

Software Engineering 101

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Génie Logiciel : Fondamentaux et Pratiques Modernes	2
1.1 Introduction au génie logiciel	2
1.1.a Le génie logiciel aujourd'hui . .	2
1.1.b Définition et objectifs du génie logiciel	3
1.1.c Qualités d'un logiciel	3
1.1.d Principes de base du développement logiciel	4
1.1.e Cycle de vie du logiciel	4
1.2 Processus de développement	5
1.2.a Évolution des méthodologies . .	5
1.2.b Modèles classiques (Waterfall, V-model)	5
1.3 Waterfall	5
1.4 V-Model	5
1.4.a Approches agiles (Scrum, Kanban)	6
1.4.b Scrum	6
1.4.c Kanban	6
1.4.d XP (Extreme Programming) . .	7
1.4.e Scrumban	7
1.4.f Processus unifié	8
1.4.g RAD (Rapid Application Development)	8
1.5 Analyse et spécification des besoins .	8
1.5.a Techniques de collecte des besoins	9
1.6 Conception logicielle	9
1.6.a Approches fondamentales	9
1.6.b Niveaux de conception	9
1.6.c Méthodologies de conception logicielle	9
1.6.d Domain-Driven Design (DDD) .	9
1.6.e Test-Driven Development (TDD)	9
1.6.f Behavior-Driven Development (BDD)	10
1.6.g Model-Driven Architecture (MDA)	10
1.6.h Feature-Driven Development (FDD)	10
1.6.i Continuous Design	10
1.6.j Event Storming	10
1.7 Modélisation avec UML	10
1.7.a Diagrammes structurels	10
1.8 Diagramme de classes	10
1.9 Diagramme d'objets	11
1.10 Diagramme de composants	11
1.11 Diagramme de déploiement	12
1.12 Diagramme de paquetages	12
1.12.a Diagrammes comportementaux	13
1.13 Diagramme de cas d'utilisation . .	13
1.14 Diagramme de séquence	13
1.15 Diagramme d'activité	14
1.16 Diagramme d'état	15
1.17 Diagramme de communication . .	16
1.17.a Bonnes pratiques UML	16
1.17.b Outils de modélisation UML .	16
1.18 Implémentation	17

1.18.a	Bonnes pratiques de programmation	17
1.19	Tests et Qualité	17
1.19.a	Assurance qualité approfondie	17
1.20	Gestion de projet logiciel	17
1.20.a	Approches de gestion de projet	17
1.21	Planification et estimation	17
1.22	Maintenance et évolution	17
1.22.a	Refactoring	18
1.23	La gestion des versions	18
1.24	Versionnement sémantique (SemVer)	18
1.24.a	Systèmes de gestion de versions (VCS)	18
1.24.b	Workflows de développement	18
1.25	Intégration continue (CI) et livraison continue (CD)	18
1.26	Déploiement continu (CD)	19

logies - Systèmes hautement concurrents - Sécurité renforcée

2. Contraintes du marché - Time-to-market réduit - Innovation continue - Adaptation rapide aux changements - Compétition mondiale

