

Génie Logiciel : Fondamentaux et Pratiques Modernes

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

Plan du cours	2	9.2 Diagramme d'objets	10
1 Introduction au génie logiciel	2	9.3 Diagramme de composants	10
1.1 Le génie logiciel aujourd'hui	2	9.4 Diagramme de déploiement	10
2 2. Principes fondamentaux	3	9.5 Diagramme de paquetages	11
2.1 Définition et objectifs du génie logiciel	3	10 Diagrammes comportementaux	11
3 3. Cycle de vie et processus	4	10.1 Diagramme de cas d'utilisation ...	11
3.1 3.1 Cycle de vie du logiciel	4	10.2 Diagramme de séquence	12
3.2 Modèles classiques	5	10.3 Diagramme d'activité	12
4 Waterfall	5	10.4 Diagramme d'état	12
5 V-Model	5	10.5 Diagramme de communication ...	13
5.1 Approches agiles	6	11 Bonnes pratiques UML	13
5.2 Scrum	6	12 Outils de modélisation UML	13
5.3 Kanban	6	13 6. Implémentation et tests	13
5.4 Kanban : Principes fondamentaux . .	6	13.1 Implémentation	13
5.5 XP (Extreme Programming)	7	13.2 Bonnes pratiques de programmation	13
5.6 Scrumban	7	14 Tests et Qualité	14
5.7 Processus unifié	8	14.1 Assurance qualité approfondie ...	14
5.8 RAD (Rapid Application Development)	8	15 7. Déploiement et maintenance	14
6 4. Gestion de projet	8	15.1 Gestion de versions	14
6.1 Approches de gestion de projet	8	15.2 Versionnement sémantique (SemVer)	14
7 5. Analyse et conception	9	15.3 Systèmes de gestion de versions (VCS)	14
7.1 5.1 Ingénierie des exigences	9	15.4 Workflows de développement ...	14
7.2 Techniques de collecte des besoins ..	9	16 Intégration continue (CI) et livraison continue (CD)	15
7.3 5.2 Conception logicielle et UML	9	17 Déploiement continu (CD)	15
8 Modélisation avec UML	9	18 8. Pratiques modernes	15
9 Diagrammes structurels	9	18.1 8.1 Gestion de versions	15
9.1 Diagramme de classes	9		

18.2	8.2 Intégration et déploiement continu	15
19	Gestion de projet	15
19.1	Approches de gestion de projet	15
20	Analyse et spécification des besoins	15
20.1	Techniques de collecte des besoins	15
21	Conception logicielle	16
21.1	Approches fondamentales	16
21.2	Niveaux de conception	16
21.3	Méthodologies de conception logicielle	16
21.4	Domain-Driven Design (DDD)	17
21.5	Test-Driven Development (TDD)	17
21.6	Behavior-Driven Development (BDD)	17
21.7	Model-Driven Architecture (MDA)	17
21.8	Feature-Driven Development (FDD)	18
21.9	Continuous Design	18
21.10	Event Storming	18
22	Planification et estimation	18
23	Maintenance et évolution	18

Ce cours explore les fondements du génie logiciel, combinant théorie et pratique pour une compréhension complète du développement logiciel moderne.

Objectifs pédagogiques: - Maîtriser les concepts fondamentaux - Comprendre les différentes approches - Acquérir des compétences pratiques - Développer un esprit critique

Plan du cours

1. Introduction

- Définition et importance
- État de l'art et défis actuels

2. Fondamentaux

- Principes d'ingénierie logicielle
- Qualités logicielles

- Cycle de vie

3. Processus et Méthodes

- Approches classiques
- Méthodes agiles
- Gestion de projet

4. Ingénierie des Exigences

- Analyse des besoins
- Spécification
- Validation

5. Conception et Architecture

- Patterns et principes
- Modélisation UML
- Architecture logicielle

6. Développement et Tests

- Pratiques de codage
- Types de tests
- Assurance qualité

7. Livraison et Maintenance

- Intégration continue
- Déploiement
- Gestion des versions

8. Pratiques DevOps

- CI/CD
- Monitoring
- Infrastructure as Code

1 Introduction au génie logiciel

1.1 Le génie logiciel aujourd'hui

Le génie logiciel est une discipline en constante évolution qui répond aux défis modernes du développement logiciel:

1. Complexité des systèmes - Architectures distribuées - Intégration de multiples technologies - Systèmes hautement concurrents - Sécurité renforcée

