Design Patterns for Mobile Devices

1. **Creational patterns**: How you *create* objects.
2. **Structural patterns**: How you *compose* objects.
3. **Behavioral patterns**: How you *coordinate* object interactions.

Design patterns usually deal with objects. They present a solution to a reoccurring problem that an object shows and help eradicate design-specific problems. In other words, they represent challenges, other developers already faced and prevent you from reinventing the wheel by showing you proven ways to solve those problems.

Creational Patterns

- Builder
- Dependency Injection
- Singleton
- Factory

Structural Patterns

- Adapter
- Facade
- Decorator
- Composite

Behavioral Patterns

- Command
- Observer

- Strategy
- State

Creational Patterns

"When I need a particularly complex object, how do I get an instance of it?" – Future You

Future You hopes the answer isn't "*Just copy and paste the same code every time you need an instance of this object*". Instead, **Creational** patterns make object instantiation straightforward and repeatable.

Builder

At a certain restaurant, you create your own sandwich: you choose the bread, ingredients and condiments you'd like on your sandwich from a checklist on a slip of paper. Even though the checklist instructs you to *build your own* sandwich, you only fill out the form and hand it over the counter. You don't build the sandwich, just customize and consume it. :]

Similarly, the **Builder** pattern simplifies the creation of objects, like slicing bread and stacking pickles, from its representation, a yummy sandwich. Thus, the same construction process can create objects of the same class with different properties.

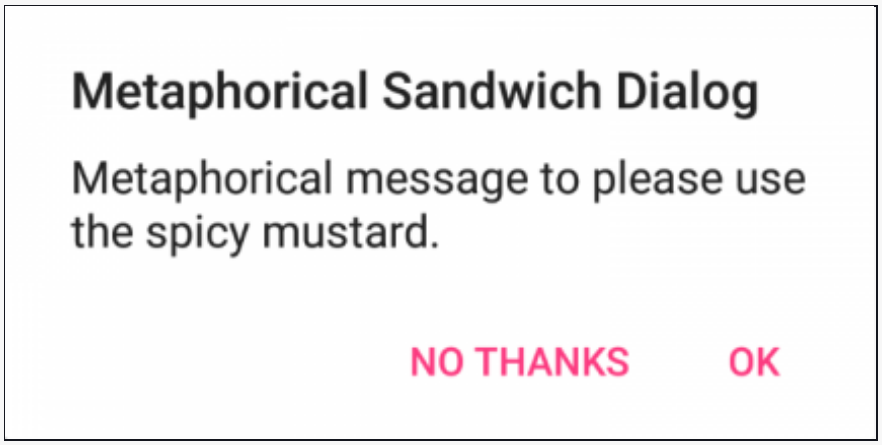
In Android, an example of the Builder pattern is `AlertDialog.Builder`:

```
AlertDialog.Builder(this)
```

```
.setTitle("Sandwich Dialog")

.setMessage("Please use the spicy mustard.")

.setNegativeButton("No thanks") { dialogInterface, i ->

  // "No thanks" action

}

.setPositiveButton("OK") { dialogInterface, i ->

  // "OK" action

}

.show()
```

This builder proceeds step-by-step and lets you specify only the parts of `AlertDialog` that you need to specify. Take a look at the AlertDialog.Builder documentation. You'll see there are quite a few commands to choose from when building your alert.

The code block above produces the following alert:



A different set of choices would result in a completely different sandwich– er, alert. :]

Dependency Injection

Dependency injection is like moving into a furnished apartment. Everything you need is already there. You don't have to wait for furniture delivery or follow pages of IKEA instructions to put together a Borgsjö bookshelf.

In software terms, dependency injection has you provide any required objects to instantiate a new object. This new object doesn't need to construct or customize the objects themselves.

In Android, you might find you need to access the same complex objects from various points in your app, such as a network client, image loader or `SharedPreferences` for local storage. You can **inject** these objects into your activities and fragments and access them right away.

Currently, there are three main libraries for dependency injection: Dagger '2', Dagger Hilt, and Koin. Let's take a look at an example with Dagger. In it you annotate a class with `@Module`, and populate it with `@Provides` methods like:

```
@Moduleclass AppModule(private val app: Application) {

  @Provides

  @Singleton

  fun provideApplication(): Application = app



  @Provides

  @Singleton
```

```
  fun provideSharedPreferences(app: Application): SharedPreferences {
    return app.getSharedPreferences("prefs", Context.MODE_PRIVATE)
  }
}
```

The module above creates and configures all required objects. As an additional best practice in larger apps, you could create multiple modules separated by function.

