Test& operator=(const Test&);
30 };
```

La classe Suite

Rôle

- regroupe un ensemble de tests unitaires
- contrôle leur exécution
- rapporte les statistiques de leur exécution

Entête du fichier Suite.hpp

```
1  #ifndef SUITE_H
2  #define SUITE_H
3  #include <vector>
4  #include <stdexcept>
5  #include "Test.h"
6  using std::vector;
7  using std::logic_error;
8
9  namespace TestSuite {
10
11 class TestSuiteError : public logic_error {
12 public:
13   TestSuiteError(const string& s = "")
14   : logic_error(s) {}
15 };
```

La classe Suite : la classe

La classe Suite

```
1  class Suite {
2    string name;
3    ostream* osptr;
4    vector<Test*> tests;
5    void reset();
6    // Disallowed ops:
7    Suite(const Suite&);
8    Suite& operator=(const Suite&);
9  public:
10   Suite(const string& name, ostream* osptr = &cout)
11   : name(name) { this->osptr = osptr; }
12   string getName() const { return name; }
13   long getNumPassed() const;
14   long getNumFailed() const;
15   const ostream* getStream() const { return osptr; }
16   void setStream(ostream* osptr) { this->osptr = osptr; }
17   void addTest(Test* t) throw(TestSuiteError);
18   void addSuite(const Suite&);
19   void run();  // Calls Test::run() repeatedly
20   long report() const;
21   void free();  // Deletes tests
22 };
23
24 } // namespace TestSuite
25 #endif // SUITE_H ///:~
```

Exemple d'utilisation Simple

Un exemple de test des méthodes d'une classe

```
1  #ifndef DATETEST_H
2  #define DATETEST_H
3  #include "Date.h"      // class a tester
4  #include "Test.h"      // test simple
5  class DateTest : public TestSuite::Test {
6    Date mybday;
7    Date today;
8    Date myevebday;
9  public:
10   DateTest(): mybday(1951, 10, 1), ↩
           myevebday("19510930") {}
11   void run() {
12     testOps();
13     testFunctions();
14     testDuration();
15   }
16   void testOps() {
17     test_(mybday < today);
18     test_(mybday <= today);
19     test_(mybday != today);
20     test_(mybday == mybday);
21     test_(mybday >= mybday);
22     test_(mybday <= mybday);
23     test_(myevebday < mybday);
24     test_(mybday > myevebday);
```

```
1      test_(mybday >= myevebday);
2      test_(mybday != myevebday);
3    }
4    void testFunctions() {
5      test_(mybday.getYear() == 1951);
6      test_(mybday.getMonth() == 10);
7      test_(mybday.getDay() == 1);
8      test_(myevebday.getYear() == 1951);
9      test_(myevebday.getMonth() == 9);
10     test_(myevebday.getDay() == 30);
11     test_(mybday.toString() == "19511001")↩
           ;
12     test_(myevebday.toString() == "↩
           19510930");
13   }
14   void testDuration() {
15     Date d2(2003, 7, 4);
16     Date::Duration dur = duration(mybday, ↩
           d2);
17     test_(dur.years == 51);
18     test_(dur.months == 9);
19     test_(dur.days == 3);
20   }
21 };
22 #endif // DATETEST_H ///:~
```

Exemple d'appel

Appel simple

```
1  #include <iostream>
2  #include "Date.h"
3  #include "DateTest.h"
4
5  using namespace std;
6  using namespace TestSuite;
7
8  int main (int argc, char **argv)
9  {
10  DateTest test;
11
12  test.run();
13
14  return test.report();
15
16 }
```

Appel évolué

```
1  #include <iostream>
2  #include "Date.h"
3  #include "DateTest.h"
4  #include "Suite.h"
5
6  using namespace std;
7  using namespace TestSuite;
8
9  int main(int argc, char **argv) {
10     Suite suite("Quelques Tests");
11     // insersion du test de Date
12     suite.addTest(new DateTest);
13     suite.run();
14     long nFail = suite.report();
15     suite.free();
16     return nFail;
17 }
```

Associer un test unitaire à un projet existant

On dispose d'un projet. Dans notre cas tout se trouve dans le même répertoire
On souhaite accoler à ce projet un outil de test unitaire, afin de s'assurer du succès des tests de non non régression

La démarche à suivre est la suivante qui sera détaillée en TP. On suppose que le projet utilise les *autotools*.

1. Creer le répertoire qui contiendra les tests mkdir testcases
2. Se déplacer dans ce répertoire, y créer et modifier le fichier Makefile.am
3. Se déplacer dans le répertoire parent de *testcases*:
   1. modifier le fichier configure.ac
   2. modifier le fichier Makefile.am
4. Associer ensuite vos fichiers tests
5. Compiler
   1. Regénérer le fichier configure (autoreconf -i)
   2. Regénérer le Makefile (./configure)
   3. Générer les exécutable (make; make check)
6. Executer les tests: aller dans le répertoire *testcases* et exécuter tests.