Tests logiciels — Cours appliqué à `hash_tool`

> Ce cours couvre les notions abordées dans l'analyse de la suite de tests de `hash_tool` : tests unitaires, d'intégration, de non-régression, CI/CD, rapports de tests. Chaque notion est expliquée puis illustrée avec des exemples tirés du projet.

Table des matières

---

1. Pourquoi tester ?

Un test est une vérification automatisée qu'un comportement attendu est bien produit par le code. Sans tests :

- Une modification dans `core.sh` peut casser `compare` sans que personne s'en aperçoive avant qu'un utilisateur perde des données.
- Impossible de refactoriser avec confiance — chaque changement est un pari.
- Le debugging est lent : il faut reproduire le problème manuellement à chaque fois.

Avec des tests :

- Un test qui échoue localise immédiatement la régression.
- Le code peut être modifié, optimisé, réorganisé — les tests garantissent que le comportement observable reste stable.
- La documentation implicite : un test qui s'appelle `test_compare_chemins_avec_espaces` documente un comportement et une contrainte du système.

Dans hash_tool

La suite `run_tests.sh` permet de valider en quelques secondes que `integrity.sh` fonctionne correctement sur 14 scénarios. Sans elle, valider manuellement chaque cas (compute, verify OK, verify après

corruption, compare avec ajout/suppression, mode quiet…) prendrait 30 minutes et serait oublié à la prochaine modification.

---

2. La pyramide des tests

La pyramide des tests est un modèle qui décrit comment répartir les efforts de test selon le niveau d'abstraction.

```
          /\
         /  \
        / E2E\        Tests de bout en bout (End-to-End)
       /______\       Rares, lents, fragiles
      /        \
     / Intégration\   Tests d'intégration
    /______________\  Moyennement nombreux
   /                \
  /   Unitaires      \ Tests unitaires
 /____________________\ Nombreux, rapides, précis
```

Niveau	Quoi	Vitesse	Fragilité	Nb recommandé
Unitaire	Une fonction isolée	Milliseconde	Faible	Maximum
Intégration	Plusieurs modules ensemble	Secondes	Moyenne	Moyen
E2E	Système complet, interface réelle	Minutes	Élevée	Minimum

Dans hash_tool

Actuellement, `run_tests.sh` et `run_tests_pipeline.sh` sont **presque exclusivement des tests d'intégration** — ils testent `integrity.sh` et `runner.sh` comme boîtes noires, sans tester les fonctions de `core.sh` individuellement. La pyramide est inversée : le bas (unitaire) est vide, le milieu (intégration) est bien couvert.

---

3. Tests unitaires

Un test unitaire vérifie une seule unité de code — généralement une fonction — en isolation complète. Il ne dépend d'aucun autre module, réseau, système de fichiers (sauf si la fonction elle-même écrit des fichiers).

Caractéristiques

- **Rapide** : pas d'I/O disque, pas de processus externes.
- **Précis** : si un test unitaire échoue, le bug est dans cette fonction, nulle part ailleurs.
- **Indépendant** : l'ordre d'exécution n'a pas d'importance.

Exemple concret — tester `core_assert_b3_valid`

```
# Sans tests unitaires : on teste via integrity.sh
./src/integrity.sh verify fichier_corrompu.b3
# → si ça échoue, est-ce core_assert_b3_valid ? core_verify ? ui.sh ?

# Avec tests unitaires : on source core.sh directement
QUIET=0
source ./src/lib/ui.sh
source ./src/lib/core.sh

test_b3_valide_fichier_absent() {
    core_assert_b3_valid "/tmp/inexistant_xyz.b3" 2>/dev/null
    # Doit retourner exit code 1
    [ $? -ne 0 ] && echo "PASS" || echo "FAIL"
}

test_b3_valide_format_invalide() {
    echo "cette ligne n'est pas un hash b3sum" > /tmp/bad.b3
    core_assert_b3_valid /tmp/bad.b3 2>/dev/null
    [ $? -ne 0 ] && echo "PASS - format invalide rejeté" || echo "FAIL"
    rm -f /tmp/bad.b3
}
```

La différence clé : on **source** `core.sh` au lieu d'appeler `integrity.sh`. Les fonctions deviennent directement accessibles dans le shell courant.