3. Exigences de qualité - Fiabilité critique - Performance optimale - Sécurité robuste - Expérience utilisateur fluide

4. Pratiques modernes - Agilité et DevOps - Architecture cloud-native - Automatisation poussée - Développement durable

1 Génie Logiciel : Fondamentaux et Pratiques Modernes

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

1.1 Introduction au génie logiciel

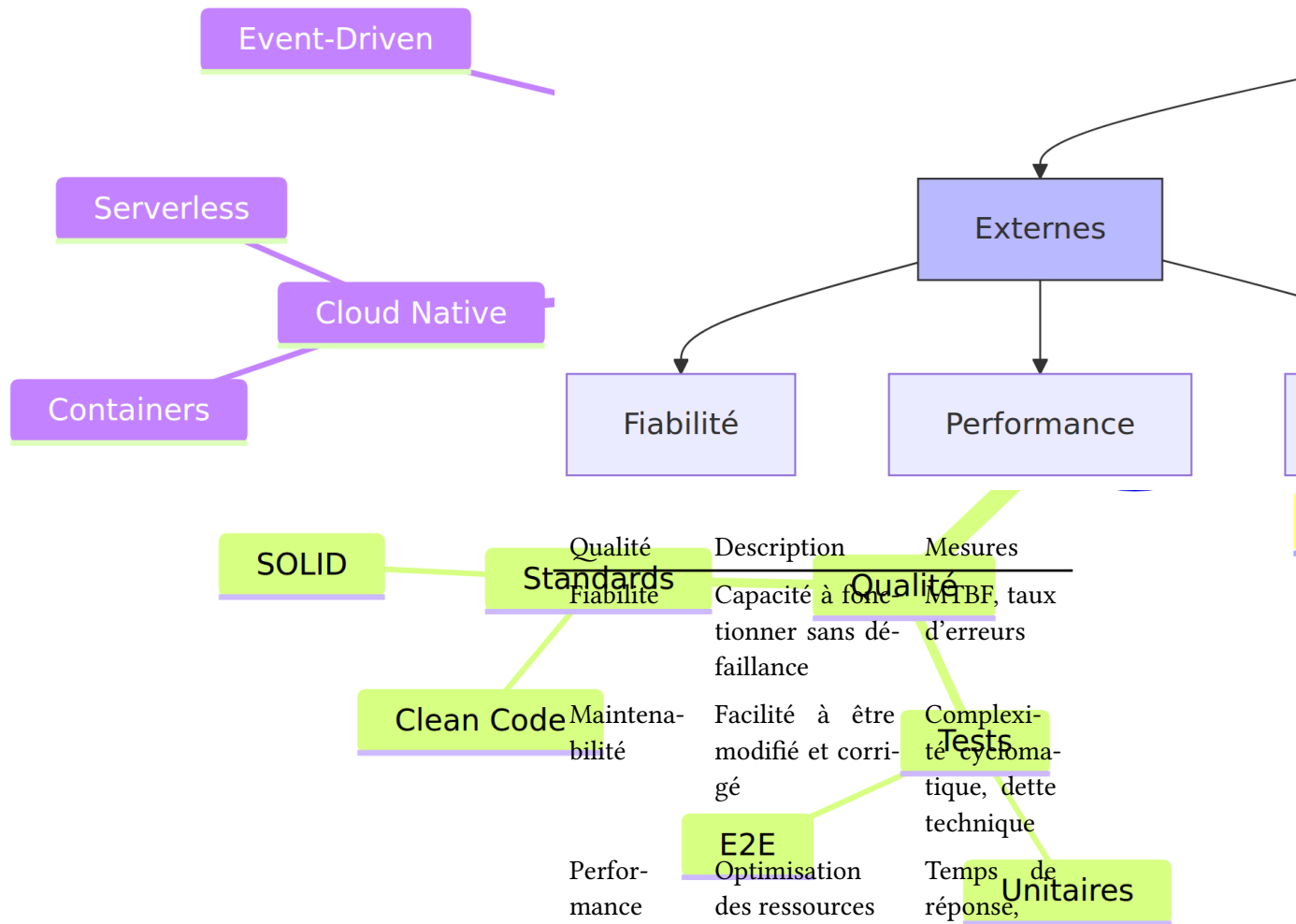
1.1.a Le génie logiciel aujourd'hui

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 techno-

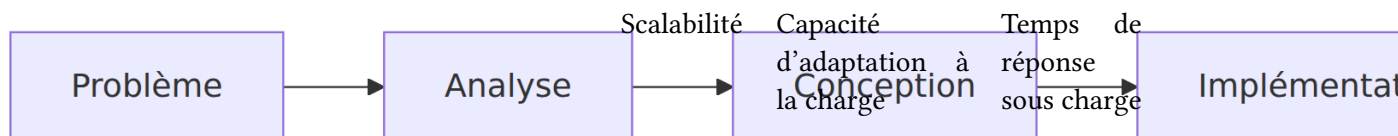
1.1.c Qualités d'un logiciel

Les qualités essentielles d'un logiciel moderne s'articulent autour de plusieurs dimensions.



1.1.b 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.