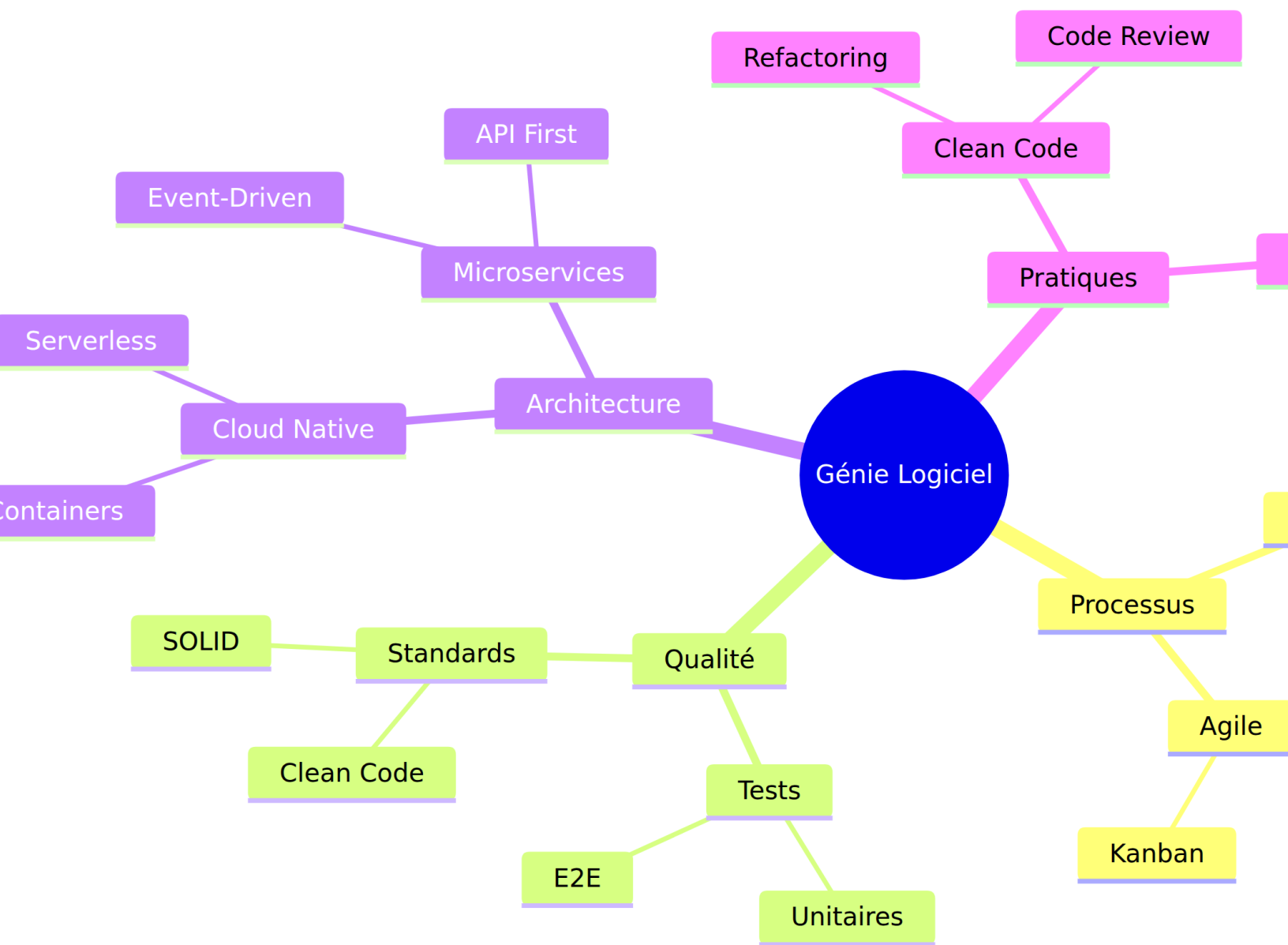


Figure 1. – Mindmap des pratiques modernes du génie logiciel

2 2. Principes fondamentaux

2.1 Définition et objectifs du génie logiciel

Le génie logiciel est une discipline qui intègre les principes d'ingénierie et de gestion pour créer des logiciels fiables et efficaces.

