Guide complet de correction des erreurs pour TypeScript

Ce guide compile les erreurs courantes rencontrées dans les projets TypeScript comme MetaSign, ainsi que leurs solutions. Il intègre des connaissances issues de notre expérience de correction d'erreurs et des meilleures pratiques actuelles.

Types d'erreurs rencontrées

1. Erreurs liées à l'accès des propriétés privées dans les tests

Problème : Tentative d'accès à des propriétés privées d'une classe depuis des tests.

```typescript
// Erreur
interface MockRegistry extends ProformeRegistry {
    activeProformeIds: Set<string>; // Erreur car activeProformeIds est privé
}
```

Solution : Créer une classe de test dédiée avec des méthodes protégées pour exposer les fonctionnalités nécessaires aux tests.

```typescript
// Solution
class TestProformeRegistry extends ProformeRegistry {
  public setTestProformes(proformes: Proforme[]): void {
    // Utilise des méthodes protégées au lieu d'accéder directement aux propriétés privées
    this.reset();
    proformes.forEach(proforme => {
      this.addProforme(proforme);
      this.activateProforme(proforme.id);
    });
  }
}
```

2. Problèmes de typage des structures de données

Problème : Incompatibilité entre la définition d'un type et son utilisation.

```typescript
// Erreur - Type défini comme string mais utilisé comme objet
export interface Proforme {
  handshape: string; // Défini comme string
}

// Usage comme objet complexe
const proforme: Proforme = {
  handshape: {
    type: 'index-pointing',
    fingers: [...],
    tension: 0.7
  }
};
```

Solution : Mettre à jour les définitions de types pour qu'elles correspondent à l'utilisation réelle.

```typescript
// Solution
export interface HandshapeConfig {
  type: string;
  fingers: FingerConfig[];
  tension: number;
}

export interface Proforme {
  handshape: HandshapeConfig; // Type mis à jour
}
```

3. Propriétés inexistantes dans les interfaces

Problème : Utilisation de propriétés qui n'existent pas dans les définitions des types.

```typescript
// Erreur - Position n'existe pas dans le type Proforme
proforme.position = this.adjustPositionForFormality(proforme.position, formalityLevel);
```

Solution : Ajouter les propriétés manquantes aux interfaces.

```typescript
// Solution
export interface Proforme {
  // Propriétés existantes
  id: string;
  name?: string;
  handshape: HandshapeConfig;

  // Ajout des propriétés manquantes
  position?: Point3D;
  represents: string;
}
```

4. Valeurs invalides pour les enums et types restreints

Problème : Utilisation de valeurs qui ne sont pas dans l'ensemble des valeurs autorisées.

```typescript
// Erreur - "standard" n'est pas une valeur valide pour context
const defaultContext: CulturalContext = {
  context: 'standard' // Erreur
};
```

Solution : Utiliser des valeurs valides selon la définition du type.

```typescript
// Solution
const defaultContext: CulturalContext = {
  context: 'conversational' // Valeur valide
};
```

5. Importations et variables non utilisées

Problème : Importation de types ou déclaration de variables qui ne sont pas utilisés.

```typescript
// Erreur
import { Vector3D } from '../types'; // Importation non utilisée
```

Solution : Supprimer les importations non utilisées ou les variables inutilisées.

```typescript
// Solution - Supprimer l'importation ou l'utiliser
import { Point3D, FingerConfig } from '../types'; // Uniquement ce qui est utilisé
```

6. Types implicites "any"

Problème : Utilisation de paramètres sans type explicite, ce qui leur donne implicitement le type `any`.

```typescript
// Erreur
const hasFranceProforme = activeProformes.some(p => p.id === 'region-france-vehicle');
```

Solution : Spécifier explicitement le type des paramètres.

```typescript
// Solution
const hasFranceProforme = activeProformes.some((proforme: Proforme) => proforme.id === 'region-
```

7. Méthodes non utilisées mais référencées

Problème : Des méthodes qui sont définies mais jamais appelées.

```typescript
// Erreur
private adjustOrientationForFormality(orientation: Vector3D, formalityLevel: number): Vector3D
  // Méthode non utilisée
}
```

