Pragmatic Android Dependency Injection

Workshop outline

- Theory talk (~45 minutes).

- Break.

- Guided app refactor (~ 2 hours).

- Wrap-up (5 minutes).

Goals

- **Understanding** of DI principles and benefits.

- **Experience** adding manual DI to common architectures.

- **Awareness** of the costs/benefits of DI frameworks.

I want this workshop to change how you write code.

Talk

What is a dependency?

When a class C uses functionality from a type D *to perform its own functions*, then D is called a **dependency** of C.

C is called a **consumer** of D.

Why do we use dependencies?

- To share logic and keep our code **DRY**.

- To model logical abstractions, **minimizing cognitive load**.

A consumer/dependency example

```java
public class TimeStamper {              // Consumer
  private SystemClock systemClock;  // Dependency

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

A consumer/dependency example

```java
public class TimeStamper {              // Consumer
  private SystemClock systemClock;   // Dependency

  public TimeStamper() {
    systemClock = new SystemClock();
  }


  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

A consumer/dependency example

```java
public class TimeStamper {              // Consumer
  private SystemClock systemClock;  // Dependency

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

Android consumers

In Android, **important consumers** include:

• activities/fragments,

• *presenters/view models.*

These classes are the hearts of our apps. Their capabilities include transforming app state into UI state, processing user input, coordinating network requests, and applying business rules. **Testing them is valuable!**

Android dependencies

In Android, **common dependencies** include:

- API clients,

- local storage,

- clocks,

- geocoders,

- user sessions.

Android consumer/dependency examples

- A login **fragment** that uses an *API client* to submit user credentials to a backend.

- A choose sandwich **presenter** that uses *local storage* to track the last sandwich ordered.

- A choose credit card **view model** that uses a *clock* to determine which cards are expired.

Dependency dependencies

Some classes act as **both** consumers and dependencies.

Example: an API client may consume a class that assists with local storage (for caching) **and** be consumed by presenters.

The relationships between all dependencies in an app are collectively referred to as the **dependency graph**.

Mommy, where do dependencies come from?

- *Consumers* create dependencies themselves (**hard-coded**).

- *Consumers* ask an external class for their dependencies (**service locator**).

- *An external class* injects a consumer's dependencies via constructors or setters (**dependency injection**).

Hard-coded dependencies, v1

```java
public class TimeStamper {
  private SystemClock systemClock;

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

Hard-coded dependencies, v1

```java
public class TimeStamper {
  private SystemClock systemClock;

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

Hard-coded dependencies, v2

```java
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + new SystemClock().now();
  }
}
```

Hard-coded dependencies, v2

```java
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + new SystemClock().now();
  }
}
```

Hard-coded dependencies, v3

```
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + SystemClock.shared().now();
  }
}
```

Hard-coded dependencies, v3

```
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + SystemClock.shared().now();
  }
}
```

Hard-coded dependencies, v4

```java
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + SystemClock.now();
  }
}
```

Hard-coded dependencies, v4

```
public class TimeStamper {
  public String getTimeStamp() {
    return "The time is " + SystemClock.now();
  }
}
```

Hard-coding hardships

A consumer's dependencies are **hidden**:

```
// Dependency on SystemClock is invisible:
TimeStamper timeStamper = new TimeStamper();
System.out.println(timeStamper.getTimeStamp());
```

Hard-coding hardships

A consumer with *impure* dependencies will be **very hard to unit test at all**:

```
public class TimeStamperTest {
  @Test
  public String testGetTimeStamp() {
    String expected = "The time is 12:34";
    String actual = new TimeStamper().getTimeStamp();
    assertEquals(expected, actual); // Almost always fails.
  }
}
```

Hard-coding hardships

A consumer that hard-codes access to *singletons* may have **brittle/slow/lying unit tests** (if state leaks between tests).

Improving on hard-coding

- Make consumer dependency needs **explicit** (by receiving instances through constructors or setters).
  => Also decouples consumer from dependency lifetime.

- Express dependency needs using **interfaces (behaviors)** rather than classes (implementations).
  => Allows mock implementations to be supplied in unit tests.

These are the elements of robust **dependency injection**!

Doing DI: Before

```java
public class TimeStamper {
  private SystemClock systemClock;

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

Doing DI: Before

```java
public class SystemClock {
  public String now() {
    return LocalDateTime
        .now()
        .format(DateTimeFormatter.ofPattern("hh:mm"));
  }
}
```

Doing DI: Identifying dependencies

```java
public class TimeStamper {
  private SystemClock systemClock;

  public TimeStamper() {
    systemClock = new SystemClock();
  }             // ^^^^^^^^^^^^^^^^^^ A (hard-coded) *dependency*!

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }
}
```