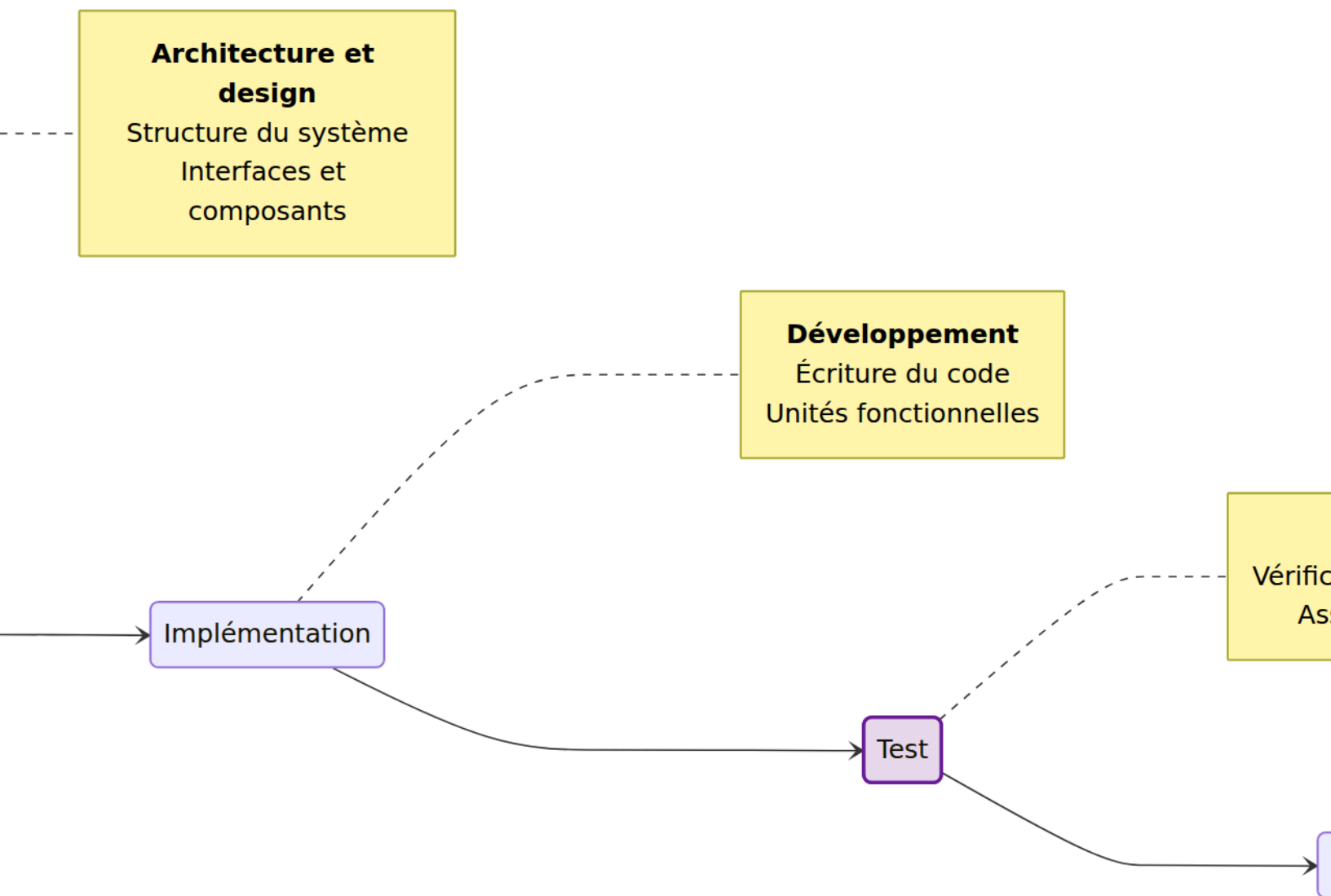


Figure 3. – Cycle de vie du logiciel

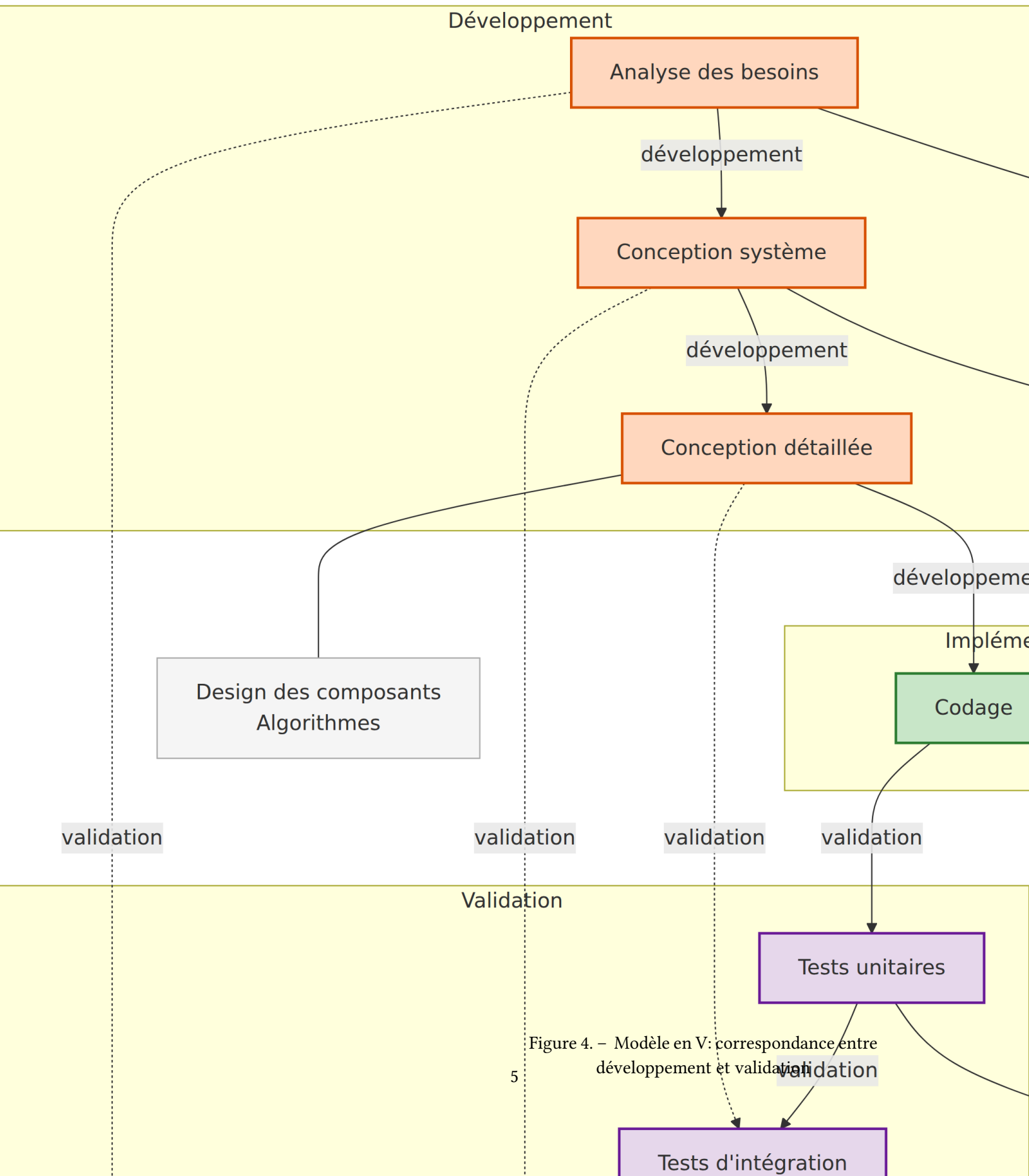
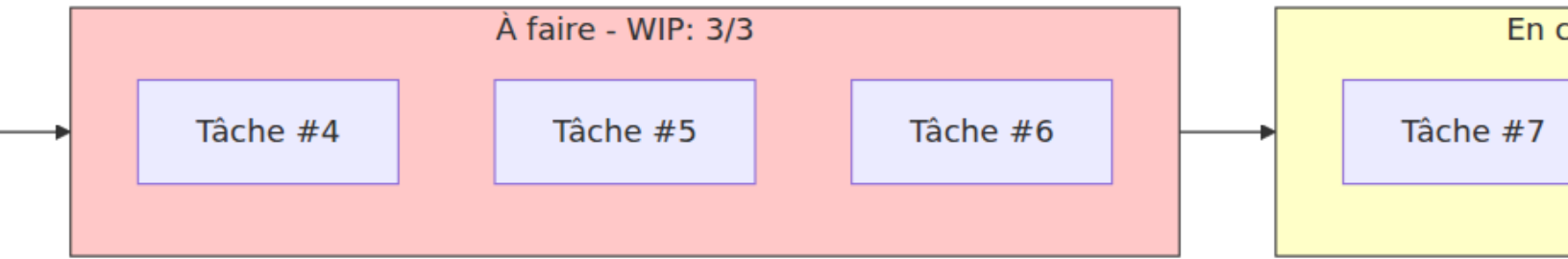


Figure 4. – Modèle en V: correspondance entre développement et validation

Avantages: Emphase sur les tests, traçabilité

5.3 Kanban



-
- Product Owner (responsable du produit)
- Scrum Master (facilitateur)
- Équipe de développement (auto-organisée)
- **Rituels:**
 - Daily Stand-up (synchronisation quotidienne)
 - Sprint Planning (planification)
 - Sprint Review (démonstration)
 - Sprint Retrospective (amélioration)
- **Forces:** Cadre structuré, livraison rapide, collaboration

Figure 5. – Tableau Kanban avec limites WIP

5.4 Kanban : Principes fondamentaux

- **Visualisation du flux de travail :** Représentation claire de chaque étape du processus
- **Limitation du travail en cours (WIP) :** Prévention de la surcharge et identification des goulots d'étranglement
- **Gestion du flux :** Optimisation pour un mouvement fluide et prévisible des tâches
- **Politiques explicites :** Règles claires pour chaque étape du processus
- **Boucles de feedback :** Réunions, métriques et révisions régulières

- **Amélioration collaborative** : Évolution progressive basée sur des données empiriques

Métriques clés:

- **Lead time** : Temps total depuis la demande jusqu'à la livraison
- **Cycle time** : Temps de traitement effectif d'une tâche
- **Throughput** : Nombre de tâches complétées par unité de temps
- **Taux de blocage** : Pourcentage de tâches bloquées

Forces: Flexibilité, visualisation immédiate de problèmes, adaptation continue, réduction du gaspillage, prévisibilité accrue

Idéal pour: Environnements avec flux de travail continu, équipes de maintenance, projets à priorités changeantes

5.5 XP (Extreme Programming)

- **Pratiques techniques rigoureuses:**
 - Programmation en binôme
 - Développement piloté par les tests (TDD)
 - Intégration continue
 - Refactoring fréquent
- **Cycles courts** et feedback constant
- **Client sur site** pour consultation immédiate
- **Design simple** et évolutif
- **Forces:** Qualité technique élevée, adaptabilité, réduction des défauts

