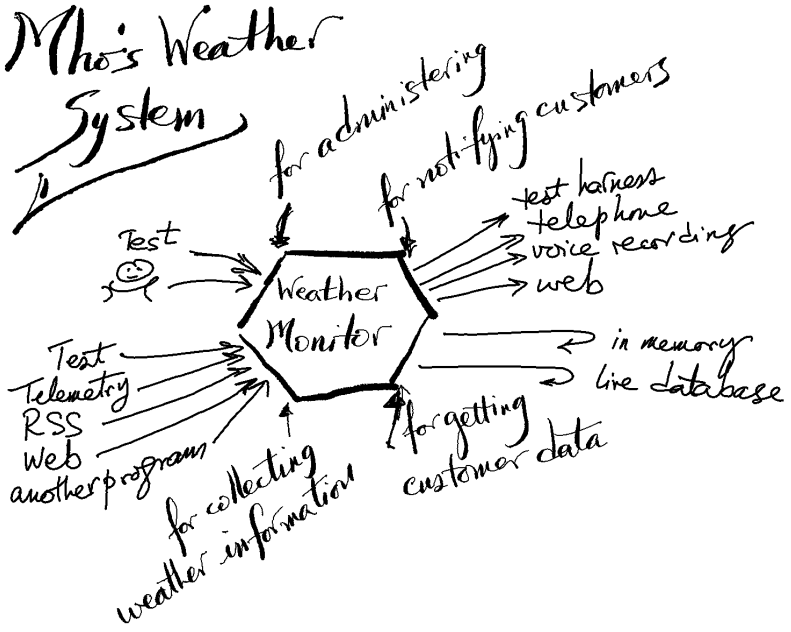
Hexagonal Architecture
(Ports & Adapters)

The 2023 version ☺

Alistair Cockburn



Outline

1. What's the point?
2. Chip component analogy
3. Development sequence
4. Hexagonal Example, w code (Ruby)
5. Required interfaces, w code (Java)
6. Hexagonal example, w code (Java)
7. Juan's Blue Zone example (Java)
8. Where do I put all the declarations & code?
9. The configurator
10. Costs and Benefits

What is the point?

Create your application to work without either a UI or a database
so you can run automated regression-tests against it,
work when the database becomes unavailable,
upgrade to new technology, and
link applications together.

Benefits

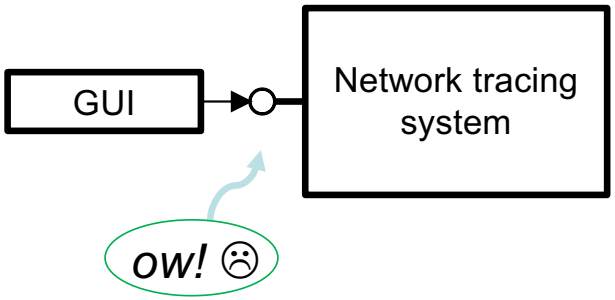
1. You get to decide the app's driven actors at initialization, over a period of years as technologies shift, or in real time.
2. You get to replace production connections with test harnesses, and back again, without changing the source code.
3. You get to avoid having to change the source code and then rebuild the system every time you make these shifts.
4. You can prevent leaks of business logic into the UI or data services, and vice versa, prevent leaks of UI or data service logic into the business logic.

Costs

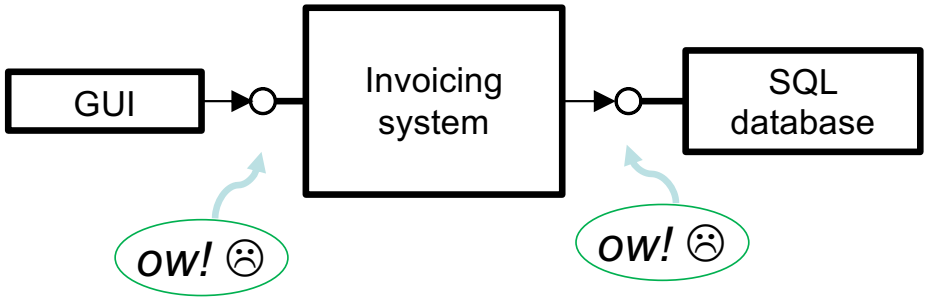
1. You must add an instance var to hold each driven actor, or get it every time.
2. You must add a constructor parameter or a setter function for each driven actor, or a call to the configurator to get it.
3. You must design and add a configurator.
4. *(Type-checked languages)* You must declare the *"required"* interfaces.
5. *(Type-checked languages)* You must add folder structure for the port declarations.

