

Abstraction Strategy: What Should Be Abstracted?

Question: "Beyond repositories, what else should we abstract? When is abstraction worth the complexity?"

Short Answer: Abstract when you have **multiple implementations** or need **test doubles**. Otherwise, keep it concrete.

Table of Contents

1. Current State Analysis
2. The Abstraction Decision Framework
3. Should Be Abstracted (High Value)
4. Could Be Abstracted (Consider)
5. Should NOT Be Abstracted
6. How to Validate Abstraction Needs
7. Anti-Patterns

Current State Analysis

Already Well-Abstracted

You're doing these right:

1. Data Access (Repositories)

```
// Application layer - Interface
export interface LifeContextRepository {
  getAll(userId: string): Promise<LifeContext[]>;
  save(data: LifeContext): Promise<LifeContext>;
}

// Infrastructure layer - Implementation
export class LifeContextRepositoryImplFirebase implements LifeContextRepository {
  async getAll(userId: string): Promise<LifeContext[]> {
    return firestore().collection('moduleUserData')...
  }
}
```

Why this works:

-  Multiple potential implementations (Firebase, Supabase, local storage)
-  Easy to mock in tests
-  Clear boundary between business logic and data layer

Value: ★★★★☆ (Essential)

2. Business Workflows (Use Cases)

```
@injectable()
export class GetLifeContextUseCase {
  constructor(
    @inject("LifeContextRepository")
    private repository: LifeContextRepository,
  ) {}

  async execute(uid: string): Promise<GetLifeContextResult> {
    // Business logic here
  }
}
```

Why this works:

- Separates business logic from frameworks
- Testable without UI
- Reusable in different contexts (web, mobile, CLI)

Value: ★★★★★ (Essential)

⚠ Opportunities for Abstraction

Things that **could** benefit from abstraction:

The Abstraction Decision Framework

Use this checklist before abstracting anything:

Abstract When

- Multiple implementations exist or are likely
Examples: Database (Firebase/Supabase), Analytics (Firebase/Mixpanel)
- Need to mock for testing
Examples: External APIs, current time, random numbers
- The abstraction is a well-known pattern
Examples: Repository, Strategy, Factory
- Switching is a real business need
Examples: Multi-tenant (different data sources per customer)
- The interface is stable and well-understood
Examples: CRUD operations, authentication

If 3+ boxes checked → Abstract it

Don't Abstract When

- Only one implementation exists and will likely stay that way
Examples: Domain entities, specific business rules
- The abstraction would be more complex than the implementation
Examples: Wrapping a simple utility function
- You're guessing about future needs
Examples: "We might need to switch X someday"
- The interface is unclear or constantly changing
Examples: Early-stage features still being designed
- It's a pure function with no side effects
Examples: Date formatting, calculations

If 2+ boxes checked → Keep it concrete

Should Be Abstracted (High Value)

1. Analytics/Tracking Service ★★★★

Current State: Direct calls scattered throughout codebase

```
// src/screens/Tools/Tools.constants.js
import Events from "../../services/analytics/Events";

const buttons = [
  {
    title: "Kick Diary",
    analytics: Events.TOOLS_KICKCOUNTER, // ← Direct reference
  },
];
```

Problem:

- ✗ Hard to test (analytics fires in tests)
- ✗ Tied to specific analytics provider
- ✗ Can't disable analytics conditionally
- ✗ Hard to add multiple analytics providers

Abstraction:

```
// domain/services/IAnalyticsService.ts
export interface IAnalyticsService {
  trackEvent(eventName: string, properties?: Record<string, any>): void;
  trackScreen(screenName: string): void;
  setUserId(userId: string): void;
}

// infrastructure/analytics/FirebaseAnalyticsService.ts
@Injectable()
export class FirebaseAnalyticsService implements IAnalyticsService {
```

2. Time Provider (For Deterministic Tests) ★★★

Current State: Direct `Date` and `dayjs()` calls

```
// application/services/LifeContextSequenceService.ts
createSequence(contexts: LifeContext[]): LifeContextSequenceElement[] {
  const now = dayjs(); // ← Hard to test "what if it's tomorrow?"

  if (dayjs().isAfter(lastContextEnd)) {
    endDate = dayjs().add(2, 'month')...
  }
}
```