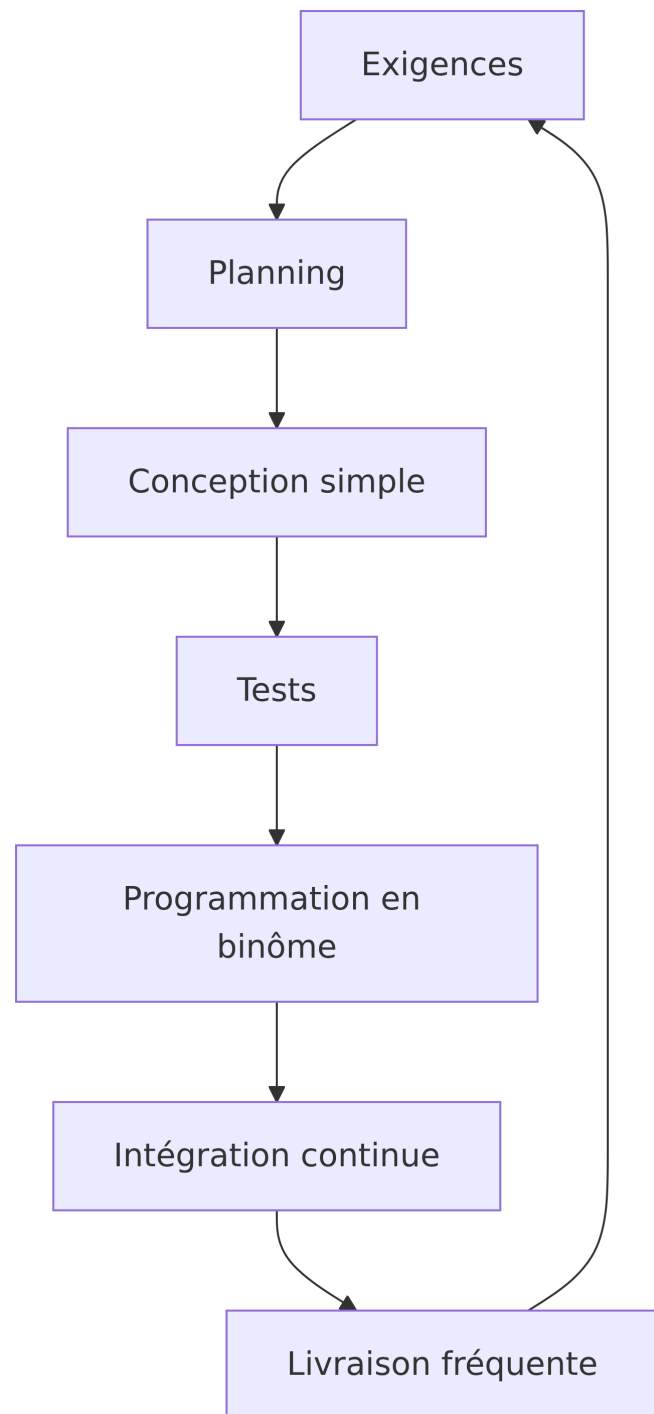


Figure 6. – Processus XP

5.6 Scrumban

- **Hybride** entre Scrum et Kanban

- Combine **itérations** de Scrum et **flux continu** de Kanban
- **Planification à la demande** (quand le backlog s'épuise)
- **Métriques de flux** pour optimiser le processus
- **Forces**: Flexibilité accrue de Scrum, visualisation améliorée

- **Phases**: 1) Planification des besoins → 2) Conception utilisateur (prototypes) → 3) Construction rapide → 4) Transition
- **Forces**: Développement rapide, forte adhésion utilisateur, visibilité précoce des résultats
- **Limites**: Évolutivité limitée, nécessite utilisateurs disponibles et développeurs expérimentés

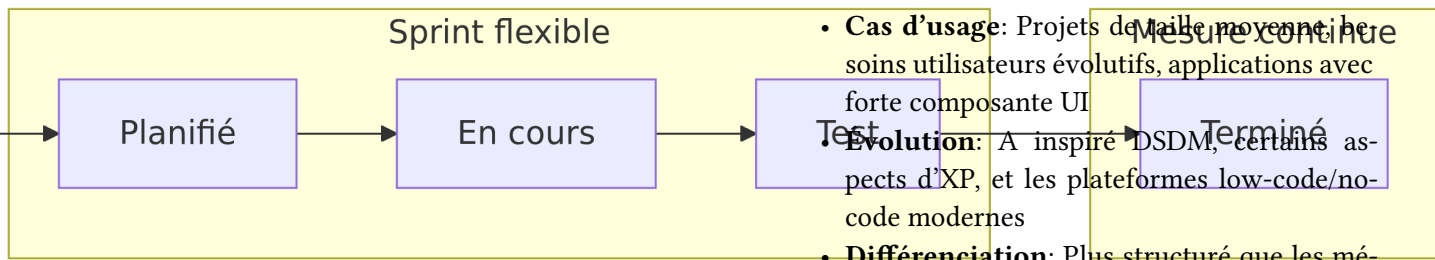


Figure 7. – Processus Scrumban

5.7 Processus unifié

Le Processus Unifié (RUP: Rational Unified

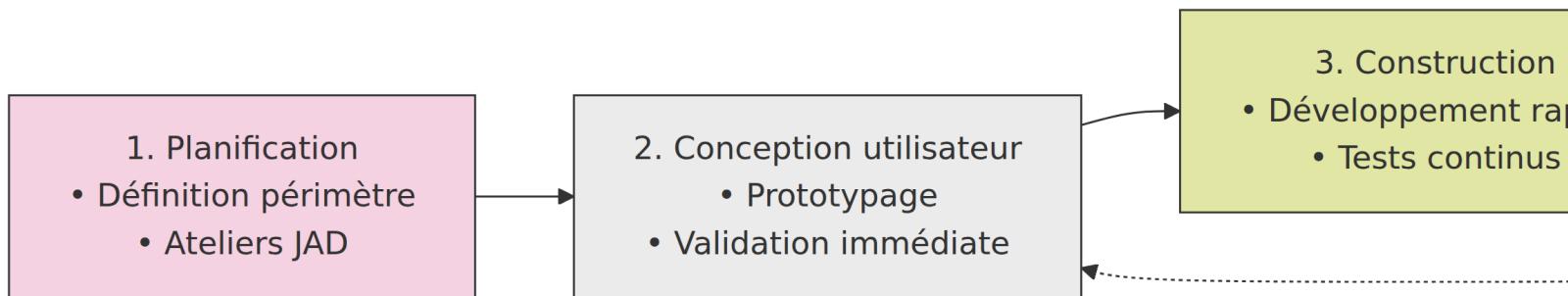


Figure 8. – Processus RAD

- **Disciples**: Modélisation, Implémentation, Test, Déploiement, etc.

5.8 RAD (Rapid Application Development)

- **Définition**: Méthodologie créée par James Martin (1991) pour accélérer le développement
- **Objectif**: Livrer rapidement des applications fonctionnelles (60-90 jours)
- **Principes**: Prototypage rapide, implication continue des utilisateurs, équipes pluridisciplinaires

6 4. Gestion de projet

- **Définition**: Ensemble des activités de planification, d'organisation, de suivi et de contrôle d'un projet de développement logiciel.
- **Objectif**: Livrer un produit conforme aux exigences, dans les délais et le budget alloués.