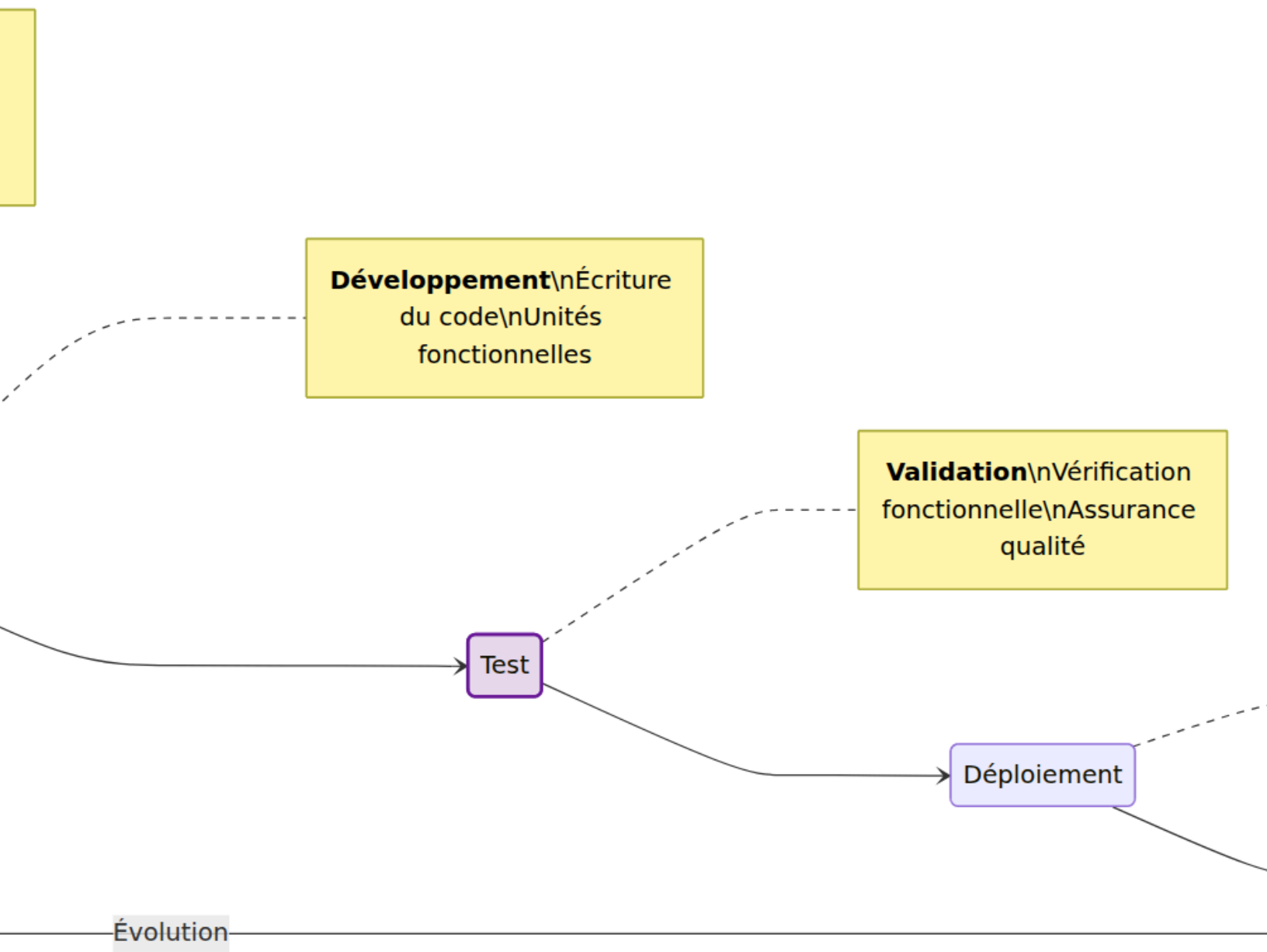


Figure 1. – Cycle de vie du logiciel

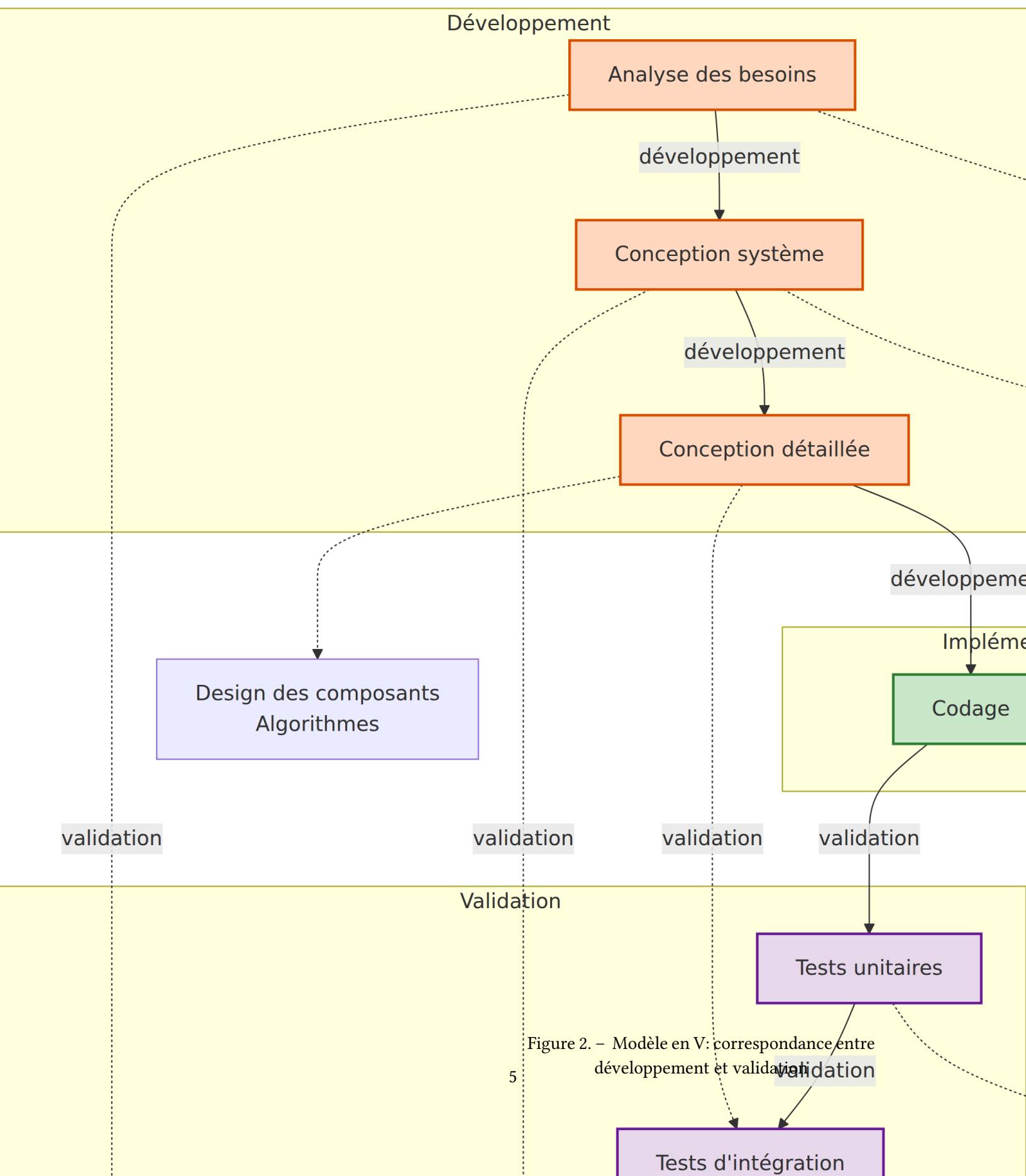


Figure 2. – Modèle en V: correspondance entre développement et validation

Avantages: Emphase sur les tests, traçabilité
Inconvénients: Rigidité, adaptation difficile au changement

1.4.a Approches agiles (Scrum, Kanban)

Les méthodes agiles favorisent:

- Livraison incrémentale et itérative
- Adaptation aux changements
- Collaboration étroite avec le client
- Amélioration continue