Pourquoi c'est important ici

`core.sh` contient l'algorithme central : `core_compare` utilise `awk`, `join`, `comm`. Un bug dans la conversion `hash chemin` → `chemin\thash` produit des faux positifs massifs (comme le bug décrit dans le changelog 0.7 : 26 569 "modifiés" pour 1 seul fichier réellement changé). Un test unitaire sur `core_compare` aurait détecté ça immédiatement.

---

4. Tests d'intégration

Un test d'intégration vérifie que plusieurs modules fonctionnent correctement **ensemble**. Il teste les interfaces entre les composants.

Ce qu'il détecte

- Un module A produit un format que le module B ne sait pas lire.
- Une variable d'environnement attendue par B n'est pas positionnée par A.
- Un `cd` dans A modifie le répertoire courant de B (c'est exactement le bug isolé dans `runner.sh` : les `cd` fuyaient entre les blocs du pipeline).

Exemple — tester l'intégration `core.sh` + `results.sh`

```
# Test d'intégration : core_verify produit des variables
# que results_write_verify sait exploiter
```

```
source ./src/lib/ui.sh
source ./src/lib/core.sh
source ./src/lib/results.sh

OUTDIR=$(mktemp -d)
echo "contenu" > /tmp/fichier_test.txt
./src/integrity.sh compute /tmp/fichier_test.txt /tmp/base_test.b3

# core_verify positionne CORE_VERIFY_STATUS, NB_OK, etc.
core_verify /tmp/base_test.b3

# results_write_verify doit savoir lire ces variables
results_write_verify "$OUTDIR" /tmp/base_test.b3 \
    "$CORE_VERIFY_STATUS" "$CORE_VERIFY_NB_OK" "$CORE_VERIFY_NB_FAIL" \
    "$CORE_VERIFY_LINES_FAIL" "$CORE_VERIFY_LINES_ERR"

# Vérifier que recap.txt est produit avec le bon contenu
grep -q "STATUT : OK" "$OUTDIR/recap.txt" && echo "PASS" || echo "FAIL"
```

Dans hash_tool

`run_tests.sh` est une suite d'intégration : elle appelle `integrity.sh` comme le ferait un utilisateur, et vérifie les fichiers produits et les messages affichés. C'est utile et nécessaire, mais insuffisant seul — un test d'intégration qui échoue ne dit pas *où* est le problème.

---

5. Tests de non-régression

Un test de non-régression (TNR) vérifie qu'une fonctionnalité qui marchait avant **marche toujours** après une modification. Il capture un comportement connu et le fige.

Principe

1. À un instant T, le système produit un output correct → on le capture comme référence.
2. À chaque modification ultérieure, on vérifie que l'output est toujours identique à la référence.
3. Si l'output change → soit c'est un bug (la modification a cassé quelque chose), soit c'est intentionnel (il faut alors mettre à jour la référence).

Exemple — non-régression du format `.b3`

```
# Étape 1 : créer la fixture (fait une seule fois, commitée dans le repo)
mkdir -p tests/fixtures/data
echo "contenu alpha" > tests/fixtures/data/alpha.txt
echo "contenu beta"  > tests/fixtures/data/beta.txt

cd tests/fixtures
../../src/integrity.sh compute ./data reference.b3
# reference.b3 est commitée dans git
```

```
# Étape 2 : test de non-régression (lancé à chaque PR)
test_format_b3_stable() {
    cd tests/fixtures
    ../../src/integrity.sh compute ./data /tmp/output_test.b3
    diff reference.b3 /tmp/output_test.b3
    [ $? -eq 0 ] && echo "PASS - format .b3 stable" || echo "FAIL -
régression détectée"
}
```

Si quelqu'un modifie `core_compute` et introduit accidentellement un espace en trop dans le séparateur, ou change l'ordre de tri — le `diff` échoue immédiatement.

Ce que le TNR détecte que les autres tests ne détectent pas

Un TNR détecte des **changements subtils de comportement** qui ne cassent pas les tests fonctionnels mais altèrent le format ou le résultat. Par exemple :

- Un `b3sum` mis à jour qui change le format de sortie.
- Un `sort` qui change de comportement selon la locale (`LC_ALL`).
- L'ajout d'une ligne vide dans le `.b3` par une nouvelle version de `find`.