Then, you make a `Component` interface to list your modules and the classes you'll inject:

```
@Singleton@Component(modules = [AppModule::class])interface AppComponent {
  fun inject(activity: MainActivity)
  // ...
}
```

The component ties together where the dependencies are *coming from*, the modules, and where they're *going to*, the injection points.

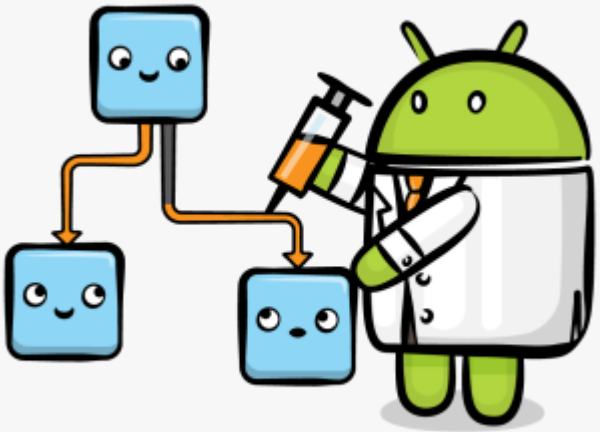
Finally, you use the `@Inject` annotation to request the dependency wherever you need it, along with `lateinit` to initialize a non-nullable property after you create the containing object:

```
@Inject lateinit var sharedPreferences: SharedPreferences
```

As an example, you could use this in your `MainActivity` and then use local storage, *without* the Activity needing to know how the `SharedPreferences` object came to be.

Admittedly, this is a simplified overview, but you can read the Dagger documentation for more implementation details. You can also click the links above in the mentioned libraries for in-depth tutorials for each topic.

This pattern may seem complicated and *magical* at first, but it can help simplify your activities and fragments.



Singleton

The **Singleton** pattern specifies that only a single instance of a class should exist with a global access point. This pattern works well when modeling real-world objects with only one instance. For example, if you have an object that makes network or database connections, having more than one instance of the project may cause problems and mix data. That's why in some scenarios you want to restrict the creation of more than one instance.

The Kotlin `object` keyword declares a singleton without needing to specify a static instance like in other languages:

```
object ExampleSingleton {

  fun exampleMethod() {

    // ...

  }

}
```

When you need to access members of the singleton object, you make a call like this:

```
ExampleSingleton.exampleMethod()
```

Behind the scenes, an `INSTANCE` static field backs the Kotlin object. So, if you need to use a Kotlin object from Java code, you modify the call like this:

```
ExampleSingleton.INSTANCE.exampleMethod();
```

By using `object`, you'll know you're using the same instance of that class throughout your app.

The Singleton is probably the most straightforward pattern to understand initially but can be dangerously easy to overuse and abuse. Since it's accessible from multiple objects, the singleton can undergo unexpected side effects that are difficult to track down, which is exactly what Future You doesn't want to deal with. It's important to understand the pattern, but other design patterns may be safer and easier to maintain.

Factory

As the name suggests, **Factory** takes care of all the object creational logic. In this pattern, a factory class controls which object to instantiate. Factory pattern comes in handy when dealing with many common objects. You can use it where you might not want to specify a concrete class.

Take a look at the code below for a better understanding:

```
// 1interface HostingPackageInterface {
  fun getServices(): List<String>
}
// 2enum class HostingPackageType {
  STANDARD,
  PREMIUM
}
// 3class StandardHostingPackage : HostingPackageInterface {
  override fun getServices(): List<String> {
    return ...
  }
}
// 4class PremiumHostingPackage : HostingPackageInterface {
  override fun getServices(): List<String> {
    return ...
```

```
  }
}
// 5object HostingPackageFactory {
  // 6
  fun getHostingFrom(type: HostingPackageType): HostingPackageInterface {
    return when (type) {
      HostingPackageType.STANDARD -> {
        StandardHostingPackage()
      }
      HostingPackageType.PREMIUM -> {
        PremiumHostingPackage()
      }
    }
  }
}
```

Here's a walk through the code:

1. This is a basic interface for all the hosting plans.
2. This enum specifies all the hosting package types.
3. StandardHostingPackage conforms to the interface and implements the required method to list all the services.