Problem:

- ✗ Tests dependent on current time
- ✗ Can't test "time travel" scenarios
- ✗ Flaky tests (pass today, fail tomorrow)

Abstraction:

```
// application/services/ITimeProvider.ts
export interface ITimeProvider {
  now(): Date;
  today(): Date; // midnight of current day
}

// infrastructure/time/SystemTimeProvider.ts
@Injectable()
export class SystemTimeProvider implements ITimeProvider {
  now(): Date {
    return new Date();
  }

  today(): Date {
    return dayjs().startOf("day").toDate();
  }
}
```

3. Feature Flags / Configuration Service ★★★

Current State: Hardcoded or environment variables

```
// Scattered throughout codebase
if (process.env.ENABLE_BETA_FEATURE === "true") {
    // Show beta feature
}

if (__DEV__) {
    // Dev-only code
}
```

Problem:

- ✗ Can't toggle features at runtime
- ✗ Can't A/B test features
- ✗ Can't enable features for specific users
- ✗ Hard to test both on/off states

Abstraction:

```
// application/services/IFeatureFlagService.ts
export interface IFeatureFlagService {
    isEnabled(flagName: string, userId?: string): boolean;
    getVariant(experimentName: string, userId?: string): string;
}

// infrastructure/featureFlags/RemoteConfigFeatureFlagService.ts
@Injectable()
export class RemoteConfigFeatureFlagService implements IFeatureFlagService {
    isEnabled(flagName: string, userId?: string): boolean {
        // Fetch from Firebase Remote Config
        return remoteConfig().getBoolean(flagName);
    }
}
```

Could Be Abstracted (Consider)

4. Notification Service ★★

Current State: Direct calls to UI notifications

```
// Scattered in components
import { showMessage } from "react-native-flash-message";

showMessage({
  message: "Success",
  type: "success",
});
```

Should abstract if:

- ✓ You want consistent notification styling
- ✓ You might switch notification libraries
- ✓ You want to test notification logic
- ✓ You need notification queueing/priority

Don't abstract if:

- ✗ Notifications are simple and won't change
- ✗ Only used in a few places
- ✗ UI framework-specific features are needed

Recommendation: ★★ (Medium - useful but not essential)

5. Storage Service (Beyond Repositories) ★★

Current State: Direct AsyncStorage/SecureStore calls

```
import AsyncStorage from "@react-native-async-storage/async-storage";
await AsyncStorage.setItem("user_preferences", JSON.stringify(prefs));
```

Abstraction:

```
// application/services/IStorageService.ts
export interface IStorageService {
  get<T>(key: string): Promise<T | null>;
  set<T>(key: string, value: T): Promise<void>;
  remove(key: string): Promise<void>;
  clear(): Promise<void>;
}

// infrastructure/storage/AsyncStorageService.ts
@Injectable()
export class AsyncStorageService implements IStorageService {
  async get<T>(key: string): Promise<T | null> {
    const value = await AsyncStorage.getItem(key);
    return value ? JSON.parse(value) : null;
  }

  async set<T>(key: string, value: T): Promise<void> {
    await AsyncStorage.setItem(key, JSON.stringify(value));
  }

  async remove(key: string): Promise<void> {
    await AsyncStorage.removeItem(key);
  }

  async clear(): Promise<void> {
    await AsyncStorage.clear();
  }
}
```

6. Network Service ★★

Current State: Direct fetch/axios calls (if any)

Should abstract if:

- Making many API calls
- Need consistent error handling
- Need retry logic
- Need request/response interceptors

Don't abstract if:

- All data access goes through Firebase (repositories already abstract it)
- Few external API calls

Recommendation: ★ (Low - you're using Firebase, not REST APIs)

Should NOT Be Abstracted

These should stay concrete:

1. Domain Entities ✗ Don't Abstract

```
// ✓ GOOD: Concrete entity
export class Pregnancy {
  constructor(
    public id: string,
    public dueDate: Date,
    public metadata: PregnancyMetadata,
  ) {}

  getDueDate(): Date {
    return this.dueDate;
  }
}

// ✗ BAD: Abstract entity (why?)
export interface IPregnancy {
  id: string;
  dueDate: Date;
  getDueDate(): Date;
}
```

