

Analyse : de l'exigence à la spécification

Quatre notes de cours pour réviser la méthode

BUT Informatique, début de deuxième année

24 septembre 2026

Résumé

Ces notes expliquent les notions et les démarches indépendamment du projet de TD. Le cas fictif d'une bibliothèque permet de suivre le même raisonnement d'un chapitre à l'autre. Les exemples C illustrent les contrats, les données et les décisions. Le fascicule de TD/TP donne séparément le travail à effectuer sur le dossier BioSeq.

Table des matières

1 Comprendre le besoin et vérifier une exigence	2
1.1 Demande, besoin et exigence	2
1.2 Les informations n'ont pas toutes le même statut	2
1.3 Exigences fonctionnelles et exigences de qualité	2
1.4 Formuler une exigence vérifiable	2
1.5 Lecture C : structure, en-tête et appel	2
1.6 Exercice corrigé	3
2 Formaliser un scénario et un contrat	4
2.1 La story exprime l'utilité, ses critères précisent l'acceptation	4
2.2 Le cas d'utilisation détaille un objectif complet	4
2.3 Isoler une décision locale	4
2.4 Le contrat C précise les adresses et les effets	4
2.5 Choisir un test discriminant	5
2.6 Exercice corrigé	5
3 Décrire les exceptions et les rendre testables	6
3.1 Les exceptions font partie de la spécification	6
3.2 Décrire les effets interdits	6
3.3 Trois niveaux de test	6
3.4 Lire un pointeur de fonction	6
3.5 Une doublure courte rend une panne reproductible	7
3.6 Exercice corrigé	7
4 Modéliser le domaine et contrôler les états	8
4.1 Le modèle de domaine partage un vocabulaire	8
4.2 Associations et multiplicités	8
4.3 Le processus éprouve le modèle	8
4.4 États, événements, gardes et actions	8
4.5 En C, une garde précède les effets	9
4.6 Exercice corrigé	9

1 Comprendre le besoin et vérifier une exigence

1.1 Demande, besoin et exigence

Une demande contient souvent une solution déjà choisie : « ajouter un bouton de rappel quotidien ». Le besoin décrit le problème à résoudre et la valeur recherchée : éviter l'attente des lecteurs et le suivi manuel des retards. L'exigence énonce une propriété que le système devra satisfaire : « un emprunt clos ne figure plus dans la liste des retards ».

Cette distinction permet de discuter plusieurs solutions sans perdre le problème initial. Elle évite aussi de confondre une demande très précise avec une compréhension complète du besoin. Une bonne question d'entretien recherche un fait : « racontez la dernière fois où un retour tardif a posé problème ». Une question fermée permet ensuite de confirmer une règle. Les deux formes ont une utilité, à des moments différents.

1.2 Les informations n'ont pas toutes le même statut

Un fait du dossier possède une source. Une hypothèse est une information à confirmer. Un objectif proposé doit être discuté avec le client. Un choix technique appartient à la conception, sauf si une contrainte externe l'impose. La présence d'un nombre ne transforme pas une hypothèse en fait.

Exemple raisonné

L'entretien révèle que des rappels sont préparés après le retour d'un livre. On en déduit un problème de cohérence du suivi. Une exigence possible est de retirer les emprunts clos de la liste des retards. Pour la vérifier, on prépare un emprunt en retard et un emprunt clos, puis on observe que seul le premier apparaît. On n'a pas encore choisi comment stocker les emprunts.

1.3 Exigences fonctionnelles et exigences de qualité

Une EF décrit un service ou un comportement attendu. Une ENF précise généralement une qualité ou une condition dans laquelle les services doivent fonctionner. « Enregistrer le retour d'un exemplaire » décrit un service. « Afficher la liste sur un volume de référence dans un délai fixé » précise une performance.

La frontière n'est pas toujours univoque. L'enjeu est d'expliciter ce que la phrase exige, son contexte et son moyen de vérification. Une contrainte technique telle que « utiliser une liste chaînée » se justifie séparément. Les familles de qualité, comme la fiabilité, la sécurité ou l'utilisabilité, aident à poser des questions oubliées. Elles ne remplacent pas les critères observables.

1.4 Formuler une exigence vérifiable

Une fiche courte contient un identifiant, la propriété attendue, sa source, le contexte, puis un protocole et un résultat attendu. Pour une performance, préciser les données, la machine, le début et la fin de la mesure, les répétitions et le seuil. Un test sur un petit jeu n'établit pas une promesse sur un volume beaucoup plus grand.