4. `PremiumHostingPackage` conforms to the interface and implements the required method to list all the services.
5. `HostingPackageFactory` is a singleton class with a helper method.
6. `getHostingFrom` inside `HostingPackageFactory` is responsible for creating all the objects.

You can use it like this:

```
val standardPackage = HostingPackageFactory.getHostingFrom(HostingPackageType.STANDARD)
```

It helps to keep all object creation in one class. If used inappropriately, a Factory class can get bloated due to excessive objects. Testing can also become difficult as the factory class itself is responsible for all the objects.

Structural Patterns

"So, when I open this class, how will I remember what's it doing and how it's put together?" – Future You

Future You will undoubtedly appreciate the Structural Patterns you used to organize the guts of your classes and objects into familiar arrangements that perform typical tasks. **Adapter** and **Facade** are two commonly-seen patterns in Android.

Adapter

A famous scene in the movie **Apollo 13** features a team of engineers tasked with fitting a square peg into a round hole. This, metaphorically, is the role of an adapter. In software terms, this pattern lets two incompatible classes work together by converting a class's interface into the interface the client expects.

Consider your app's business logic. It might be a Product or a User or Tribble. It's the square peg. Meanwhile, a `RecyclerView` is the same basic object across all Android apps. It's the round hole.

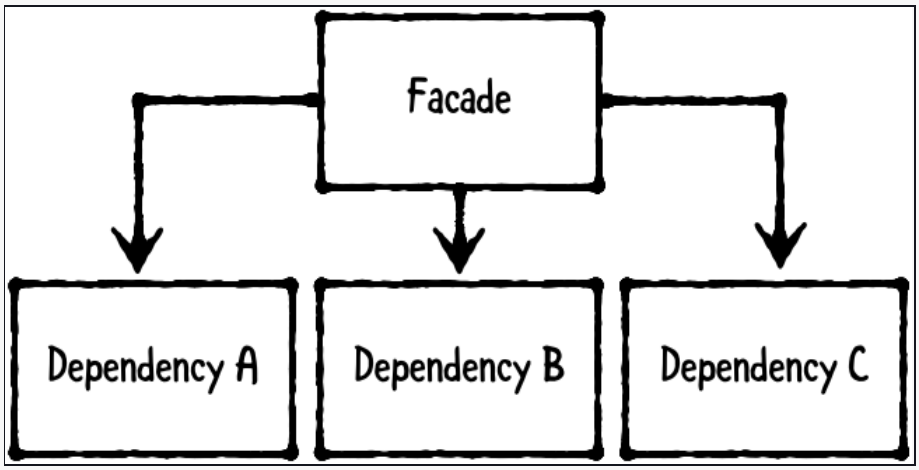
In this situation, you can use a subclass of `RecyclerView.Adapter` and implement the required methods to make everything work:

```
class TribbleAdapter(private val tribbles: List<Tribble>) : RecyclerView.Adapter<TribbleViewHolder>() {

 override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int): TribbleViewHolder {

   val inflater = LayoutInflater.from(viewGroup.context)

   val view = inflater.inflate(R.layout.row_tribble, viewGroup, false)

   return TribbleViewHolder(view)

 }



 override fun onBindViewHolder(viewHolder: TribbleViewHolder, itemIndex: Int) {

   viewHolder.bind(tribbles[itemIndex])

 }



 override fun getItemCount() = tribbles.size

}
```

`RecyclerView` doesn't know what a Tribble is, as it's never seen a single episode of **Star Trek**, not even the new movies. :] Instead, it's the adapter's job to handle the data and send the `bind` command to the correct `ViewHolder`.

Facade

The **Facade** pattern provides a higher-level interface that makes a set of other interfaces easier to use. The following diagram illustrates this idea in more detail:



If your Activity needs a list of books, it should be able to ask a single object for that list *without* understanding the inner workings of your local storage, cache and API client. Beyond keeping your Activities and Fragments code clean and concise, this lets Future You make any required changes to the API implementation without impacting the Activity.