Why not abstract:

- Domain entities are your business model
- They should be concrete, not polymorphic
- No need for multiple implementations
- Makes code harder to understand

2. Pure Functions ✕ Don't Abstract

```
// ✓ GOOD: Pure function
export const formatDate = (date: Date, format: string): string => {
  return dayjs(date).format(format);
};

// ✗ BAD: Abstracted utility (over-engineering)
export interface IDateFormatter {
  format(date: Date, format: string): string;
}
```

Why not abstract:

- Pure functions are already testable
- No side effects to isolate
- No implementations to swap
- Adds unnecessary complexity

3. UI Components ✗ Don't Abstract

```
// ✓ GOOD: Concrete component
export const Button = ({ title, onPress }: ButtonProps) => {
  return <TouchableOpacity onPress={onPress}>...</TouchableOpacity>;
};

// ✗ BAD: Abstract button (why?)
export interface IButton {
  render(): JSX.Element;
}
```

Why not abstract:

- UI frameworks are already component-based
- No need for abstraction layer
- Makes code harder to read
- React components are the abstraction

4. Simple Value Objects ✗ Don't Abstract

```
// ✓ GOOD: Concrete value object
export class Week {
  constructor(public readonly value: number) {
    if (value < 1 || value > 52) {
      throw new Error("Invalid week");
    }
  }
}

// ✗ BAD: Abstract value object (why?)
export interface IWeek {
  value: number;
}
```

5. Configuration Constants ✖ Don't Abstract

```
// ✓ GOOD: Simple constants
export const API_TIMEOUT = 5000;
export const MAX_RETRY_ATTEMPTS = 3;

// ✖ BAD: Over-abstracted config (unnecessary)
export interface IConfiguration {
  getApiTimeout(): number;
  getMaxRetryAttempts(): number;
}
```

How to Validate Abstraction Needs

The "Three Implementation Test"

Before abstracting, ask:

"Can I name three concrete implementations of this abstraction?"

Example 1: Analytics ✓

1. FirebaseAnalyticsService
2. MixpanelAnalyticsService
3. NoOpAnalyticsService (for tests)

Result: Good abstraction candidate

Example 2: Date Formatting ✗

1. DayjsDateFormatter
2. ??? (Can't think of a second)
3. ??? (Definitely can't think of a third)

Result: Keep it as a function

The "Test Without Mocking" Test

Try to test WITHOUT mocking:

```
// If you can test this easily without mocking, don't abstract
describe("formatDate", () => {
  it("should format date", () => {
    expect(formatDate(new Date("2024-01-15"), "YYYY-MM-DD")).toBe("2024-01-15");
  });
});

// If you can't test without mocking, consider abstraction
describe("GetLifeContextUseCase", () => {
  it("should fetch contexts", async () => {
    // Need to mock repository - good abstraction!
    const mockRepo = { getAll: jest.fn().mockResolvedValue([]) };
    const useCase = new GetLifeContextUseCase(mockRepo);
  });
});
```

The "Business vs Technical" Test

Business abstractions: Usually good

- Repositories (data access patterns)
- Use cases (business workflows)
- Domain services (business rules)

Technical abstractions: Evaluate carefully

- Analytics (good - might switch providers)
- Logging (good - might need multiple outputs)
- Date utilities (bad - pure functions work fine)

Anti-Patterns

1. Abstracting Too Early ("Premature Abstraction")

```
// ✗ BAD: Abstracting before you need it
export interface IEmailSender {
  send(to: string, subject: string, body: string): Promise<void>;
}

// Only one implementation, will likely stay that way
export class SendGridEmailSender implements IEmailSender {}
```

Problem: YAGNI (You Aren't Gonna Need It)

Solution: Wait until you have a real need for multiple implementations

2. "Interface for Everything" Syndrome

```
// ❌ BAD: Everything is an interface
export interface IUser {}
export interface IUserService {}
export interface IUserRepository {}
export interface IUserValidator {}
export interface IUserDTO {}
export interface IUserMapper {}
```

Problem: Unnecessary abstraction layers

Solution: Only abstract at boundaries (repositories, external services)

