Using Feature Flags to Avoid External Interruptions

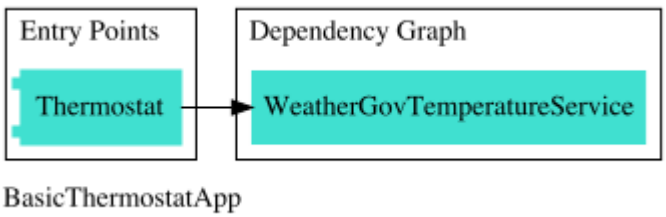
Most applications include code details to integrate with external systems. These systems, especially in development environments, can be unreliable. The following describes how feature flags can be used to temporarily swap out external details and avoid interruptions.

Let's imagine a `Thermostat` application that, among other things, determines if the outside temperature is between two values:

```
class Thermostat(private val temperatureService: TemperatureService) {
    fun isBetween(minTemp: Int, maxTemp: Int): Boolean {
        val currentTemp = temperatureService.get()
        return currentTemp > minTemp && currentTemp < maxTemp
    }
}

interface TemperatureService {
    fun get(): Int
}
```

We could implement the `TemperatureService` with a version that calls a weather.gov URL and parses the response data:

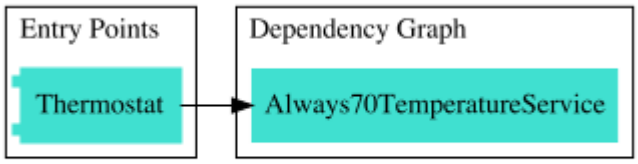
```
class WeatherGovTemperatureService: TemperatureService {
    override fun get() =
        HourlyForecast
            .from("https://api.weather.gov/gridpoints/TOP/31,80/forecast/hourly")
            .currentTemperature()
}

fun createBasicThermostatApp() = Thermostat(WeatherGovTemperatureService())
```



Arrows indicate creational dependencies. They point from an object to the objects used to create it

Or we could create a version that fakes a `TemperatureService` and always returns 70 degrees:

```
class Always70TemperatureService: TemperatureService {
    override fun get() = 70
}

fun createFakeThermostatAppWithFakes() = Thermostat(Always70TemperatureService())
```

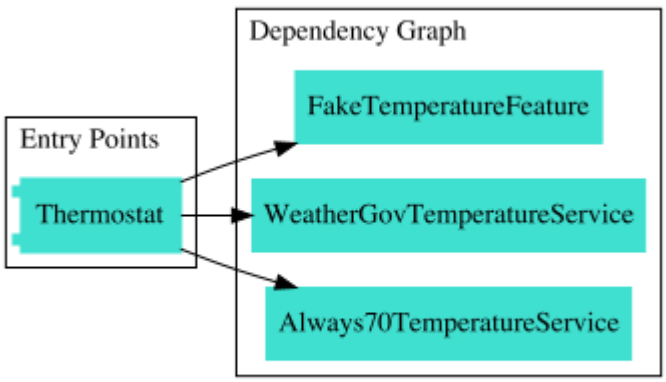


FakeThermostatApp

In either case, the `Thermostat` only depends on the `TemperatureService` interface and does not need to be changed to use different implementations. This concept is called dependency inversion and helps loosen the coupling between related objects in an application.

With two versions of `TemperatureService`, we can use a feature flag to control which version the app will use:

```
interface FakeTemperatureFeature {
    val isEnabled : Boolean
}

fun createThermostatAppWithFlags(): Thermostat {
    val feature =
        object: FakeTemperatureFeature { override val isEnabled = true }
    return Thermostat(
        if (feature.isEnabled) Always70TemperatureService()
        else WeatherGovTemperatureService()
    )
}
```

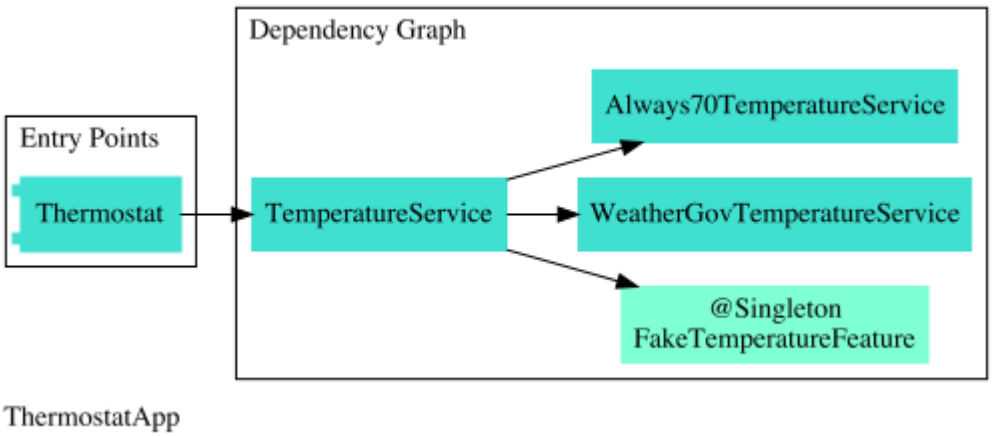


ThermostatAppWithFlags

Switching the state of `FakeTemperatureFeature.isEnabled` is now the only change needed to create a `Thermostat` with either version of the `TemperatureService`.