✓ À faire : Démarche en quatre étapes

Partir d'un fait. Formuler le problème sans imposer une solution. Écrire une propriété observable. Chercher un cas qui pourrait la mettre en défaut. On recherche ensuite le composant du programme responsable de cette propriété.

1.5 Lecture C : structure, en-tête et appel

Un en-tête partage des déclarations et des types. Le fichier .c porte les définitions. Le compilateur vérifie les unités séparément, puis l'éditeur de liens réunit les définitions et les appels. Inclure un

en-tête ne compile pas le fichier .c correspondant.

```
1 /* bilan.h : extraits de l'interface partagée */
2 typedef struct { int emprunts_examines; int retards; } bilan_t;
3 int aucun_retard(const bilan_t *b);
4 /* bilan.c : définition */
5 int aucun_retard(const bilan_t *b) { return b->retards == 0; }
6 /* dans main : objet, puis appel avec son adresse */
7 bilan_t bilan = { .emprunts_examines = 7, .retards = 2 };
8 int resultat = aucun_retard(&bilan);
```

bilan.retards lit le champ d'un objet. b->retards le lit à travers une adresse valide. &bilan produit cette adresse. Le contrat impose ici que b ne soit pas nul. Les gardes d'en-tête empêchent des déclarations répétées dans une même unité de compilation. Les exemples complets compilables figurent dans exemples-cm/.

1.6 Exercice corrigé

Classer et améliorer : « le logiciel est fiable », « enregistrer un retour », « stocker dans une liste chaînée », « un refus laisse l'échéance inchangée ».

La première formule reste invérifiable sans incident et garantie précis. La deuxième décrit un service dont il faut préciser le résultat. La troisième décrit une représentation technique. La quatrième énonce une conséquence testable : comparer l'échéance avant et après un refus. Plusieurs classements EF/ENF peuvent se discuter, mais le résultat attendu doit rester clair.

✓✓ Point d'arrêt — Pour vérifier votre compréhension

Pouvez-vous reformuler une demande en besoin ? Distinguer fait et hypothèse ? Proposer un test qui observe aussi une absence ? Retrouver déclaration, définition et appel d'une fonction, puis expliquer un champ lu à travers une adresse ?

2 Formaliser un scénario et un contrat

2.1 La story exprime l'utilité, ses critères précisent l'acceptation

Une story relie un acteur, une action et un bénéfice. « En tant que lecteur, je veux demander la prolongation de mon emprunt afin de terminer ma lecture sans déplacement inutile. » Elle amorce une conversation : combien de renouvellements, quelles réservations bloquantes, quelle durée supplémentaire ?

INVEST sert à relire cette formulation : indépendance, négociabilité, valeur, possibilité d'estimer, taille limitée et possibilité de tester. Il n'impose pas un paragraphe pour chaque lettre. Une tâche technique nécessaire peut rester une tâche technique, sans prendre artificiellement la forme d'une story.

Un critère d'acceptation décrit une situation, une action et une conséquence observable. Exemple : avec un emprunt déjà renouvelé deux fois, une nouvelle demande est refusée et l'échéance est conservée. Affirmer que « la fonction est appelée » ne suffit pas à démontrer ce service.

2.2 Le cas d'utilisation détaille un objectif complet

Le CU identifie l'acteur, le déclencheur, les préconditions et les résultats de succès et de refus. Son nominal numérote les interactions. Le lecteur sélectionne un emprunt, le système présente ses conditions, le lecteur demande le renouvellement, le système décide et enregistre, puis il confirme le résultat.

Le scénario reste lisible par l'acteur. La liste des appels système constitue une description de conception. Un scénario trop vague ne permet pas de discuter le comportement ; un scénario trop technique masque les décisions utiles.

2.3 Isoler une décision locale

Dans notre cas fictif, on autorise au plus deux renouvellements, sauf réservation par un autre lecteur. La fonction de décision reçoit un emprunt actif dont le compteur est cohérent. Elle indique si l'autorisation est possible sans modifier l'objet. Une autre fonction effectuera le renouvellement.

Réservation bloquante	Compteur actuel	Autoriser
Non	0 ou 1	Oui
Non	2	Non
Oui	0 ou 1	Non
Oui	2	Non

La table montre des classes de cas. On choisit ensuite des valeurs concrètes, en particulier les valeurs proches des seuils. Le traitement d'un compteur incohérent doit apparaître dans le contrat ou dans les obligations de l'appelant.