Square's Retrofit is an open-source Android library that helps you implement the Facade pattern. You create an interface to provide API data to client classes like so:

```
interface BooksApi {

  @GET("books")

  fun listBooks(): Call<List<Book>>
```

```
}
```

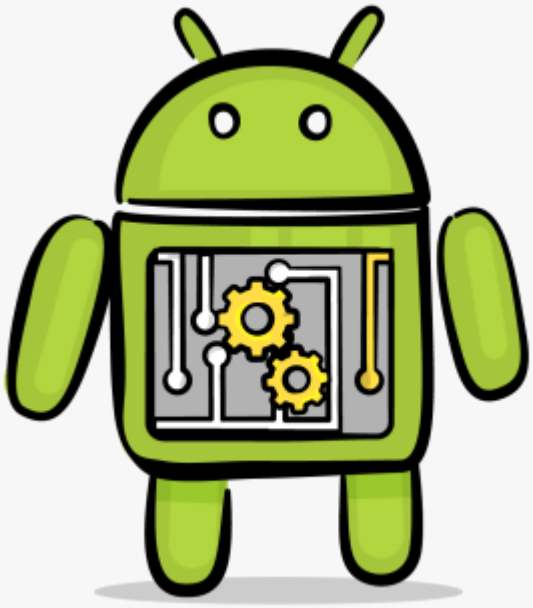
The client needs to call `listBooks()` to receive a list of `Book` objects in the callback. It's nice and clean. For all it knows, you could have an army of Tribbles assembling the list and sending it back via transporter beam. :]

This lets you make all types of customizations underneath without affecting the client. For example, you can specify a customized JSON deserializer that the Activity has no clue about:

```
val retrofit = Retrofit.Builder()

  .baseUrl("http://www.myexampleurl.com")

  .addConverterFactory(GsonConverterFactory.create())

  .build()

val api = retrofit.create<BooksApi>(BooksApi::class.java)
```

Notice the use of `GsonConverterFactory`, working behind the scenes as a JSON deserializer. With Retrofit, you can further customize operations with `Interceptor` and `OkHttpClient` to control caching and logging behavior without the client knowing what's going on.

The less each object knows about what's going on behind the scenes, the easier it'll be for Future You to manage changes in the app.



Decorator

The **Decorator** pattern dynamically attaches additional responsibilities to an object to extended its functionality at runtime. Take a look at the example below:

```
//1interface Salad {
  fun getIngredient(): String
}
//2class PlainSalad : Salad {
  override fun getIngredient(): String {
    return "Arugula & Lettuce"
  }
}
//3open class SaladDecorator(protected var salad: Salad) : Salad {
```

```
  override fun getIngredient(): String {
    return salad.getIngredient()
  }
}
//4class Cucumber(salad: Salad) : SaladDecorator(salad) {
  override fun getIngredient(): String {
    return salad.getIngredient() + ", Cucumber"
  }
}
//5class Carrot(salad: Salad) : SaladDecorator(salad) {
  override fun getIngredient(): String {
    return salad.getIngredient() + ", Carrot"
  }
}
```

Here's what the above code defines:

1. A `Salad` interface helps with knowing the ingredients.
2. Every salad needs a base. This base is `Arugula & Lettuce` thus, `PlainSalad`.
3. A `SaladDecorator` helps add more toppings to the `PlainSalad`.
4. `Cumcumber` inherts from `SaladDecorator`.

5. `Carrot` inherts from `SaladDecorator`.

By using the `SaladDecorator` class, you can extend your salad easily without having to change `PlainSalad`. You can also remove or add any salad decorator on runtime. Here's how you use it:

```
val cucumberSalad = Cucumber(Carrot(PlainSalad()))

print(cucumberSalad.getIngredient()) // Arugula & Lettuce, Carrot, Cucumberval carrotSalad = Carrot(PlainSalad())

print(carrotSalad.getIngredient()) // Arugula & Lettuce, Carrot
```

Composite

You use the **Composite** pattern when you want to represent a tree-like structure consisting of uniform objects. A Composite pattern can have two types of objects: composite and leaf. A composite object can have further objects, whereas a leaf object is the last object.

Take a look at the following code to understand it better:

```
//1interface Entity {

  fun getEntityName(): String

}

//2class Team(private val name: String) : Entity {

  override fun getEntityName(): String {

    return name
```

```
  }
}
//3class Raywenderlich(private val name: String) : Entity {
  private val teamList = arrayListOf<Entity>()

  override fun getEntityName(): String {
    return name + ", " + teamList.map { it.getEntityName() }.joinToString(", ")
  }

  fun addTeamMember(member: Entity) {
    teamList.add(member)
  }
}
```

In the code above you have:

1. `Component`, an interface `Entity` in Composite pattern.
2. A `Team` class implements an Entity. It's a `Leaf`.
3. `Raywenderlich` also implements an Entity interface. It's a `Composite`.