Doing DI: Identifying behaviors

```java
public class TimeStamper {
  private SystemClock systemClock;

  public TimeStamper() {
    systemClock = new SystemClock();
  }

  public String getTimeStamp() {
    return "The time is " + systemClock.now();
  }                              // ^^^^^^^^^^^^^^^^^ The *behavior* we rely on.
}
```

Doing DI: Defining interfaces

```java
// Describes the *behavior* our consumer relies on:
public interface IClock {
  String now();
}


// Is now one possible supplier of IClock behavior:
public class SystemClock implements IClock {
  @Override
  public String now() {
    return LocalDateTime
        .now()
        .format(DateTimeFormatter.ofPattern("hh:mm"));
  }
}
```

Doing DI: Demanding dependencies (constructor)

```java
public class TimeStamper {
  private IClock clock;

  public TimeStamper(IClock clock) {
    this.clock = clock;
  }

  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (constructor)

```java
public class TimeStamper {
  private IClock clock;

  public TimeStamper(IClock clock) {
    this.clock = clock;
  }

  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (constructor)

```java
public class TimeStamper {
  private IClock clock;

  public TimeStamper(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (constructor)

```java
public class TimeStamper {
  private IClock clock;

  public TimeStamper(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (setter)

```java
public class TimeStamper {
  private IClock clock;

  public void setClock(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (setter)

```java
public class TimeStamper {
  private IClock clock;

  public void setClock(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (setter)

```java
public class TimeStamper {
  private IClock clock;

  public void setClock(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Demanding dependencies (setter)

```java
public class TimeStamper {
  private IClock clock;

  public void setClock(IClock clock) {
    this.clock = clock;
  }


  public String getTimeStamp() {
    return "The time is " + clock.now();
  }
}
```

Doing DI: Manual injection

Owners of consumers create/locate and inject dependencies:

```
// Constructor injection in production code:
TimeStamper timeStamper = new TimeStamper(new SystemClock());
System.out.println(timeStamper.getTimeStamp());
```

Doing DI: Manual injection

```java
// Mock clock created for use during tests:
public class MockClock implements IClock {
  private String fixedTime;

  public MockClock(String fixedTime) {
    this.fixedTime = fixedTime;
  }

  @Override
  public String now() {
    return fixedTime;
  }
}
```

Doing DI: Manual injection