6.1 Approches de gestion de projet

- **Prédictives (plan-driven)**
 - Approche traditionnelle (cascade, cycle en V)
 - Planification détaillée en amont

- Contrôle du respect du plan
- Adapté aux projets à exigences stables
- Adaptatives (change-driven)
 - Approches agiles (Scrum, Kanban, XP)
 - Planification progressive
 - Inspection et adaptation continues
 - Adapté aux projets à forte incertitude

7 5. Analyse et conception

7.1 5.1 Ingénierie des exigences

- L'ingénierie des exigences comprend:
 1. **Élicitation:** Découvrir les besoins des parties prenantes
 2. **Analyse:** Clarifier et prioriser les exigences
 3. **Spécification:** Documenter les exigences
 4. **Validation:** Vérifier la cohérence et la pertinence
 5. **Gestion:** Suivre l'évolution des exigences

7.2 Techniques de collecte des besoins

- Interviews et questionnaires
- Ateliers et brainstorming
- Observation et ethnographie
- Prototypage et scénarios d'utilisation
- Analyse de documents existants

! Règles SMART pour les exigences

- Spécifique
- Mesurable
- Atteignable
- Réaliste
- Temporellement défini

7.3 5.2 Conception logicielle et UML

8 Modélisation avec UML

Le langage UML (Unified Modeling Language) est une notation standardisée pour visualiser, spécifier, construire et documenter les artefacts d'un système logiciel. Développé dans les années 1990 et maintenu par l'OMG (Object Management Group), il constitue aujourd'hui le standard de facto pour la modélisation orientée objet.

9 Diagrammes structurels

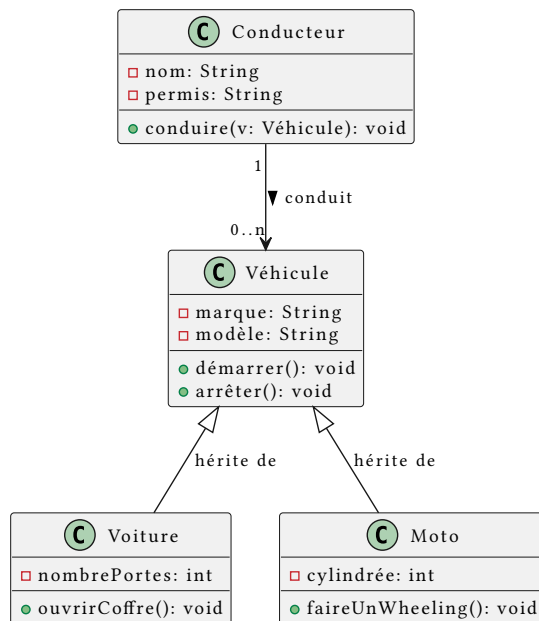
- Les diagrammes structurels représentent l'architecture statique du système
- Ils montrent les classes, interfaces, composants, packages, etc.
- Ils incluent les relations, attributs, méthodes, etc.
- Ils sont utiles pour la conception et la documentation
- Principaux types: diagramme de classes, diagramme d'objets, diagramme de composants, diagramme de déploiement, diagramme de paquetages

9.1 Diagramme de classes

Objectif : Représenter les classes, leurs attributs, méthodes et relations.

Éléments clés :

- Classes et interfaces
- Attributs et méthodes avec visibilité
- Relations : association, agrégation, composition, héritage, etc.
- Cardinalités et rôles

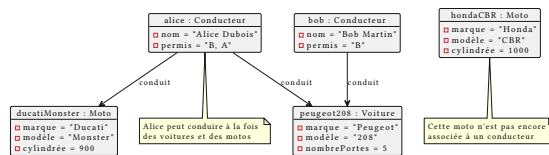


9.2 Diagramme d'objets

Objectif : Montrer une « photo instantanée » d'instances de classes et leurs relations à un moment donné.

Éléments clés :

- Instances d'objets nommées
- Valeurs d'attributs concrètes
- Liens entre objets (instances d'associations)

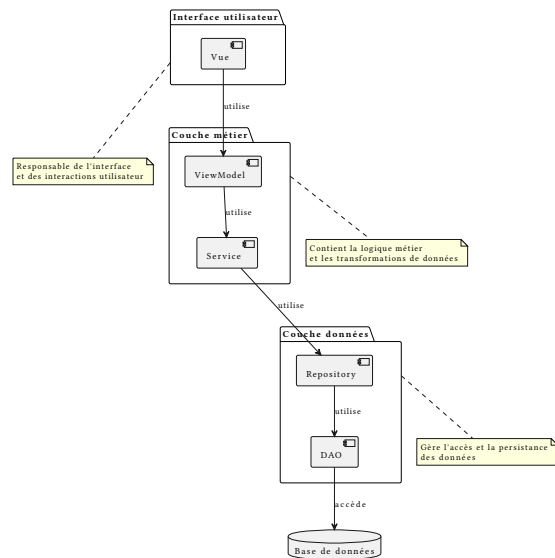


9.3 Diagramme de composants

Objectif : Représenter l'organisation physique des composants d'un système et leurs dépendances.

Éléments clés :

- Composants logiciels
- Interfaces fournies et requises
- Relations de dépendance



9.4 Diagramme de déploiement

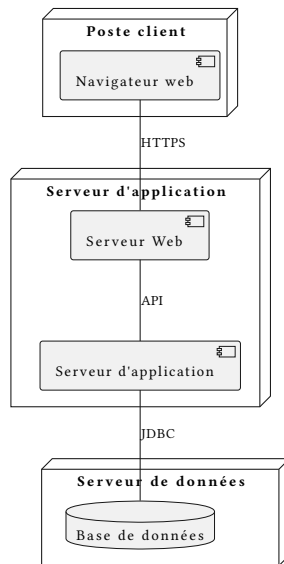
Objectif : Montrer la configuration physique du matériel et la distribution des composants logiciels.

Éléments clés :

- Nœuds (hardware)
- Artefacts (composants déployés)
- Connexions réseau

Architecture à 3 niveaux - Déploiement physique

Diagramme de déploiement montrant la répartition des composants sur différents serveurs avec les protocoles de communication



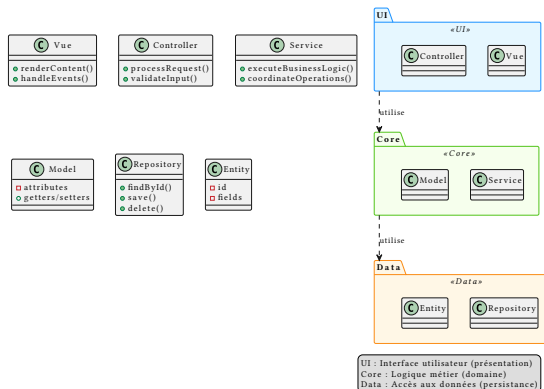
9.5 Diagramme de paquetages

Objectif : Organiser les éléments du modèle en groupes pour simplifier les diagrammes complexes.