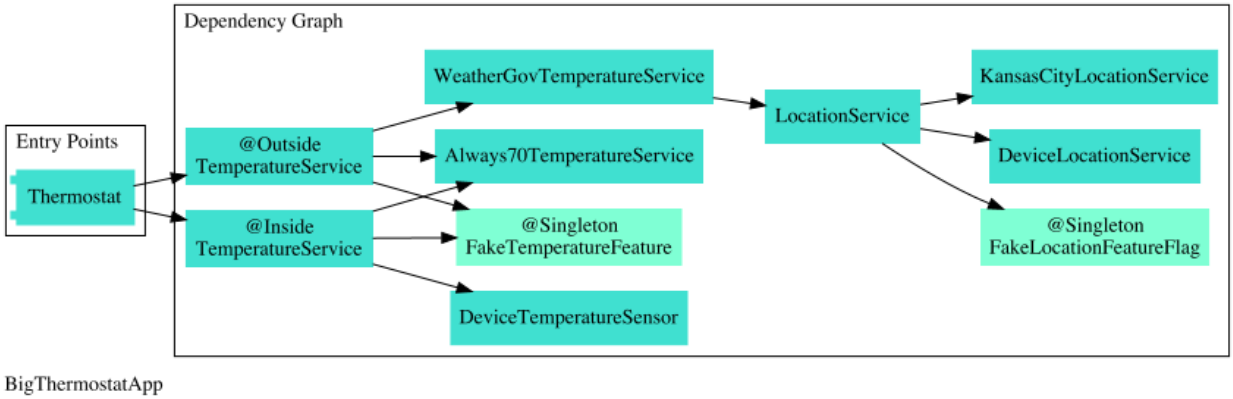
So far we have used simple factory functions to isolate the dependencies needed to *create* a `Thermostat`. But as the application grows in complexity, so can the the effort to manage these relationships. A dependency injection framework, like Dagger, can greatly reduce factory boilerplate and validate the object graph each time it is compiled:

```
@Module
class ThermostatModule {
    @Provides fun thermostat(service: TemperatureService) = Thermostat(service)
    @Provides fun nationalWeatherTempService() = WeatherGovTemperatureService()
    @Provides fun always70TempService() = Always70TemperatureService()
    @Provides @Singleton fun fakeTemperatureFeature() =
        object: FakeTemperatureFeature { override val isEnabled = true }
    @Provides fun temperatureService(
        feature: FakeTemperatureFeature,
        real: Provider<WeatherGovTemperatureService>,
        fake: Provider<Always70TemperatureService>
    ) = if (feature.isEnabled) fake.get() else real.get()
}

@Component(modules = [ThermostatModule::class]) @Singleton
interface ThermostatApp {
    fun thermostat(): Thermostat
}

fun createThermostatApp() = DaggerThermostatApp.create().thermostat()
```



The @Singleton annotation binds created objects to the scope of the @Component so the same instance can be reused

A future version of the dependency graph, with additional feature and flags, might look like this:



Finally, making features easy to switch can be useful in development, but we don't want those changes to be accidentally released. We can prevent this with automated tests:

```
fun testThatFakeTemperatureFeatureIsFalse() {
    val feature = ThermostatModule().fakeTemperatureFeature()
    assert(!feature.isEnabled) {
        "$feature.isEnabled should be false but was ${feature.isEnabled}"
    }
}
```

Summary

1. Invert dependencies on external details
2. Write a fake version of the external details
3. Use a feature flag to declare which version to use
4. Isolate feature decisions in factories
5. Use injection to minimize creational boilerplate
6. Add automated tests to prevent releasing features accidentally

Additional Notes

Creating fake versions may seem like extra work, but if you're writing unit tests, you probably already have test doubles that cover what you need. In the example below, `always60` or `always70` are test doubles that could be swapped for a real `TemperatureService`.

```
fun testThatIsBetweenReturnsExpectedValues() {
    val always60 = object: TemperatureService { override fun get() = 60 }
    val always70 = object: TemperatureService { override fun get() = 70 }
    val minTemp = 68
    val maxTemp = 72
    assert(Thermostat(always60).isBetween(minTemp, maxTemp) == false)
    assert(Thermostat(always70).isBetween(minTemp, maxTemp) == true)
}
```

You don't need to make all integrations flaggable from the start. The next time you're blocked by problems with an outside system, try applying a flag for just the calls needed to unblock your current task. It should be possible to get part of an application working with object fakes even if the rest is not.

`FakeTemperatureFlag.isEnabled` has a boolean type because only two value states were necessary: true & false (feature flags with only two states are often called feature toggles). But there are situations where flags with more than two states may be suitable. We might want to test with a set of fakes that all throw errors and a set of fakes that all return stubbed values. A flag for this might have three states: real, stubs, and errors.