```java
// Constructor injection in test code:
public class TimeStamperTest {
  @Test
  public String testGetTimeStamp() {
    String expected = "The time is 12:34";
    IClock mockClock = new MockClock("12:34");
    String actual = new TimeStamper(mockClock).getTimeStamp();
    assertEquals(expected, actual); // Always passes.
  }
}
```

Doing DI: Manual injection

- ✅ Simplest injection technique.

- ✅ Dependency lifetimes controlled using familiar methods.

- ✅ Sufficient for all unit testing needs.

- ❌ Repetitive.

- ❌ Can scale poorly if your dependency graph is deep e.g. `new D1(new D2(new D3(...), ...), ...)`.

- ❌ Insufficient for reliable UI testing.

Doing DI: Framework injection

Most Java/Kotlin DI frameworks are structured similarly:

- Centralized code describes the entire dependency graph

- Consumers add `@Inject` annotations to their dependencies

- Classes call an `inject` method to trigger injection

The details are (much) more complicated, but that's the gist.

Doing DI: Framework injection

- ✅ DRY.

- ✅ Makes dependency graph very explicit.

- ✅ Sufficient for all unit testing needs.

- ✅ Sufficient for all UI testing needs.

- ❌ Frameworks are difficult to learn and use effectively.

- ❌ Dependency lifetime management can get complicated.

- ❌ Longer build times/some performance impact.

I say...

Use a framework if:

- your app needs extensive UI test coverage

- your app has a deep dependency graph

- your app swaps dependency implementations at runtime

- you are already comfortable with DI principles

Otherwise, **prefer manual constructor injection.**

The community says...

So, dependency injection frameworks...
Dagger 2 had a learning curve, but it
seemed like a necessary evil. Dagger-
Android broke me. I still don't understand
most of it. @insertkoin_io has me enthused
again. Don't @ me about dependency
injection vs service locators 🙈

Craig Russell @trionkidnapper
7:36pm - 21 Mar 2019

@botteaap @lehtimaeki we don't use dagger. 😉 Dependency injection is key in order to test code, but a DI framework really isn't.

Jeroen Mols @molsjeroen
6:26am - 22 Mar 2019

@trionkidnapper @botteaap @lehtimaeki
no dependency injection framework at all.
(That's also what is making our builds crazy