---

6. Tests de cas limites (edge cases)

Un cas limite est une entrée qui se situe aux frontières du comportement normal — là où les bugs se cachent le plus souvent.

Catégories de cas limites

Valeurs vides ou nulles

```
# Que se passe-t-il avec un dossier vide ?
mkdir /tmp/dossier_vide
./src/integrity.sh compute /tmp/dossier_vide /tmp/vide.b3
# core_assert_target_valid doit lever une erreur
```

Caractères spéciaux dans les noms

Les noms de fichiers peuvent contenir des espaces, des newlines, des caractères HTML. Chacun peut casser un traitement textuel naïf.

```
# Espace dans le nom
echo "contenu" > "tests/data/fichier avec espace.txt"
# Apostrophe
echo "contenu" > "tests/data/l'important.txt"
# Caractère HTML — dangereux dans report.html
echo "contenu" > "tests/data/<script>alert.txt"
```

```
# Newline dans le nom (cas extrême mais légal sur Linux)
echo "contenu" > $'tests/data/nom\navec\nnewline.txt'
```

Taille extrême

```
# Fichier de taille zéro
touch tests/data/fichier_vide.bin
# → core_compute : bytes_done += 0, ETA correcte ?
```

Cas limites de comparaison

```
# Tous les fichiers identiques → modifies.b3 vide
# Tous les fichiers supprimés → disparus.txt contient tout
# Un seul fichier dans la base → comportement sur base minimale
```

Pourquoi les cas limites cassent souvent le code

Le code est typiquement écrit et testé sur des cas "normaux". Les cas limites exposent des hypothèses implicites :

- `awk '{print $2}'` — hypothèse : le chemin ne contient pas d'espace. Faux.
- `grep -c '.'` — hypothèse : retourne 0 sur flux vide. Faux, `grep` retourne exit code 1 sur flux vide, ce qui crash un script avec `set -e`. (C'est le bug corrigé en v0.6.)
- `sort -k2` — hypothèse : le tri sur le champ 2 s'arrête au champ 2. Faux, `sort -k2` trie du champ 2 jusqu'à la fin de ligne. (Bug corrigé en v0.6, remplacé par `sort -k2,2`.)

---

7. ShellCheck — analyse statique

L'analyse statique examine le code **sans l'exécuter** pour détecter des erreurs potentielles. Pour bash, l'outil de référence est **ShellCheck**.

Ce que ShellCheck détecte

```
# Variable non quotée → éclate sur les espaces
for f in $(find . -type f); do     # ← SC2044 : use find -exec or while
read
    echo $f                        # ← SC2086 : double quote to prevent
globbing
done

# Comparaison de chaînes avec [ ] au lieu de [[ ]]
if [ $VAR == "valeur" ]; then      # ← SC2039 : use [[ ]] in bash

# Variable utilisée avant d'être définie
echo $UNDEFINED_VAR                # ← SC2154 : variable referenced but
```

```
not assigned

# Pipe dans un sous-shell
cat file | read var                # ← SC2031 : var will be in a subshell
```

Dans hash_tool

Le test T00 dans `run_tests.sh` lance ShellCheck sur `integrity.sh` :

```
# T00 - ShellCheck
if command -v shellcheck &>/dev/null; then
    assert_exit_zero "ShellCheck integrity.sh" shellcheck "$INTEGRITY"
else
    echo "  SKIP - shellcheck non installé"
fi
```

ShellCheck est la première ligne de défense — il détecte des bugs sans même exécuter les scripts. Zéro warning ShellCheck est une condition non négociable avant toute PR.

---

8. Le protocole TAP

TAP (Test Anything Protocol) est un format texte standard pour les résultats de tests. Il est lisible par les humains et parseable par les outils CI.