2.4 Le contrat C précise les adresses et les effets

```

1 typedef struct {
2     int nb_renouvellements;
3     int reservation_bloquante;
4 } emprunt_t;
5 int renouvelable(const emprunt_t *e) {
6     if (e == NULL) return 0;
7     return e->nb_renouvellements < 2
8         && !e->reservation_bloquante;
9 }
```

Ce contrat enrichit explicitement la règle : une adresse absente donne 0. La fonction protège l'accès aux champs par une garde. `const` interdit leur modification à travers `e` ; il ne fige pas magiquement

l'objet pour tous les autres accès. Un pointeur valide est une adresse d'objet existant et correctement initialisé, pas seulement une valeur différente de NULL.

Objet, copie et adresse

Une affectation `emprunt_t b = a` copie ici les deux champs entiers. Une affectation `const emprunt_t *p = &a` crée une adresse vers le même objet. Modifier `b` ne change pas `a`. Lire `p->champ` lit le champ de `a`. L'objet doit rester vivant pendant l'utilisation de son adresse.

2.5 Choisir un test discriminant

Une condition erronée qui examine seulement le compteur accepte le premier cas de la table. Le cas « compteur égal à un et réservation bloquante » distingue cette condition de la règle attendue. Il doit refuser. Tester un seul succès ne suffirait donc pas.

La méthode de modification est la suivante : partir du contrat, choisir un cas, prédire le résultat du programme actuel, exécuter, expliquer l'écart, corriger, puis rejouer les cas précédents. On ajoute un test personnel en indiquant précisément quelle erreur il permettrait de détecter.

2.6 Exercice corrigé

Deux expressions utilisent `compteur < 2` et `!reservation` : l'une les relie par `||`, l'autre par `&&`. Avec `compteur 2` et sans réservation, la première accepte et la seconde refuse. Le contrat exige un refus. Il faut satisfaire les deux conditions. Si l'adresse peut être absente, on doit en outre vérifier cette absence avant toute lecture de champ.

✓✓ Point d'arrêt — Pour vérifier votre compréhension

Distinguer story, critère, scénario et contrat. Construire une table qui fait varier une condition à la fois. Expliquer `const`, `NULL`, `&` et `->`. Proposer un cas qui réfute une condition incomplète.

3 Décrire les exceptions et les rendre testables

3.1 Les exceptions font partie de la spécification

Pour chaque étape du nominal, chercher une entrée absente, une règle non satisfaite, une opération impossible ou un résultat inconnu. Un flux alternatif précise son point de départ, ses actions, sa terminaison ou sa reprise, puis le résultat observable. « Afficher une erreur » ne précise pas l'état des données après l'échec.

Dans la bibliothèque, un refus dû à une réservation laisse l'échéance inchangée. Une panne d'envoi après un renouvellement enregistré pose une autre question. Le choix d'annuler ou de conserver le renouvellement appartient à la décision métier. Dans notre cas fictif, on conserve le renouvellement acquis et on rend l'échec de notification visible. Le code doit donc distinguer ces deux résultats.

3.2 Décrire les effets interdits

Une postcondition peut demander une absence : aucun changement d'échéance après refus, aucune confirmation de réussite après échec d'enregistrement, aucun second traitement d'un événement déjà appliqué. Les tests doivent observer ces absences, pas seulement une valeur de retour.

Un protocole complet

Préparer des données et un incident contrôlé. Déclencher une action. Vérifier le résultat, les changements permis et les effets interdits. La conséquence métier, le code retourné et le compteur d'une opération peuvent fournir des observations complémentaires.

3.3 Trois niveaux de test

Le test unitaire vérifie une règle locale avec ses entrées contrôlées. Le test d'intégration vérifie la coopération de composants, par exemple une écriture suivie d'une lecture dans un fichier temporaire. Le test d'acceptation vérifie un service décrit du point de vue de l'acteur.

La stratégie combine ces niveaux selon les risques. Une doublure n'établit pas le fonctionnement du composant réel qu'elle remplace. Un test avec fichier réel ne suffit pas non plus à isoler chaque règle. Le choix dépend de ce qu'on cherche à démontrer, et non du nom du fichier qui contient le test.

3.4 Lire un pointeur de fonction

```
1 int ajouter(int a, int b) { return a + b; }
2 int (*operation)(int, int) = ajouter;
3 /* Dans une fonction : operation(7, 3) renvoie 10. */
```

En partant du nom, `operation` est un pointeur vers une fonction de deux entiers renvoyant un entier. Les parenthèses sont essentielles : `int *operation(int, int)` déclarerait une fonction renvoyant un pointeur. Le pointeur doit être initialisé avec une fonction compatible avant l'appel.