3. Single-Method Interfaces

```
// ✗ BAD: Interface with one method (just use a function)
export interface IDateFormatter {
  format(date: Date): string;
}

// ✓ GOOD: Just use a function
export type DateFormatter = (date: Date) => string;
```

4. Leaking Abstractions

```
// ✗ BAD: Abstraction exposes implementation details
export interface IAnalyticsService {
  trackEvent(event: FirebaseAnalyticsEvent): void; // ← Firebase-specific!
}

// ✓ GOOD: Generic interface
export interface IAnalyticsService {
  trackEvent(eventName: string, properties?: Record<string, any>): void;
}
```

Decision Matrix

Use this to decide:

Factor	Abstract	Don't Abstract
Multiple implementations	✓ Yes or likely	✗ No and unlikely
Need mocking for tests	✓ External dependencies	✗ Pure functions
Crossing layer boundaries	✓ Yes (e.g., data access)	✗ No (same layer)
Well-known pattern	✓ Repository, Strategy	✗ Custom one-off
Stable interface	✓ CRUD, auth patterns	✗ Still evolving
Business requirement	✓ Multi-tenant, plugins	✗ Theoretical future

Recommended Abstractions for Your App

Immediate (Do Now) ★★★★

1. Analytics Service

- Clear business need (might switch providers)
- Already scattered across codebase
- Easy to test without side effects
- **Effort:** 2-3 hours

Near-Term (Consider) ★★★

2. Time Provider

- If you have flaky time-dependent tests
- Enables deterministic testing
- **Effort:** 1-2 hours

3. Feature Flag Service

- If you need A/B testing
- If you do gradual rollouts
- **Effort:** 3-4 hours (includes Remote Config setup)

Optional (As Needed) ★★

4. Notification Service

- If notifications become complex
- **Effort:** 1-2 hours

Implementation Example: Analytics Service

Step 1: Define Interface

```
// domain/services/IAnalyticsService.ts
export interface IAnalyticsService {
  trackEvent(eventName: string, properties?: Record<string, any>): void;
  trackScreen(screenName: string): void;
  setUserId(userId: string): void;
  setUserProperties(properties: Record<string, any>): void;
}
```

Step 2: Implement for Firebase

```
// infrastructure/analytics/FirebaseAnalyticsService.ts
import analytics from "@react-native-firebase/analytics";
import { injectable } from "tsyringe";
import { IAnalyticsService } from "@domain/services/IAnalyticsService";

@injectable()
export class FirebaseAnalyticsService implements IAnalyticsService {
  trackEvent(eventName: string, properties?: Record<string, any>): void {
    analytics().logEvent(eventName, properties);
  }

  trackScreen(screenName: string): void {
    analytics().logScreenView({
      screen_name: screenName,
      screen_class: screenName,
    });
  }

  setUserId(userId: string): void {
    analytics().setUserId(userId);
  }

  setUserProperties(properties: Record<string, any>): void {
    analytics().setUserProperties(properties);
  }
}
```

Summary

Abstraction Philosophy

"Duplication is far cheaper than the wrong abstraction." - Sandi Metz

Three rules:

1. **Abstract at boundaries** (data, external services)
2. **Abstract when you have proof** (multiple implementations exist or coming soon)
3. **Keep domain concrete** (entities, value objects, business rules)

Your Action Plan

Phase 1 (Now):

- Keep repositories (already doing well)
- Keep use cases (already doing well)
- Add Analytics abstraction (high value, clear need)

Phase 2 (If Needed):

- Time Provider (if tests are flaky)
- Feature Flags (if doing A/B testing)

Never:

- Abstract domain entities
- Abstract pure functions
- Abstract UI components
- Abstract "Hello World" code

Quick Reference Checklist

Before abstracting, check:

- [] Can I name 3 implementations?
- [] Do I need to mock this for tests?
- [] Is this crossing a layer boundary?
- [] Is the interface stable and well-understood?
- [] Is there a business reason (not just "might need it")?

3+ checks = Abstract

2 or fewer = Keep concrete

See Also:

- [Architecture Analysis Report](#) - Current architecture
- [Quick Reference](#) - Daily decisions
- Martin Fowler: "When to make a type" - <https://martinfowler.com/ieeeSoftware/whenType.pdf>

Last Updated: 2025-11-19