Format

```
TAP version 14
1..5
ok 1 - core_assert_b3_valid : fichier absent → exit 1
ok 2 - core_assert_b3_valid : format invalide → exit 1
not ok 3 - core_compare : chemins avec espaces
# Expected: beta.txt in modifies.b3
# Got: (empty)
ok 4 - core_make_result_dir : horodatage anti-écrasement
ok 5 - mode --quiet : stdout vide
```

Structure

- `1..N` : nombre total de tests annoncé en tête.
- `ok N - description` : test réussi.
- `not ok N - description` : test échoué.
- `# commentaire` : diagnostic supplémentaire (indentation sous un `not ok`).

Implémentation dans bash

```bash
#!/usr/bin/env bash
TOTAL=0; PASS=0; FAIL=0

plan() { echo "1..$1"; }
ok()     { TOTAL=$((TOTAL+1)); PASS=$((PASS+1)); echo "ok $TOTAL - $1"; }
not_ok() { TOTAL=$((TOTAL+1)); FAIL=$((FAIL+1)); echo "not ok $TOTAL - $1";
}

plan 3

# Test 1
if core_assert_b3_valid /tmp/inexistant 2>/dev/null; then
    not_ok "fichier absent doit échouer"
else
    ok "fichier absent → exit 1"
fi

# Test 2
echo "aa  ./fichier.txt" > /tmp/valid.b3  # hash trop court
if core_assert_b3_valid /tmp/valid.b3 2>/dev/null; then
    not_ok "hash invalide doit échouer"
else
    ok "hash invalide → exit 1"
fi
```

Avantage

Le format TAP est **interopérable** : GitHub Actions, GitLab CI, Jenkins, et des dizaines d'outils savent parser TAP et afficher des rapports visuels sans configuration supplémentaire.

---

9. CI/CD — Intégration et déploiement continus

Définitions

CI (Continuous Integration) : à chaque push ou pull request, un serveur exécute automatiquement les tests. Si un test échoue, la modification est bloquée ou signalée.

CD (Continuous Deployment) : si les tests passent, le code est automatiquement déployé en production.

Pour un outil comme hash_tool, le CD n'est pas pertinent (pas de service web à déployer). La CI, en revanche, est essentielle.

Pourquoi la CI est indispensable

Sans CI : les tests ne sont lancés que si le développeur y pense. En pratique, ils ne sont lancés qu'avant les releases, et pas systématiquement.

Avec CI : chaque modification est testée automatiquement, dans un environnement propre (pas les dépendances locales du développeur), sur les versions exactes des outils installés.

GitHub Actions — anatomie d'un workflow

```
# .github/workflows/ci.yml
name: CI

on:
  push:              # déclenché à chaque push
  pull_request:      # déclenché à chaque PR

jobs:
  tests:
    runs-on: ubuntu-latest  # environnement propre, recréé à chaque run

    steps:
      # 1. Récupérer le code
      - uses: actions/checkout@v4

      # 2. Installer les dépendances
      - run: sudo apt-get install -y b3sum jq shellcheck

      # 3. Lancer les suites de tests
      - run: cd tests && ./run_tests.sh
      - run: cd tests && ./run_tests_pipeline.sh

      # 4. ShellCheck
      - run: shellcheck src/integrity.sh runner.sh src/lib/*.sh

      # 5. Uploader les artefacts (résultats) même si les tests échouent
      - uses: actions/upload-artifact@v4
        if: always()   # ← important : s'exécute même si les steps
précédents échouent
        with:
          name: test-results
          path: /tmp/integrity-test*/
          retention-days: 7
```

Matrice de tests

Un test peut passer sur Ubuntu 22.04 et échouer sur 24.04 si la version de `b3sum` ou de `awk` a changé de comportement. La matrice permet de tester plusieurs environnements en parallèle :

```
jobs:
  tests:
    strategy:
      matrix:
        os: [ubuntu-22.04, ubuntu-24.04]
        bash: ["5.1", "5.2"]

    runs-on: ${{ matrix.os }}
    steps:
```

```
      - run: bash --version
      - run: cd tests && ./run_tests.sh
```

Test Docker dans la CI

```
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker build -t hash_tool .
      - run: docker run --rm hash_tool version
      - name: Test compute dans Docker
        run: |
          mkdir -p /tmp/testdata /tmp/testbases
          echo "contenu" > /tmp/testdata/fichier.txt
          docker run --rm \
            -v /tmp/testdata:/data:ro \
            -v /tmp/testbases:/bases \
            hash_tool compute /data /bases/test.b3
          test -f /tmp/testbases/test.b3 && echo "PASS" || echo "FAIL"
```

---

10. Isolation et reproductibilité

Un test doit être **isolé** : son résultat ne dépend pas des autres tests, de l'état du système, ni de l'ordre d'exécution. Il doit être **reproductible** : relancé 10 fois dans 10 environnements différents, il donne le même résultat.