Logically and technically the organization, in this case `Raywenderlich`, adds an Entity to the Team. Here's how you use it:

```
val composite = Raywenderlich("Ray")val ericTeamComposite = Raywenderlich("Eric")val aaqib = Team("Aaqib")val vijay = Team("Vijay")

ericTeamComposite.addTeamMember(aaqib)

ericTeamComposite.addTeamMember(vijay)

composite.addTeamMember(ericTeamComposite)

print(composite.getEntityName()) // Ray, Eric, Aaqib, Vijay
```

Behavioral Patterns

"So... how do I tell which class is responsible for what?" – Future You

Behavioral Patterns let you assign responsibility for different app functions. Future You can use them to navigate the project's structure and architecture.

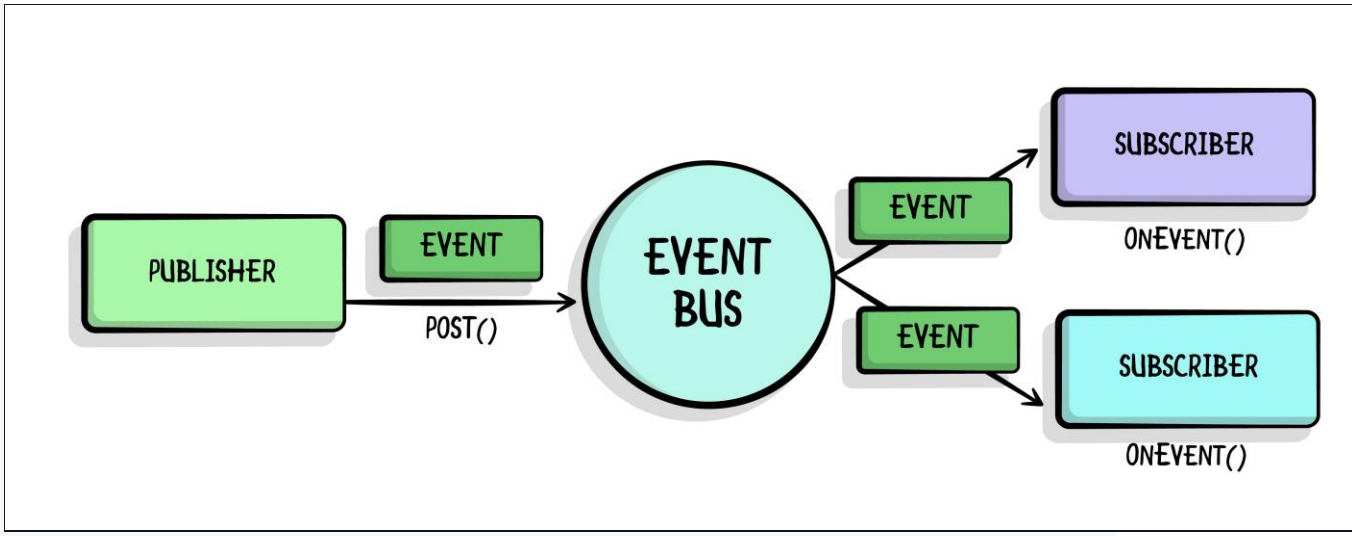
These patterns can vary in scope, from the relationship between two objects to your app's entire architecture. Often, developers use several behavioral patterns together in the same app.

Command

When you order some Saag Paneer at an Indian restaurant, you don't know which cook will prepare your dish. You just give your order to the waiter, who posts the order in the kitchen for the next available cook.

Similarly, the **Command** pattern lets you issue requests without knowing the receiver. You encapsulate a request as an object and send it off. Deciding how to complete the request is an entirely separate mechanism.