Éléments clés :

- Paquetages (namespaces/modules)
- Dépendances entre paquetages
- Visibilité des éléments

Architecture logicielle en couches avec séparation des responsabilités



10 Diagrammes comportementaux

- Les diagrammes comportementaux illustrent le fonctionnement dynamique du système
- Ils montrent les interactions entre objets, les flux de contrôle, les états, etc.
- Ils sont utiles pour comprendre le comportement du système et valider les exigences
- Principaux types: diagramme de cas d'utilisation, diagramme de séquence, diagramme d'activité, diagramme d'état, diagramme de communication

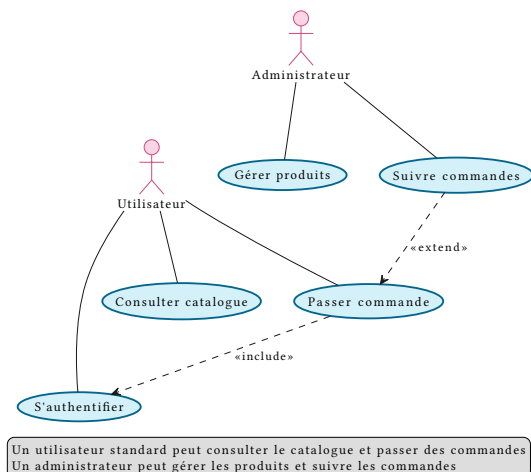
10.1 Diagramme de cas d'utilisation

Objectif : Décrire les fonctionnalités du système du point de vue utilisateur.

Éléments clés :

- Acteurs (utilisateurs ou systèmes externes)
- Cas d'utilisation (fonctionnalités)
- Relations entre cas d'utilisation (inclusion, extension)

Diagramme de cas d'utilisation - Système e-commerce

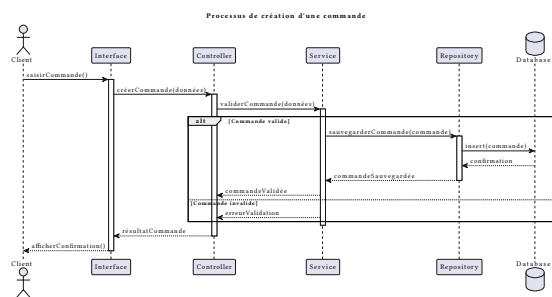


10.2 Diagramme de séquence

Objectif : Modéliser les interactions entre objets dans le temps, montrant l'ordre des messages échangés.

Éléments clés :

- Objets (instances de classes)
- Lignes de vie et messages
- Activations (périodes d'exécution)
- Fragments d'interaction (alternatives, boucles)



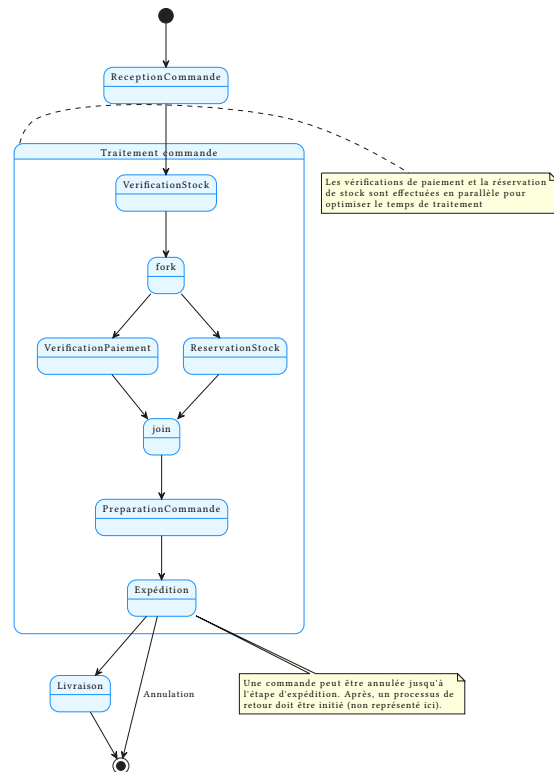
10.3 Diagramme d'activité

Objectif : Décrire les flux de travail, processus métier ou algorithmes avec leurs points de décision.

Éléments clés :

- Actions et activités
- Transitions et conditions
- Bifurcations et jointures (parallélisme)
- Partitions (swimlanes)

Traitement d'une commande - Processus avec tâches parallèles

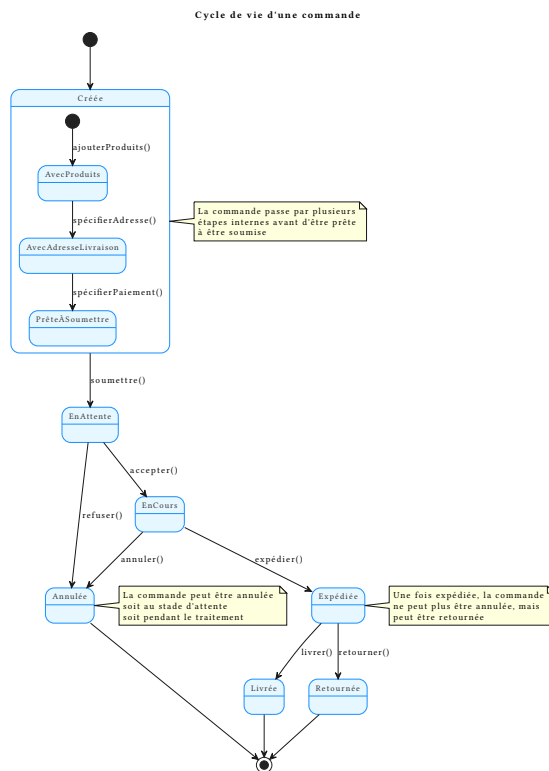


10.4 Diagramme d'état

Objectif : Illustrer les différents états d'un objet au cours de son cycle de vie et les transitions entre ces états.

Éléments clés :

- États (représentant une condition ou situation)
- Transitions (changements d'état)
- Événements déclencheurs
- Actions et activités dans les états

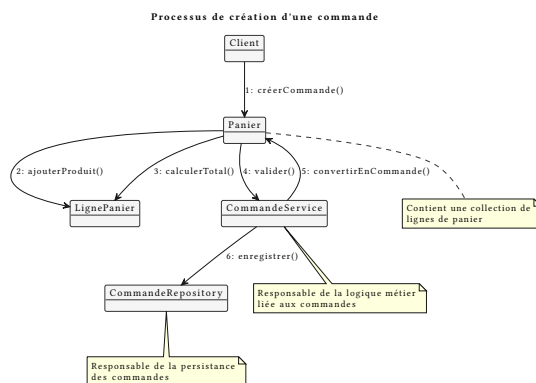


10.5 Diagramme de communication

Objectif : Montrer les interactions entre objets en se concentrant sur la structure des échanges plutôt que sur leur chronologie.