💡 Manifeste Agile

- **Individus et interactions** plutôt que processus et outils
- **Logiciel fonctionnel** plutôt que documentation exhaustive
- **Collaboration avec le client** plutôt que négociation de contrat
- **Adaptation au changement** plutôt que suivi d'un plan

1.4.b Scrum

- **Sprints:** Cycles de travail de 2-4 semaines
- **Rôles clés:**
 - 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

Voici une amélioration de la section Kanban sous forme d'un slide complet et visuellement engageant :

1.4.c Kanban

Tableau Kanban avec limites WIP

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:

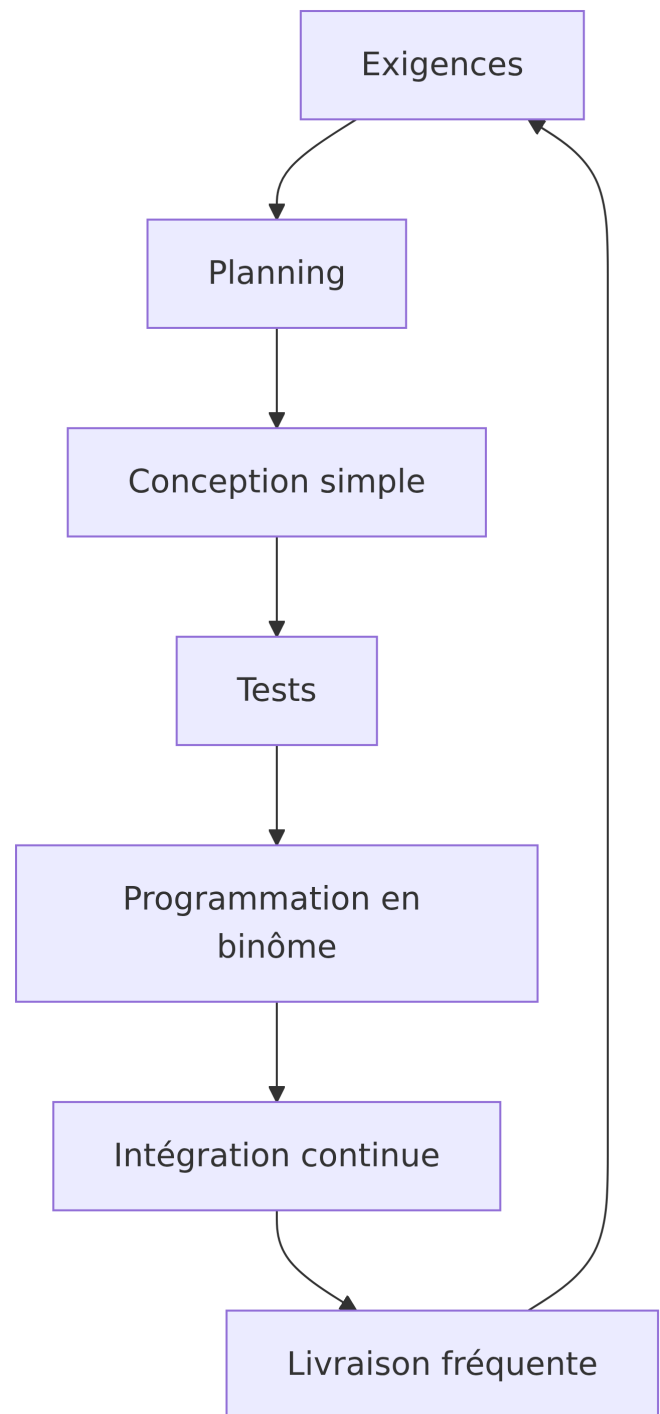
- **Lead time** : Temps total depuis la demande jusqu'à la livraison
- **Cycle time** : Temps de traitement effectif d'une tâche
- **Throughput** : Nombre de tâches complétées par unité de temps
- **Taux de blocage** : Pourcentage de tâches bloquées

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

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

1.4.d XP (Extreme Programming)

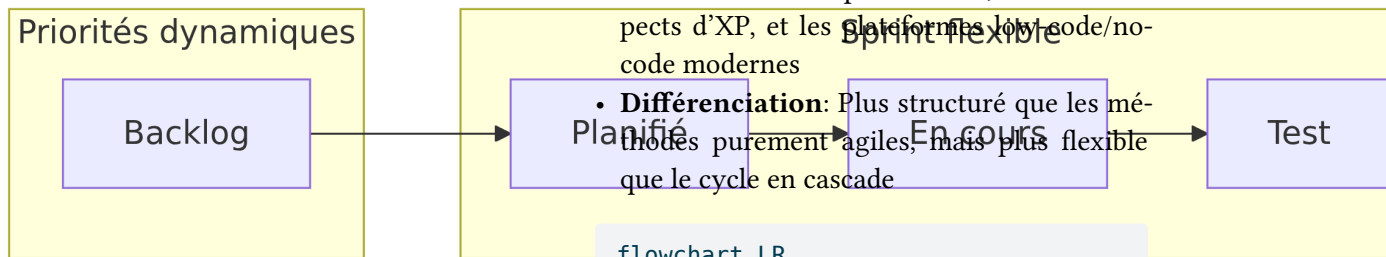
- **Pratiques techniques rigoureuses:**
 - Programmation en binôme
 - Développement piloté par les tests (TDD)
 - Intégration continue
 - Refactoring fréquent
- **Cycles courts** et feedback constant
- **Client sur site** pour consultation immédiate
- **Design simple** et évolutif
- **Forces:** Qualité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



1.4.e 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
- **Limites:** Évolutivité limitée, nécessite utilisateurs disponibles et développeurs expérimentés
- **Cas d'usage:** Projets de taille moyenne, besoins utilisateurs évolutifs, applications avec forte composante UI
- **Évolution:** A inspiré DSDM, certains aspects d'XP, et les plateformes low-code/no-code modernes



1.4.f Processus unifié

Le Processus Unifié (RUP) combine aspects itératifs et incrémentiels:

- **Phases:** Inception, Élaboration, Construction, Transition
- **Aspects:** Risques, Architecture, Valeur métier, Compétences
- **Disciples:** Modélisation, Implémentation, Test, Déploiement, etc.