Les ennemis de l'isolation

État partagé : une variable globale modifiée par un test affecte le suivant.

```
# ✗ Mauvais : RESULTATS_DIR partagé entre les tests
export RESULTATS_DIR=/tmp/resultats_partages
test_verify_ok() { ... }
test_verify_echec() { ... }   # peut lire les résultats du test précédent
```

```
# ✓ Correct : chaque test a son propre WORKDIR
test_verify_ok() {
    local WORKDIR=$(mktemp -d)
    export RESULTATS_DIR="$WORKDIR/resultats"
    # ... test ...
    rm -rf "$WORKDIR"   # nettoyage garanti
}
```

Répertoire courant : un test qui fait `cd` et ne revient pas casse le suivant.

```
# ✖ Mauvais
test_compute() {
    cd /tmp/montest
    ../integrity.sh compute . base.b3
    # Si le test échoue ici, le cd ne revient jamais
}

# ✓ Correct : sous-shell isolé
test_compute() {
    (
        cd /tmp/montest
        ../integrity.sh compute . base.b3
    )   # le cd est confiné dans le sous-shell
}
```

Fichiers temporaires : un test qui échoue à mi-chemin laisse des fichiers qui perturbent le suivant.

```
# ✓ Correct : trap pour nettoyage garanti même en cas d'échec
test_compute() {
    local tmpdir=$(mktemp -d)
    trap "rm -rf $tmpdir" EXIT   # nettoyage garanti
    # ... test ...
}
```

Dans hash_tool

`run_tests.sh` utilise un `WORKDIR=$(mktemp -d /tmp/integrity-test.XXXXXX)` créé en `setup()` et détruit en `teardown()`. Tous les tests opèrent dans ce répertoire temporaire, jamais dans les fichiers du projet ou du système hôte.

---

11. Couverture de tests

La couverture (coverage) mesure quelle proportion du code est exercée par les tests.

Types de couverture

Couverture de lignes : chaque ligne est-elle exécutée au moins une fois ?

Couverture de branches : chaque branche d'un `if`/`case` est-elle testée (chemin vrai ET chemin faux) ?

```
# Cette fonction a 2 branches
if (( fsize > 0 )); then
    bytes_done=$(( bytes_done + fsize ))
    # ← branche testée si fsize > 0
fi
```

```
# ← branche testée si fsize == 0 (fichier vide)

# Un test avec uniquement des fichiers non-vides → couverture 50% de cette
condition
```

Couverture de chemins : chaque combinaison possible de branches est-elle testée ? (Combinatoire explosive, rare en pratique.)

Lacunes de couverture dans hash_tool

En analysant le code, voici des branches **non testées** :

```
# Dans _core_file_size() — branche BSD
_core_file_size() {
    if stat -c%s "$f" 2>/dev/null; then   # ← testé sur Linux
        return
    fi
    stat -f%z "$f"   # ← jamais testé (macOS uniquement)
}

# Dans core_compute — callback vide
core_compute "$target" "$hashfile" ""   # ← jamais testé sans callback

# Dans ui_progress_callback — cas bytes_done == 0
# La branche (bytes_done > 0 && eta_seconds > 0) est testée
# Mais (bytes_done == 0) — premier fichier, juste après démarrage ?
```

Comment mesurer la couverture en bash

Il n'existe pas d'outil de couverture natif pour bash équivalent à `coverage.py`. La méthode pragmatique est manuelle : relire chaque branche du code et vérifier qu'un test l'exerce.

Pour les projets bash critiques, `bashcov` (basé sur `xtrace`) ou simplement `set -x` avec analyse de log permettent de voir quelles lignes sont exécutées.