Éléments clés :

- Objets et leurs liens
- Messages numérotés
- Conditions et itérations



11 Bonnes pratiques UML

1. **Niveau de détail adapté** : Inclure uniquement les informations pertinentes pour l'audience cible
2. **Cohérence** : Maintenir une terminologie et des conventions cohérentes entre diagrammes
3. **Modularité** : Diviser les modèles complexes en sous-modèles plus simples
4. **Annotation** : Ajouter des notes pour clarifier les aspects ambigus
5. **Évolution progressive** : Commencer par des modèles simples et les affiner itérativement

12 Outils de modélisation UML

- **Outils dédiés** : Enterprise Architect, Visual Paradigm, Modelio
- **IDE avec support UML** : IntelliJ IDEA, Eclipse avec plugins
- **Outils collaboratifs** : Lucidchart, draw.io
- **Langages textuels** : PlantUML, Mermaid

13 6. Implémentation et tests

13.1 Implémentation

- **Définition**: Phase de transformation des spécifications en code exécutable
- **Objectif**: Produire un logiciel fonctionnel selon les spécifications établies
- **Résultats**: Code source, documentation technique, tests automatisés

13.2 Bonnes pratiques de programmation

- **DRY** (Don't Repeat Yourself): Éviter la duplication de code
- **KISS** (Keep It Simple, Stupid): Privilégier la simplicité
- **YAGNI** (You Aren't Gonna Need It): Ne pas anticiper des besoins hypothétiques

- **Separation of Concerns:** Diviser le code en responsabilités distinctes
- **Fail Fast:** Détecter et signaler les erreurs au plus tôt

14 Tests et Qualité

14.1 Assurance qualité approfondie

L'assurance qualité englobe plusieurs aspects:

1. **Stratégie de test**
 - Pyramide de tests (unitaires, intégration, système)
 - Couverture du code
 - Automatisation
2. **Métriques**
 - Complexité cyclomatique (Mesure de la complexité d'un programme informatique)
 - Dette technique (Coût de la réparation des problèmes non résolus)
 - Taux de défauts (Nombre de défauts par ligne de code)

15 7. Déploiement et maintenance

15.1 Gestion des versions

La gestion des versions est un aspect crucial du développement logiciel moderne, permettant de:

- Suivre l'évolution du code source et des composants
- Coordonner le travail des équipes
- Faciliter la livraison de nouvelles fonctionnalités
- Assurer la traçabilité des changements
- Maintenir plusieurs versions du produit

15.2 Versionnement sémantique (SemVer)

- Principe: MAJOR.MINOR.PATCH
 - **MAJOR:** Changements incompatibles avec l'API précédente
 - **MINOR:** Ajouts de fonctionnalités compatibles
 - **PATCH:** Corrections de bugs compatibles
- Exemples:
 - **0.0.1:** Version initiale
 - **1.0.0:** Version initiale stable
 - **1.1.0:** Ajout de nouvelles fonctionnalités
 - **1.1.1:** Correction de bugs
 - **2.0.0:** Changements cassant la compatibilité
- Suffixes additionnels:
 - **-alpha:** Version instable en développement interne
 - **-beta:** Version en test externe
 - **-rc.1:** Release candidate (quasi-finale)

15.3 Systèmes de gestion de versions (VCS)

- **Centralisés:** SVN, CVS
 - Serveur central unique
 - Check-out, check-in
 - Risque de conflits
- **Décentralisés:** Git, Mercurial
 - Copie locale complète
 - Branches, merges, commits locaux
 - Facilité de collaboration

15.4 Workflows de développement

- Définition: organisation des branches et des cycles de vie du code source
- Objectifs: Isoler les développements, faciliter les tests, garantir la stabilité
- Exemples:
 - **GitFlow:** Branches principales (main, develop) et branches de support (feature, release, hotfix)

- **Feature Flow:** Branches principales (main) et branches de fonctionnalités temporaires
- **Forking Workflow:** Forks et pull requests pour contribuer à un projet

16 Intégration continue (CI) et livraison continue (CD)

- Intégration continue (CI)
 - Build et tests automatiques à chaque commit
 - Détection rapide des problèmes d'intégration
 - Validation de la qualité du code (linting, tests unitaires)
- Livraison continue
 - Package automatique en artefacts déployables
 - Tests d'intégration et système automatisés
 - Validation par les équipes QA

17 Déploiement continu (CD)

- Mise en production automatique après validation CI
- Déploiement progressif (canary, blue-green)
- Surveillance et rollback automatique si problème

18 8. Pratiques modernes

18.1 8.1 Gestion de versions

18.2 8.2 Intégration et déploiement continus

19 Gestion de projet

- Définition: Ensemble des activités de planification, d'organisation, de suivi et de contrôle d'un projet de développement logiciel.
- Objectif: Livrer un produit conforme aux exigences, dans les délais et le budget alloués.

19.1 Approches de gestion de projet

- Prédictives (plan-driven)
 - Approche traditionnelle (cascade, cycle en V)
 - Planification détaillée en amont
 - Contrôle du respect du plan
 - Adapté aux projets à exigences stables
- Adaptatives (change-driven)
 - Approches agiles (Scrum, Kanban, XP)
 - Planification progressive
 - Inspection et adaptation continues
 - Adapté aux projets à forte incertitude

20 Analyse et spécification des besoins

- L'ingénierie des exigences comprend:
 1. **Élicitation:** Découvrir les besoins des parties prenantes
 2. **Analyse:** Clarifier et prioriser les exigences
 3. **Spécification:** Documenter les exigences
 4. **Validation:** Vérifier la cohérence et la pertinence
 5. **Gestion:** Suivre l'évolution des exigences

20.1 Techniques de collecte des besoins

- Interviews et questionnaires
- Ateliers et brainstorming
- Observation et ethnographie
- Prototypage et scénarios d'utilisation
- Analyse de documents existants

! Règles SMART pour les exigences

- Spécifique
- Mesurable
- Atteignable
- Réaliste
- Temporellement défini

21 Conception logicielle

- Définition: Processus de transformation des exigences en représentation implémentable
- Objectif: Créer un plan détaillé pour guider l'implémentation
- Position: Phase intermédiaire entre analyse des besoins et codage
- Résultat: Ensemble de modèles, schémas et spécifications techniques