1.4.g 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
- **Phases:** 1) Planification des besoins → 2) Conception utilisateur (prototypes) → 3) Construction rapide → 4) Transition
- **Forces:** Développement rapide, forte adhésion utilisateur, visibilité précoce des résultats

1.5 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

1.5.a 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

1.6 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

1.6.a 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

1.6.b 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

1.6.c Méthodologies de conception logicielle

```
timeline
    title Évolution des méthodologies de conception
    1970s : Structured Design
    1980s : Object-Oriented Design
    1990s : Design Patterns : UML
    2000s : Agile Methods : TDD : Refactoring
    2010s :DDD : BDD : Microservices
    2020s : API-first : Event-driven : Continuous Design
```

1.6.d Domain-Driven Design (DDD)

Philosophie: Alignement du code sur le modèle mental du métier

Étapes clés: 1. Établir un langage omniprésent 2. Identifier les bounded contexts 3. Modéliser le domaine (entités, agrégats, services) 4. Implémenter en préservant l'intégrité du modèle

1.6.e Test-Driven Development (TDD)

Philosophie: Les tests guident la conception

Étapes clés: 1. Écrire un test qui échoue (RED) 2. Écrire le code minimal pour passer le test (GREEN) 3. Améliorer le code sans changer son comportement (REFACTOR) 4. Répéter

1.6.f Behavior-Driven Development (BDD)

Philosophie: Les comportements attendus définissent le logiciel

Étapes clés: 1. Spécifier le comportement en langage naturel structuré (Given-When-Then) 2. Automatiser les spécifications en tests exécutables 3. Développer pour satisfaire les comportements 4. Utiliser les scénarios comme documentation vivante

1.6.g Model-Driven Architecture (MDA)

Philosophie: Les modèles abstraits sont la source de vérité

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

1.6.h 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

1.6.i 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

1.6.j Event Storming

Philosophie: Exploration collaborative du domaine par les événements

Étapes clés:

1. Identifier les événements du domaine (post-its)
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

Voici une section améliorée et détaillée sur la modélisation UML pour votre cours :

1.7 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.