On peut placer cette adresse dans une structure et fournir en plus une adresse de données à l'implémentation :

```
1 typedef struct {
2     void *ctx;
3     int (*envoyer)(void *, const char *);
4 } notification_t;
5 int avertir(const notification_t *n) {
6     return n->envoyer(n->ctx, "Nouvelle echance");
7 }
```

Le service appelle le contrat. Le montage choisit l'implémentation. `ctx` contient l'adresse d'un état propre à celle-ci. L'implémentation doit recevoir le type réel attendu et l'objet doit rester vivant. Le compilateur ne contrôle pas entièrement cet accord lorsqu'on utilise `void *`.

3.5 Une doublure courte rend une panne reproductible

```
1 typedef struct { int echec; int appels; } simulation_t;
2 int envoyer_simule(void *ctx, const char *message) {
3     simulation_t *s = ctx;
4     (void)message;
5     s->appels++;
6     return s->echec ? -1 : 0;
7 }
8 /* Dans le test : */
9 simulation_t s = { .echec = 1, .appels = 0 };
10 notification_t n = { .ctx = &s, .envoyer = envoyer_simule };
```

Une réponse programmée joue le rôle d'un stub. Un enregistrement des appels permet de vérifier les interactions; une doublure munie d'attentes peut jouer le rôle d'un mock. Un fake est une implémentation simplifiée mais fonctionnelle. Notre petite simulation ne reproduit pas un serveur : elle contrôle une réponse et compte les appels. Le test réalise les assertions.

Isoler une opération derrière ce contrat permet de choisir la panne sans changer le métier. C'est une application de l'inversion des dépendances. La contrepartie est un chemin d'appel indirect qu'il faut apprendre à lire.

3.6 Exercice corrigé

Avec `s.echec = 1`, un appel à `avertir(&n)` renvoie -1 et incrémente `appels`. On met ensuite `s.echec = 0` et on rappelle la même fonction : retour 0 et compteur 2. Le champ fonction n'a pas changé. Modifier l'état de la simulation et remplacer l'adresse de fonction sont deux manipulations différentes.

✓✓ Point d'arrêt — Pour vérifier votre compréhension

Rédiger un flux alternatif et sa garantie d'échec. Distinguer test local et intégration. Suivre un appel indirect jusqu'à la bonne définition, puis retrouver le type et la durée de vie du contexte. Choisir une assertion qui observe un effet interdit.

4 Modéliser le domaine et contrôler les états

4.1 Le modèle de domaine partage un vocabulaire

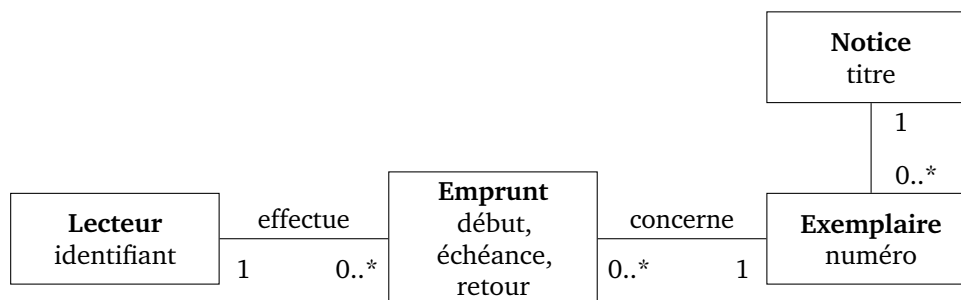
Le domaine contient les concepts nécessaires aux règles métier, leurs attributs et leurs relations. La conception choisit les tableaux, les fonctions, les fichiers ou les autres moyens de les représenter. Ces deux niveaux sont liés, mais on ne les confond pas.

Dans la bibliothèque, « livre » peut désigner une notice du catalogue ou l'exemplaire physique. Le glossaire fixe les sens. Pour chaque candidat extrait des scénarios, on décide : concept utile, attribut d'un concept, synonyme à fusionner ou élément à exclure. Une adresse peut être un simple attribut dans un périmètre, et devenir un concept dans un autre.

4.2 Associations et multiplicités

Un lecteur peut effectuer zéro à plusieurs emprunts. Chaque emprunt concerne exactement un lecteur. On place donc 0..* près d'Emprunt et 1 près de Lecteur. La multiplicité placée près d'une classe se lit en partant d'un objet de l'autre classe.

Un emprunt concerne un exemplaire ; un exemplaire peut avoir plusieurs emprunts successifs. Cela n'autorise pas nécessairement plusieurs emprunts actifs en même temps. Cette dernière contrainte doit être écrite comme un invariant.