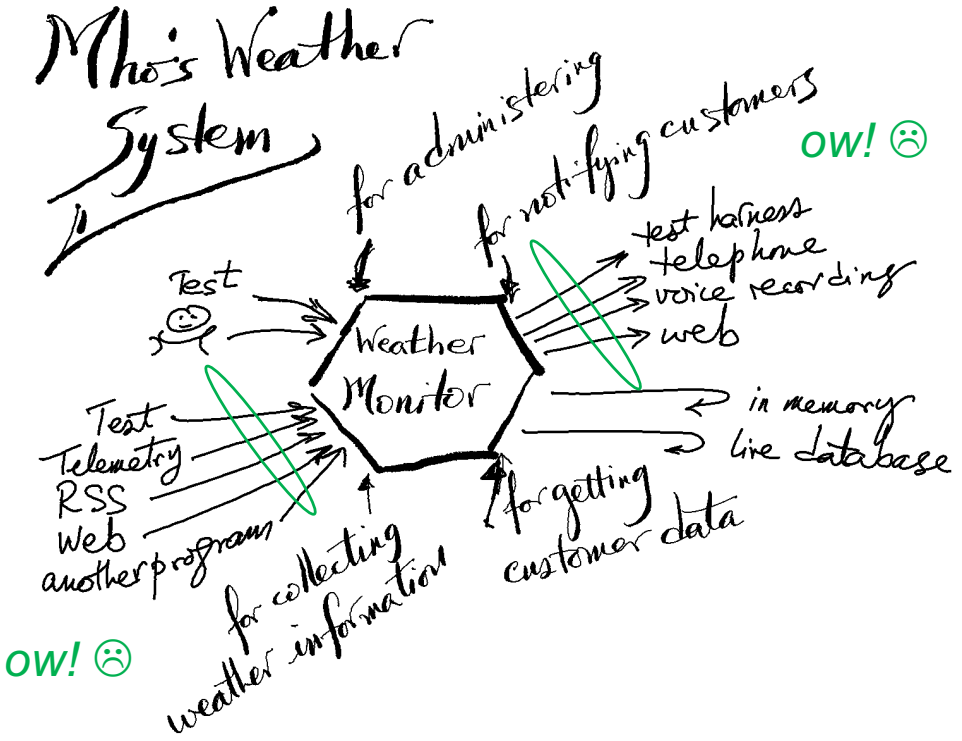
Why? When would it have been worth it?

Network tracing program

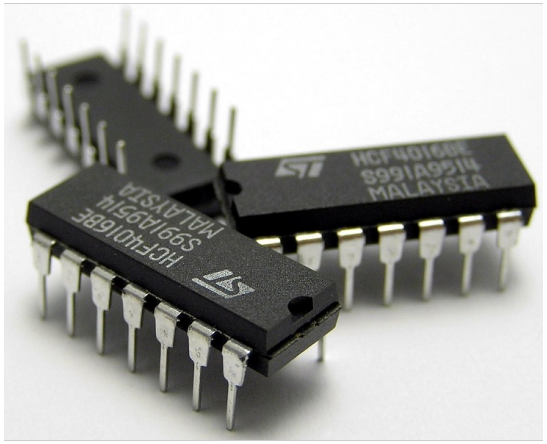


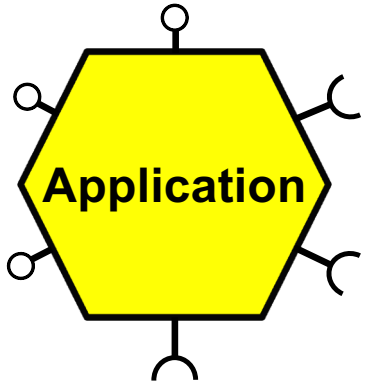
Invoicing system





The app is a “component”