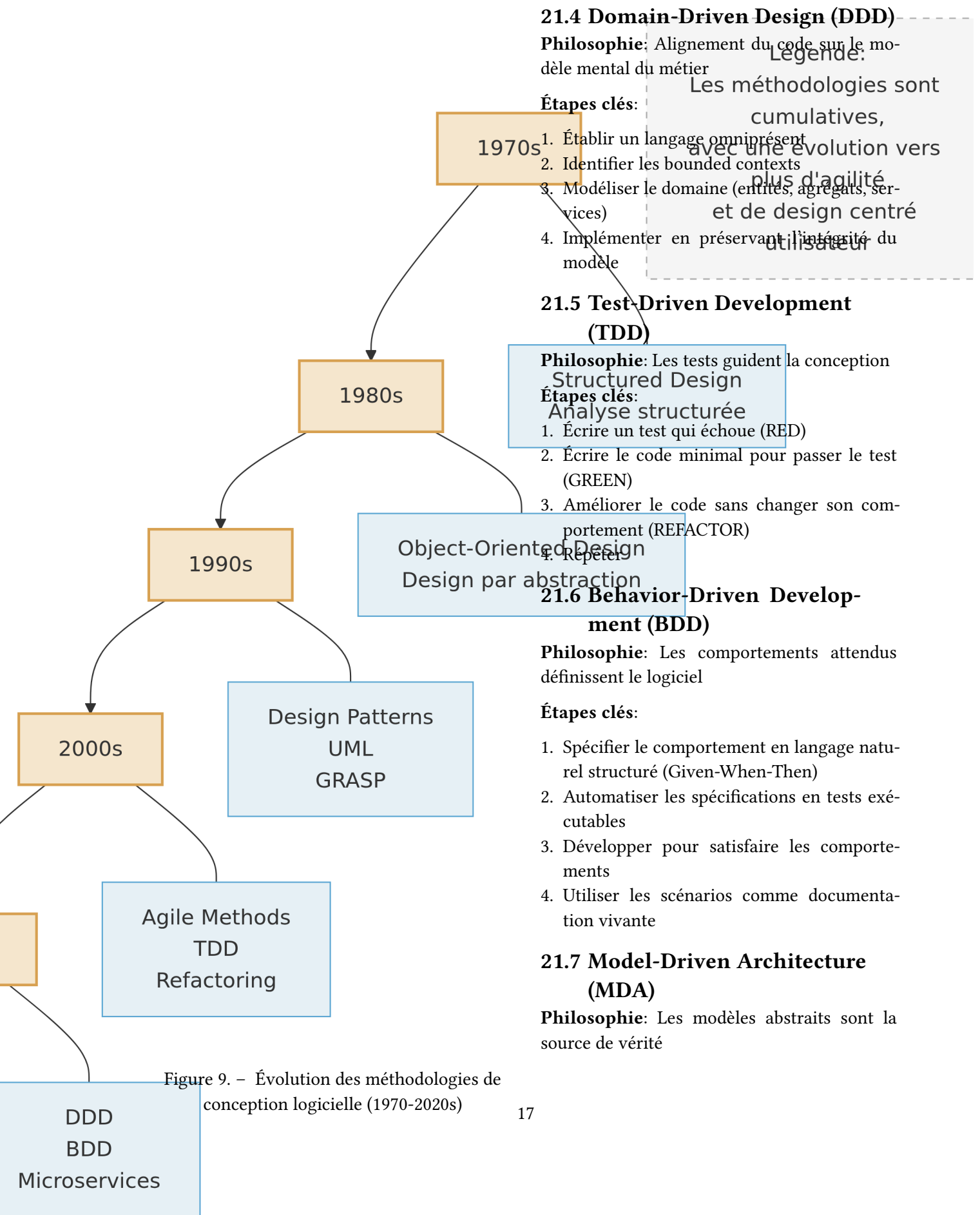
21.1 Approches fondamentales

- Conception descendante (Top-Down)
 - Commence par la vision globale, Décompose progressivement en sous-systèmes, Favorise la compréhension des objectifs généraux
- Conception ascendante (Bottom-Up)
 - Commence par les composants de bas niveau, Les assemble progressivement en systèmes plus grands, Favorise la réutilisation et le pragmatisme
- Conception orientée objet (OOD)
 - Organisation autour d'objets plutôt que d'actions, Encapsulation des données avec comportements associés, Relations basées sur héritage, composition, agrégation

21.2 Niveaux de conception

- Architecture logicielle
 - Structure globale du système
 - Décomposition en sous-systèmes/modules
 - Définition des interfaces principales
 - Choix des styles architecturaux (couches, microservices, etc.)
- Conception détaillée
 - Spécification précise des classes/modules
 - Algorithmes et structures de données
 - Protocoles de communication
 - Relations entre composants

21.3 Méthodologies de conception logicielle



Étapes clés:

1. Créer un modèle indépendant de la plateforme (PIM)
2. Transformer en modèle spécifique à la plateforme (PSM)
3. Générer le code à partir du PSM
4. Affiner et compléter l'implémentation

21.8 Feature-Driven Development (FDD)

Philosophie: Organiser le développement autour des fonctionnalités utilisateur

Étapes clés:

1. Développer un modèle global
2. Construire une liste de fonctionnalités
3. Planifier par fonctionnalité
4. Concevoir par fonctionnalité
5. Implémenter par fonctionnalité

21.9 Continuous Design

Philosophie: La conception évolue progressivement avec le code

Étapes clés:

1. Commencer avec un design minimal viable
2. Intégrer continuellement les changements
3. Refactoriser régulièrement
4. Utiliser les métriques pour guider l'évolution
5. Adapter la conception aux nouveaux besoins

21.10 Event Storming

Philosophie: Exploration collaborative du domaine par les événements

Étapes clés:

1. Identifier les événements du domaine (posts)
2. Ajouter les commandes qui déclenchent ces événements
3. Identifier les agrégats et règles métier

4. Organiser en bounded contexts
5. Dériver le modèle de domaine

22 Planification et estimation

- Techniques d'estimation
 - Analogique: Comparaison avec projets similaires
 - Paramétrique: Utilisation de métriques (points de fonction, lignes de code)
 - Bottom-up: Décomposition et estimation des éléments individuels
 - Expert: Jugement basé sur l'expérience
 - Planning poker: Estimation collaborative et consensuelle
- Outils de planification
 - WBS (Work Breakdown Structure) : Décomposition hiérarchique des tâches
 - Diagrammes de Gantt (PERT, CPM) : Visualisation des dépendances et durées
 - Backlog produit et planification de sprint : Liste des fonctionnalités et ordonnancement par priorité
 - Story mapping : Visualisation des fonctionnalités par scénarios utilisateur
 - Méthode du chemin critique (CPM) : Identification des tâches critiques
 - Story mapping : Visualisation des fonctionnalités par scénarios utilisateur

23 Maintenance et évolution

- Types de maintenance
 - **Corrective:** Correction des défauts
 - **Adaptative:** Adaptation aux changements d'environnement
 - **Perfective:** Amélioration des fonctionnalités
 - **Préventive:** Prévention des problèmes futurs