La date de début décrit une occurrence de l'emprunt, et non le lecteur ou l'exemplaire isolément. On fait donc d'Emprunt un concept avec une identité. Une classe-association sert à porter des informations sur une association. Pour un historique où une même paire lecteur/exemplaire revient, il faut représenter clairement les occurrences distinctes.

Héritage : une spécialisation à justifier

Un livre peut être une spécialisation de document, avec les propriétés communes de document et des particularités utiles au périmètre. Une ressemblance de noms ne suffit pas. Un modèle n'est pas meilleur parce qu'il contient un héritage ; il doit rendre les règles discutables avec le métier. Le choix ne prescrit pas automatiquement un mécanisme C.

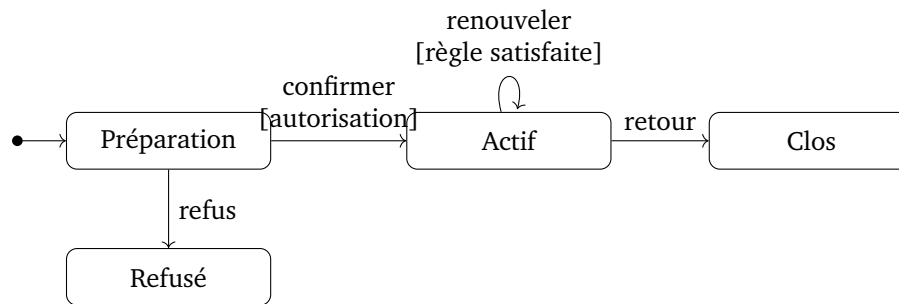
4.3 Le processus éprouve le modèle

Dans le processus d'emprunt, le personnel identifie lecteur et exemplaire, vérifie l'autorisation, confirme la remise, enregistre l'échéance et informe le lecteur. À chaque étape, on demande quelles informations le modèle doit représenter. Une autorisation à vérifier peut révéler une règle ou une donnée oubliée.

Le processus décrit une activité qui peut impliquer plusieurs objets. L'automate décrit la vie d'un objet donné. Cette distinction évite de faire d'une succession d'écrans l'automate d'un objet métier.

4.4 États, événements, gardes et actions

Une tentative d'emprunt commence en Préparation. Une confirmation autorisée la rend Active ; un refus la rend Refusée. Un emprunt actif peut être renouvelé si les règles le permettent, en restant actif. Son retour le rend Clos. Clos et Refusé sont terminaux dans ce périmètre.



L'événement indique ce qui arrive. La garde est la condition qui autorise la transition. L'action réalise les modifications. Une transition interdite doit produire un résultat défini et préserver les données concernées.

✓ À faire : Deux invariants utilisables

Un emprunt clos ne redevient pas actif par un second retour. Un renouvellement refusé ne modifie ni échéance ni compteur. Ces phrases permettent de choisir des états initiaux, des événements et des observations pour les tests.

4.5 En C, une garde précède les effets

```

1 typedef enum { PREPARATION, ACTIF, CLOS, REFUSE } etat_t;
2 typedef struct { etat_t etat; int retours; } emprunt_t;
3 int clore(emprunt_t *e) {
4     if (e == NULL) return -1;
5     switch (e->etat) {
6     case ACTIF:
7         e->etat = CLOS;
8         e->retours++;
9         return 0;
10    default:
11        return -1;
12    }
13 }

```

Le paramètre ne porte pas `const` : le contrat autorise une modification. Un appel sur un emprunt actif réussit. Un second appel refuse et laisse le compteur à un. Le test doit vérifier le retour et l'état conservé, sans se limiter à l'absence de plantage.

Si la transition dépend d'une opération qui peut échouer, l'ordre doit respecter le contrat. On n'annonce pas l'action comme réalisée avant d'avoir constaté son succès. Les garanties après interruption demandent en plus de préciser ce qui persiste hors de la mémoire du programme.

4.6 Exercice corrigé

La multiplicité « plusieurs emprunts par exemplaire » décrit l'historique. Elle ne prouve pas que deux emprunts actifs peuvent coexister. Un programme qui affecterait `CLOS` avant de regarder l'état initial transformerait à tort une demande refusée. Le test choisit `REFUSE`, appelle `clore`, attend -1 et vérifie que l'état est resté `REFUSE`.

✓✓ Point d'arrêt — Pour vérifier votre compréhension

Lire une multiplicité dans les deux sens. Distinguer domaine et conception, processus et automate. Formuler un invariant et une transition interdite. Retrouver la garde et l'effet dans le code. Concevoir un test qui contrôle aussi l'absence de modification après refus.