fast) We pass stuff manually in constructors
and use default methods (or factories) to
simpilfy object creation.

Jeroen Mols @molsjeroen
6:53am - 22 Mar 2019

@molsjeroen @trionkidnapper @botteaap @lehtimaeki We also do the same. it makes readability and navigation in code much easier.

Ozan Doksöz @odoksoez

7:39am - 22 Mar 2019

@molsjeroen @trionkidnapper @botteaap @lehtimaeki We do this too, and I find it makes onboarding new devs really fast.

Emmett Wilson @MathematicalFnk
7:20am - 22 Mar 2019

</talk>
Questions?

Guided App Refactor

Speedy Subs

Speedy Subs is a small sandwich-ordering app.

Each major screen is structured differently (MVC vs MVP vs MVVM).

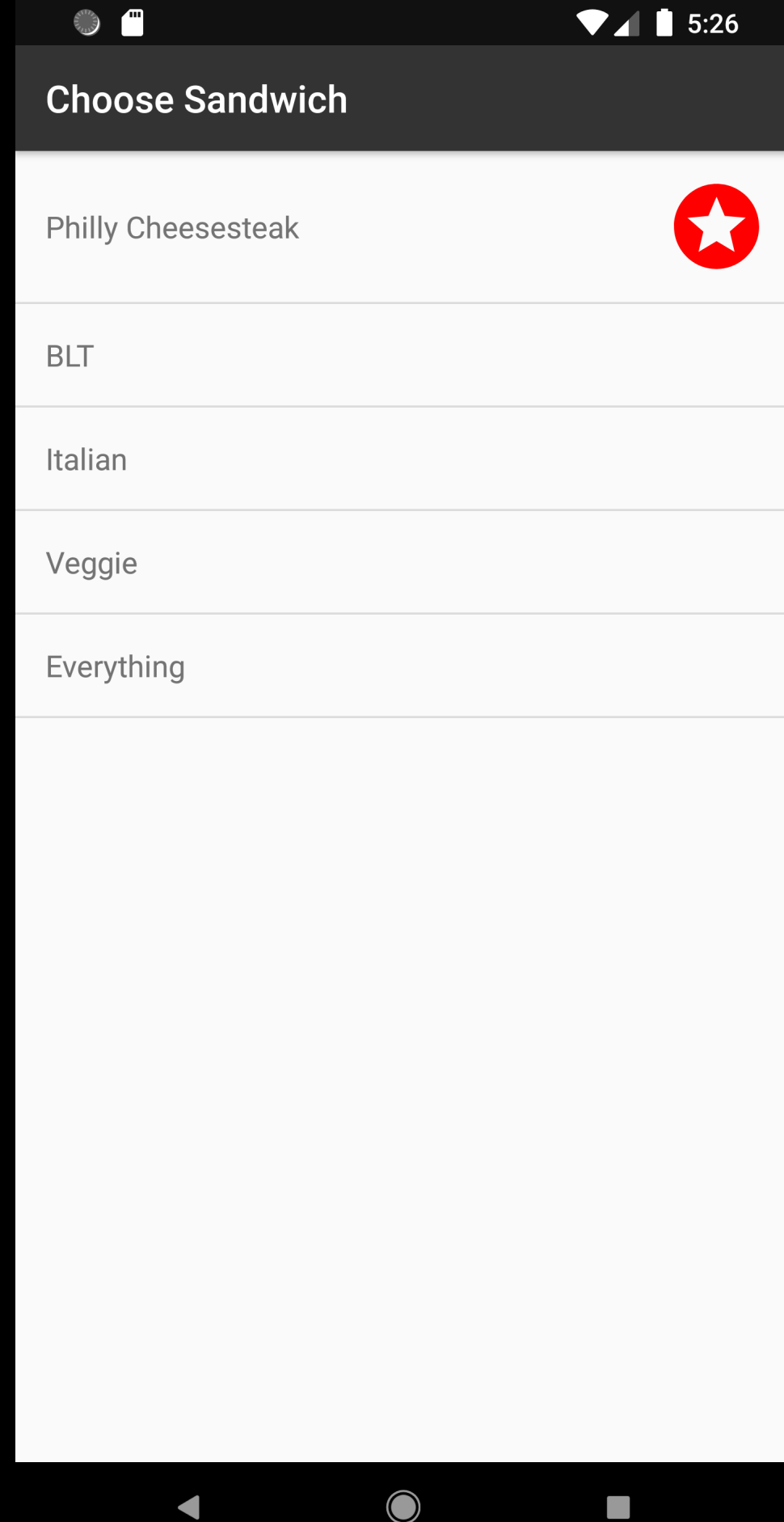We will refactor each screen to allow unit testing via DI.

Login

- MVC (fat Fragment)
- **Username is validated**
- **Password is validated**
- *Login request is made on submit*
- Choose Sandwich screen is launched on success

Login

any

strings

SUBMIT

Choose Sandwich

- MVP
- *Sandwiches are fetched from network on screen launch*
- **Last-ordered sandwich is listed first**
- **Other sandwiches are listed in order received**
- Choose Credit Card screen is launched on row tap
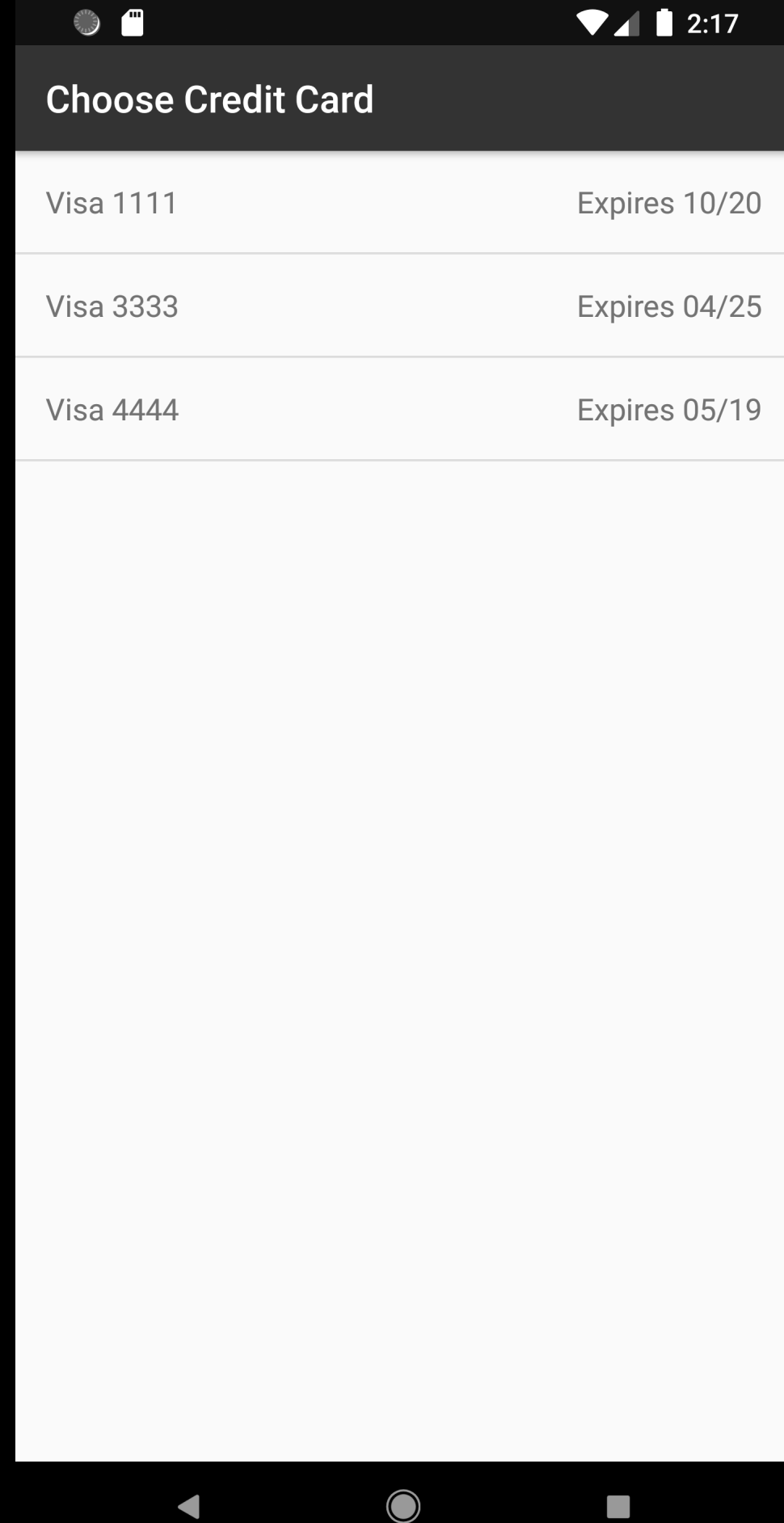
Choose Credit Card

- MVVM (w/ LiveData)
- Credit cards are initially populated from login response
- *Screen implements pull-to-refresh*
- **Only non-expired credit cards are listed**
- *Order is submitted on row tap*
- Choose Sandwich screen is launched on success

Confirmation

- Back and Done buttons return us to the login screen.

Order Confirmation

Order 1236340346 successfully placed!

DONE

Key classes

- `MainActivity`: application entry point.

- `Session`: stores info about the current customer and order.

- `OrderingApi`: group of methods for calling (fake) backend. Simulates delayed network responses.

Ready, set, refactor

Wrap-up

DI IRL

- Refactor to MV(something) first.

- Plan to implement DI progressively.

- Focus on areas in need of tests (important + fragile).

- Start manual, swap to a framework if needed.

Further learning

- (talk) Dependency Injection Made Simple by Dan Lew

- (book) Dependency Injection Principles, Practices, and Patterns by Steven van Deursen and Mark Seemann