---

12. Fixtures et données de test

Une fixture est un ensemble de données préparées à l'avance, dans un état connu, utilisées comme entrée des tests.

Types de fixtures

Données dynamiques : créées dans le `setup()` du test, détruites dans le `teardown()`.

```
setup() {
    mkdir -p "$WORKDIR/data/sub"
    echo "contenu alpha" > "$WORKDIR/data/alpha.txt"
```

```
    echo "contenu beta"  > "$WORKDIR/data/beta.txt"
}
```

C'est l'approche de `run_tests.sh` : les fichiers de test sont créés à chaque run. Avantage : pas de fichiers à maintenir dans le repo. Inconvénient : si la fixture est complexe (arborescence de 500 fichiers avec des noms spéciaux), le setup devient lui-même un code à maintenir et à tester.

Fixtures statiques (commitées dans git) : fichiers présents dans le repo, utilisés comme référence immuable.

```
tests/
└── fixtures/
    ├── data/
    │   ├── alpha.txt
    │   └── beta.txt
    └── reference.b3   ← résultat attendu, commité dans git
```

```
test_format_b3_stable() {
    cd tests/fixtures
    ../../src/integrity.sh compute ./data /tmp/output.b3 >/dev/null 2>&1
    diff reference.b3 /tmp/output.b3
    [ $? -eq 0 ] && pass "format stable" || fail "régression format"
}
```

Si `reference.b3` change dans un PR, c'est visible dans le diff git — c'est un signal fort qu'il faut examiner.

Fixtures pour les cas limites

Certains cas limites sont difficiles à créer dynamiquement de manière fiable. Les fixtures statiques les capturent une fois pour toutes :

```
tests/fixtures/
├── edge_cases/
│   ├── fichier avec espaces.txt
│   ├── fichier&special<chars>.txt
│   └── .fichier_cache
└── reference_edge.b3
```

---

13. Synthèse — Appliquer tout ça à hash_tool

Ce qui existe et fonctionne bien

`run_tests.sh` (T00-T14) et `run_tests_pipeline.sh` (TP01-TP12b) forment une suite d'intégration solide. L'isolation via `mktemp`, le `teardown()` systématique, les helpers `pass()`/`fail()` — c'est une base

professionnelle.

Ce qui manque, par ordre de priorité

1. CI GitHub Actions — sans CI, les tests ne sont lancés que si on y pense. Action : créer `.github/workflows/ci.yml`.

2. Tests unitaires de `core.sh` — créer `tests/run_tests_core.sh` qui source `core.sh` directement et teste chaque fonction en isolation. Priorité : `core_compare` (algorithme complexe, bug historique).

3. Fixture de non-régression du format `.b3` — commiter `tests/fixtures/reference.b3` et ajouter un test qui vérifie que `core_compute` produit exactement ce fichier.

4. Cas limites manquants :

- Fichier avec newline dans le nom (T15)
- Caractères HTML dans le nom de fichier → vérifier l'échappement dans `report.html` (T16)
- `--quiet` sur `compare` (T18)
- Fichier de taille zéro — comportement ETA (T19)

5. Rapport TAP — modifier les suites pour produire un output TAP, consommable par la CI sans configuration supplémentaire.

Architecture cible de la suite de tests

```
tests/
├── run_tests.sh              ← intégration integrity.sh (T00-T20+)
├── run_tests_pipeline.sh     ← intégration runner.sh (TP01-TP12b+)
├── run_tests_core.sh         ← unitaires core.sh (nouveau)
├── run_tests_docker.sh       ← intégration entrypoint.sh (nouveau)
└── fixtures/
    ├── data/
    │   ├── alpha.txt
    │   ├── beta.txt
    │   ├── fichier avec espaces.txt
    │   └── <script>edge.txt
    └── reference.b3           ← non-régression format .b3
```

Règle d'or

Un test n'a de valeur que s'il est **lancé automatiquement**. Un test dans un fichier que personne ne lance est une illusion de couverture. La CI est le seul mécanisme qui garantit que les tests sont effectivement exécutés à chaque modification.

Fin du cours — toutes les notions sont illustrées avec des exemples tirés du code réel de `hash_tool`.