Greenrobot's EventBus is a popular Android framework that supports this pattern in the following manner:



An `Event` is a command-style object that's triggered by user input, server data or pretty much anything else in your app. You can create specific subclasses which carry data as well:

```
class MySpecificEvent { /* Additional fields if needed */ }
```

After defining your event, you obtain an instance of `EventBus` and register an object as a subscriber. For example, if you register an Activity you'll have:

```
override fun onStart() {
  super.onStart()
  EventBus.getDefault().register(this)
}
override fun onStop() {
  super.onStop()
  EventBus.getDefault().unregister(this)
```

```
}
```

Now that the object is a subscriber, tell it what type of event to subscribe to and what it should do when it receives one:

```
@Subscribe(threadMode = ThreadMode.MAIN)fun onEvent(event: MySpecificEvent?) {

  /* Do something */

}
```

Finally, create and post one of those events based on your criteria:

```
EventBus.getDefault().post(MySpecificEvent())
```

Since so much of this pattern works its magic at run-time, Future You might have a little trouble tracing this pattern unless you have good test coverage. Still, a well-designed flow of commands balances out the readability and should be easy to follow later.

Observer

The **Observer** pattern defines a one-to-many dependency between objects. When one object changes state, its dependents get a notification and updates automatically.

This pattern is versatile. You can use it for operations of indeterminate time, such as API calls. You can also use it to respond to user input.

It was originally popularized by the RxAndroid framework, also known as **Reactive Android**. This library lets you implement this pattern throughout your app:

```
apiService.getData(someData)
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())
 .subscribe (/* an Observer */)
```

In short, you define `Observable` objects that will *emit* values. These values can emit all at once, as a continuous stream or at any rate and duration.

`Subscriber` objects will *listen* for these values and react to them as they arrive. For example, you can open a subscription when you make an API call, listen to the server's response and react accordingly.

More recently Android also introduced a native way to implement this pattern through LiveData. You can learn more about this topic here.

Strategy

You use a **Strategy** pattern when you have multiple objects of the same nature with different functionalities. For a better understanding, take a look at the following code:

```
// 1interface TransportTypeStrategy {
 fun travelMode(): String
}
```

```
// 2class Car : TransportTypeStrategy {
  override fun travelMode(): String {
    return "Road"
  }
}
class Ship : TransportTypeStrategy {
  override fun travelMode(): String {
    return "Sea"
  }
}
class Aeroplane : TransportTypeStrategy {
  override fun travelMode(): String {
    return "Air"
  }
}
// 3class TravellingClient(var strategy: TransportTypeStrategy) {
  fun update(strategy: TransportTypeStrategy) {
    this.strategy = strategy
  }
```

```
  fun howToTravel(): String {
    return "Travel by ${strategy.travelMode()}"
  }
}
```

Here's a code breakdown:

1. A `TransportTypeStrategy` interface has a common type for other strategies so it can be interchanged at runtime.
2. All the concrete classes conform to `TransportTypeStrategy`.
3. `TravellingClient` composes strategy and uses its functionalities inside the functions exposed to the client side.

Here's how you use it:

```
val travelClient = TravellingClient(Aeroplane())
print(travelClient.howToTravel()) // Travel by Air// Change the Strategy to Ship
travelClient.update(Ship())
print(travelClient.howToTravel()) // Travel by Sea
```

State

In the **State** pattern, the state of an object alters its behavior accordingly when the internal state of the object changes. Take a look at the following snippets:

```
// 1interface PrinterState {
  fun print()
}
// 2class Ready : PrinterState {
  override fun print() {
    print("Printed Successfully.")
  }
}
// 3class NoInk : PrinterState {
  override fun print() {
    print("Printer doesn't have ink.")
  }
}
// 4class Printer() {
  private val noInk = NoInk()
  private val ready = Ready()
  private var state: PrinterState
  private var ink = 2
```

```
init {
 state = ready
}

private fun setPrinterState(state: PrinterState) {
 this.state = state
}

// 5
fun startPrinting() {
 ink--
 if (ink >= 0) {
  setPrinterState(ready)
 } else {
  setPrinterState(noInk)
 }
 state.print()
}
```

```
  // 6
  fun installInk() {
    ink = 2
    print("Ink installed.")
  }
}
```

Here’s a code breakdown:

1. `PrinterState` defines the states of a printer.
2. `Ready` is a concrete class implementing `PrinterState` to define a ready state of the printer.
3. `NoInk` is a concrete class implementing `PrinterState` to define that the printer has no ink.
4. `Printer` handler does all the printing.
5. `startPrinting` starts printing.
6. `installInk` installs ink.

Here’s how you use it:

```
val printing = Printer()
printing.startPrinting() // Printed Successfully.
printing.startPrinting() // Printed Successfully.
printing.startPrinting() // Printer doesn't have ink.
```

```
printing.installInk() // Ink installed.

printing.startPrinting() // Printed Successfully.
```