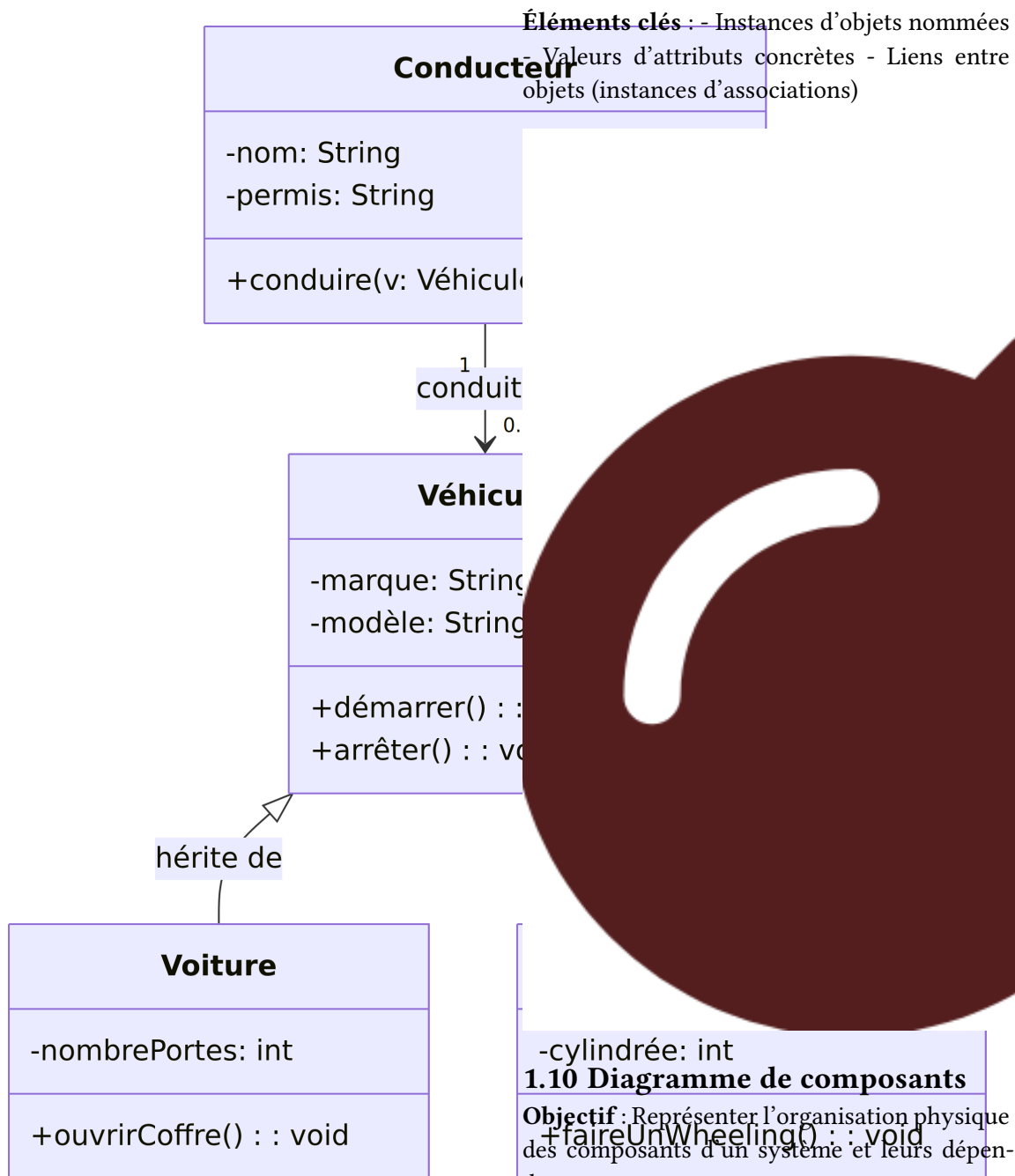
1.7.a Diagrammes structurels

Les diagrammes structurels représentent l'architecture statique du système :

1.8 Diagramme de classes

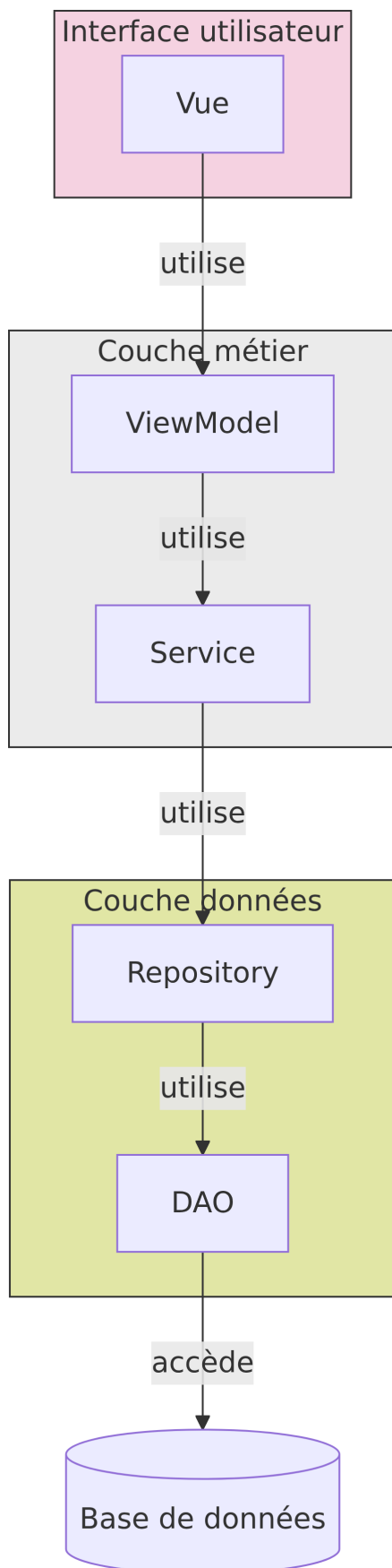
Objectif : Représenter les classes, leurs attributs, méthodes et relations.

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



1.9 Diagramme d'objets

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



1.11 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



1.12 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

tionnalités) - Relations entre cas d'utilisation (inclusion, extension)



1.12.a Diagrammes comportementaux

Les diagrammes comportementaux illustrent le fonctionnement dynamique du système :