Solution : Supprimer les méthodes non utilisées ou ajouter des commentaires expliquant pourquoi elles sont conservées.

```typescript
// Solution 1: Supprimer la méthode si elle n'est pas nécessaire

// Solution 2: Commenter pourquoi elle est conservée
/**
 * Cette méthode n'est pas utilisée actuellement mais est conservée pour
 * une utilisation future lorsque le type d'orientation sera mis à jour.
 */
```

8. Erreurs d'assignation avec exactOptionalPropertyTypes

Problème : Avec `exactOptionalPropertyTypes: true` dans tsconfig, on ne peut pas assigner directement `undefined` à une propriété qui n'inclut pas explicitement `undefined` dans son type.

Solution : Définir explicitement les propriétés optionnelles pour inclure `undefined` ou utiliser un opérateur d'union.

```typescript
// Solution
interface User {
  name: string;
  email?: string | undefined; // Explicitement optionnel
}
```

9. Erreurs de déstructuration d'objets

Problème : La déstructuration d'objets peut causer des erreurs si les propriétés n'existent pas.

Solution : Utiliser des valeurs par défaut ou vérifier l'existence des propriétés.

```typescript
// Solution
const { name, age = 0 } = user || {}; // Valeur par défaut + vérification de null/undefined
```

Meilleures pratiques de tests

1. Tests unitaires avec une classe dédiée

Pour tester des classes avec des propriétés privées, créez une classe de test spécifique qui expose des méthodes pour les tests.

typescript

```typescript
// Classe de production
export class DataService {
  private data: Map<string, any> = new Map();

  // ... méthodes publiques
}

// Classe de test
export class TestDataService extends DataService {
  public getTestData(): Map<string, any> {
    return this.data; // Accès à la propriété privée via une méthode protégée
  }
}
```

2. Utilisation du pattern AAA (Arrange-Act-Assert)

Structure vos tests selon le pattern AAA pour plus de clarté et de maintenabilité.

typescript

```typescript
test('should correctly adjust tension based on formality level', () => {
  // Arrange
  const registry = new TestProformeRegistry();
  const testProforme = createValidProforme('test-id', 'test-shape', 'test-concept');

  // Act
  registry.setTestProformes([testProforme]);
  registry.simulateAdaptProformalityLevel(0.9);

  // Assert
  const adjustedProforme = registry.getProforme('test-id');
  expect(adjustedProforme?.handshape.tension).toBeGreaterThan(0.5);
});
```

Bonnes pratiques de codage

1. **Respecter l'encapsulation** : Utiliser des méthodes protégées dans la classe de base pour les tests au lieu d'accéder directement aux propriétés privées.
2. **Documentation JSDoc** : Toujours ajouter des commentaires JSDoc complets pour les classes, interfaces et méthodes.
3. **Types explicites** : Éviter l'utilisation de `any` et toujours spécifier les types explicitement.
4. **Vérification des valeurs nulles** : Toujours vérifier si les propriétés existent avant de les utiliser.
5. **Organisation du code** : Garder les fichiers sous la limite de 300 lignes et séparer les responsabilités.

6. **Conventions de nommage** : Utiliser des noms descriptifs pour les variables, méthodes et classes.
7. **Classes de test dédiées** : Créer des classes spécifiques pour les tests qui étendent les classes de base.
8. **Interfaces concises** : Maintenir les interfaces aussi légères que possible, avec uniquement les propriétés nécessaires.
9. **Séparation des préoccupations** : Une classe ne devrait avoir qu'une seule responsabilité.
10. **Gestion des erreurs proactive** : Anticiper les erreurs potentielles et les gérer de manière appropriée.

Ressources supplémentaires

- Documentation officielle TypeScript : https://www.typescriptlang.org/docs/
- Liste complète des codes d'erreur TypeScript : https://github.com/microsoft/TypeScript/blob/main/src/compiler/diagnosticMessages.json
- TypeScript TV - Explications d'erreurs : https://typescript.tv/errors/