1.13 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 (fonc-

1.14 Diagramme de séquence

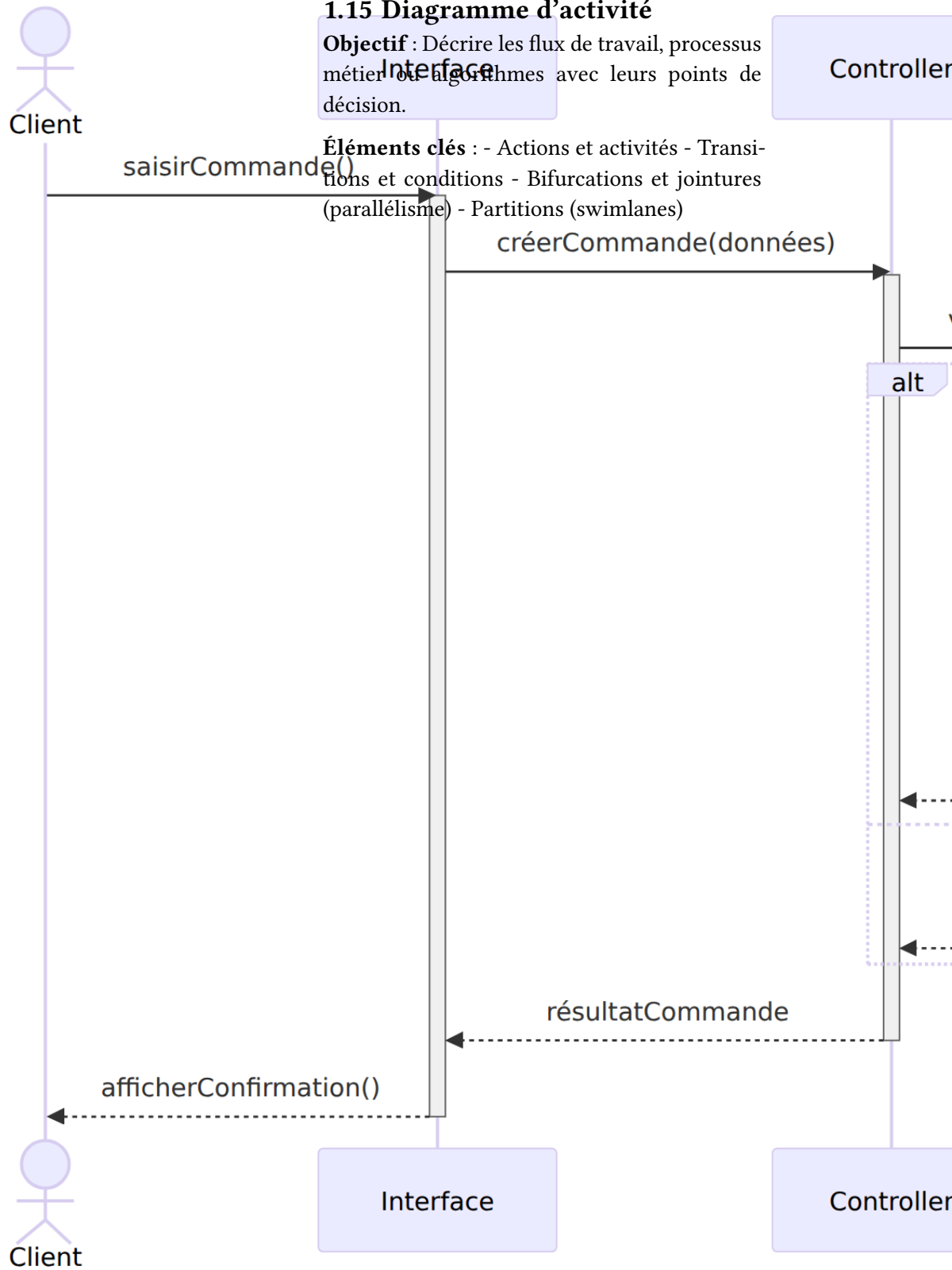
Objectif : Modéliser les interactions entre objets dans le temps, montrant l'ordre des messages échangés.

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

1.15 Diagramme d'activité

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

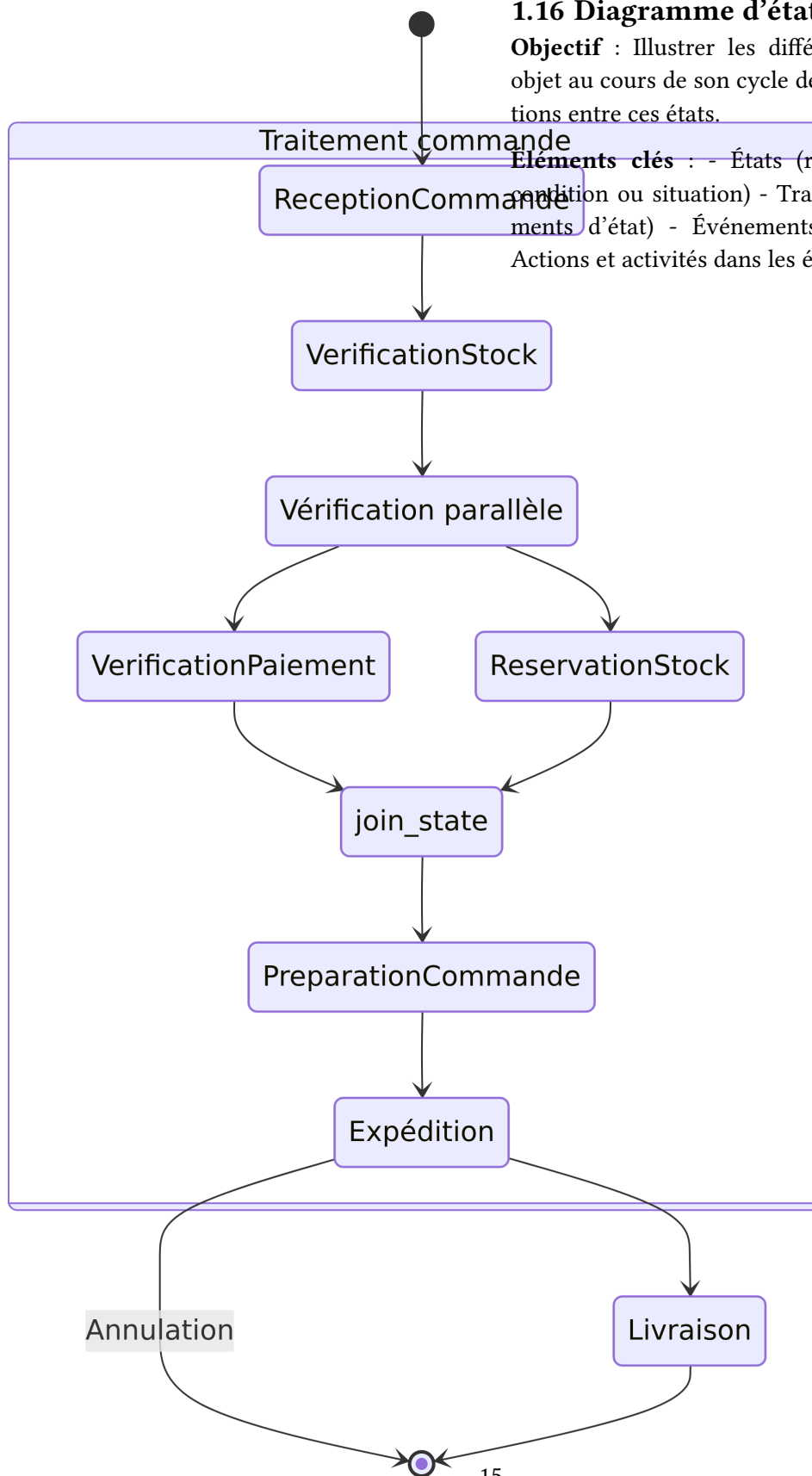
Éléments clés : - Actions et activités - Transitions et conditions - Bifurcations et jointures (parallélisme) - Partitions (swimlanes)



1.16 Diagramme d'état

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

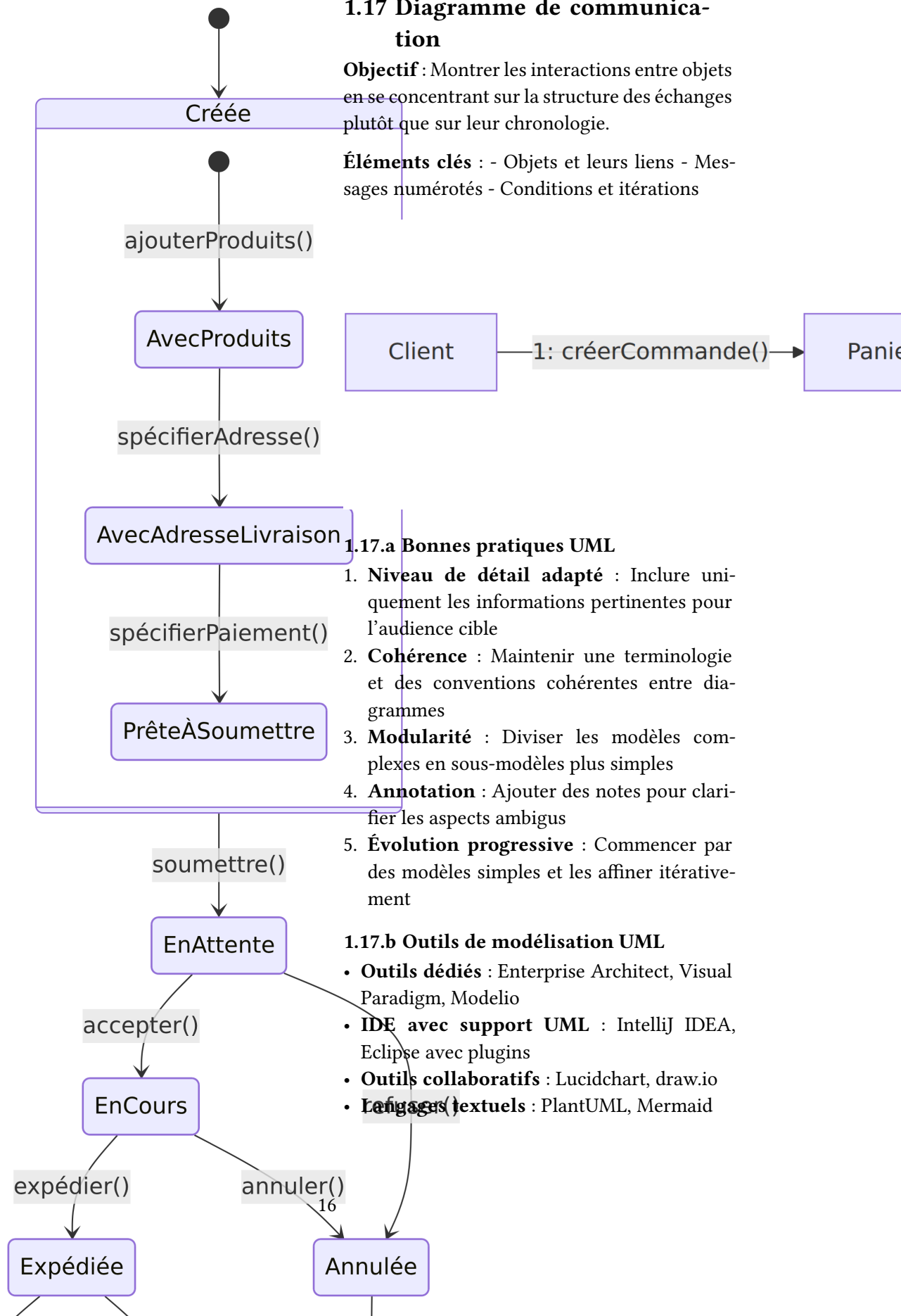
Éléments clés : - États (représentant une condition ou situation) - Transitions (changements d'état) - Événements déclencheurs - Actions et activités dans les états



1.17 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



1.17.a 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

1.17.b 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

1.18 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

1.18.a 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

1.19 Tests et Qualité

1.19.a Assurance qualité approfondie

L'assurance qualité englobe plusieurs aspects:

1. **Stratégie de test**
 - Pyramide de tests
 - Couverture du code
 - Automatisation
2. **Métriques**
 - Complexité cyclomatique
 - Dette technique
 - Taux de défauts

1.20 Gestion de projet logiciel

- **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.

1.20.a 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

1.21 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

1.22 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

1.22.a Refactoring

Le refactoring améliore la structure du code sans en modifier le comportement externe:

- Extraction de méthode
- Simplification des conditions
- Élimination du code dupliqué
- Réorganisation des classes

1.23 La 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

1.24 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)

1.24.a 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

1.24.b 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

1.25 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

1.26 Déploiement continu (CD)

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