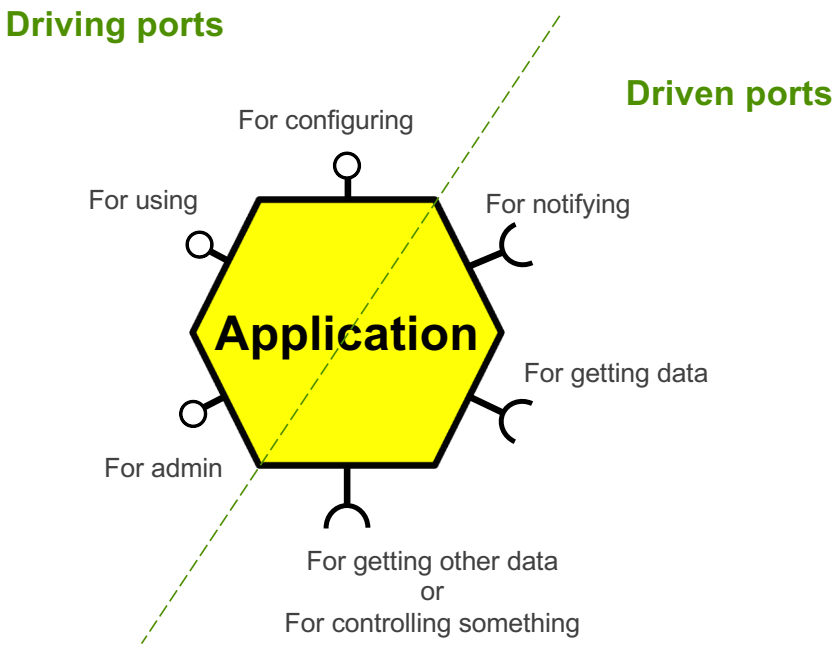




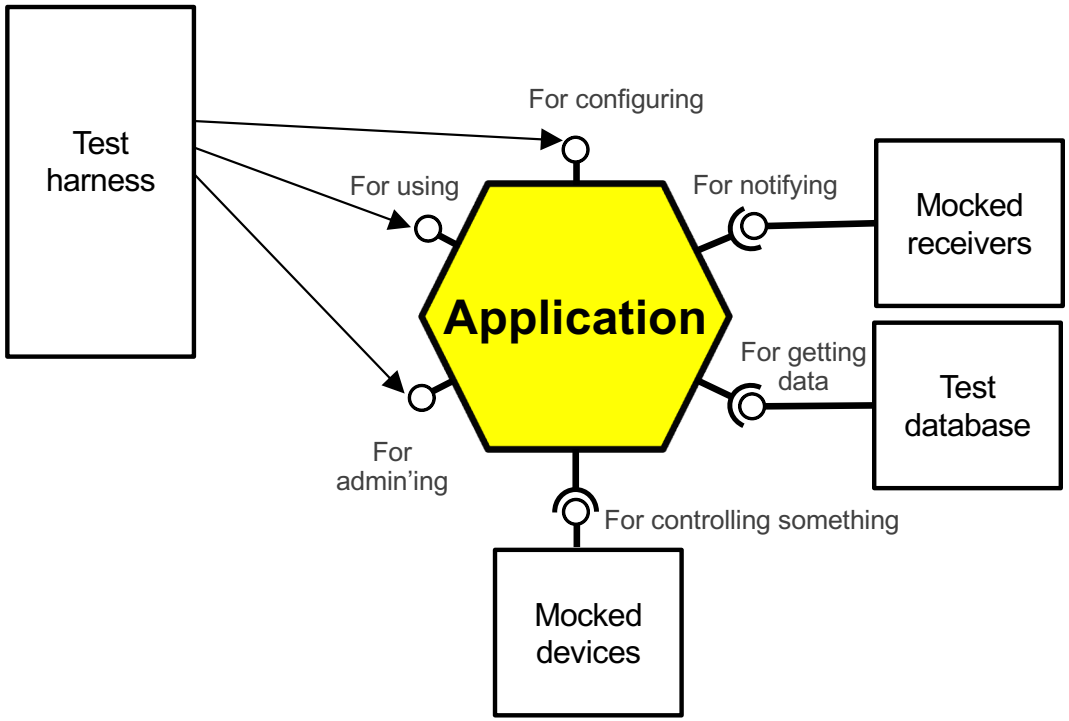
The app is a component, with ports

We name the ports for their purpose: "For_doing_something".

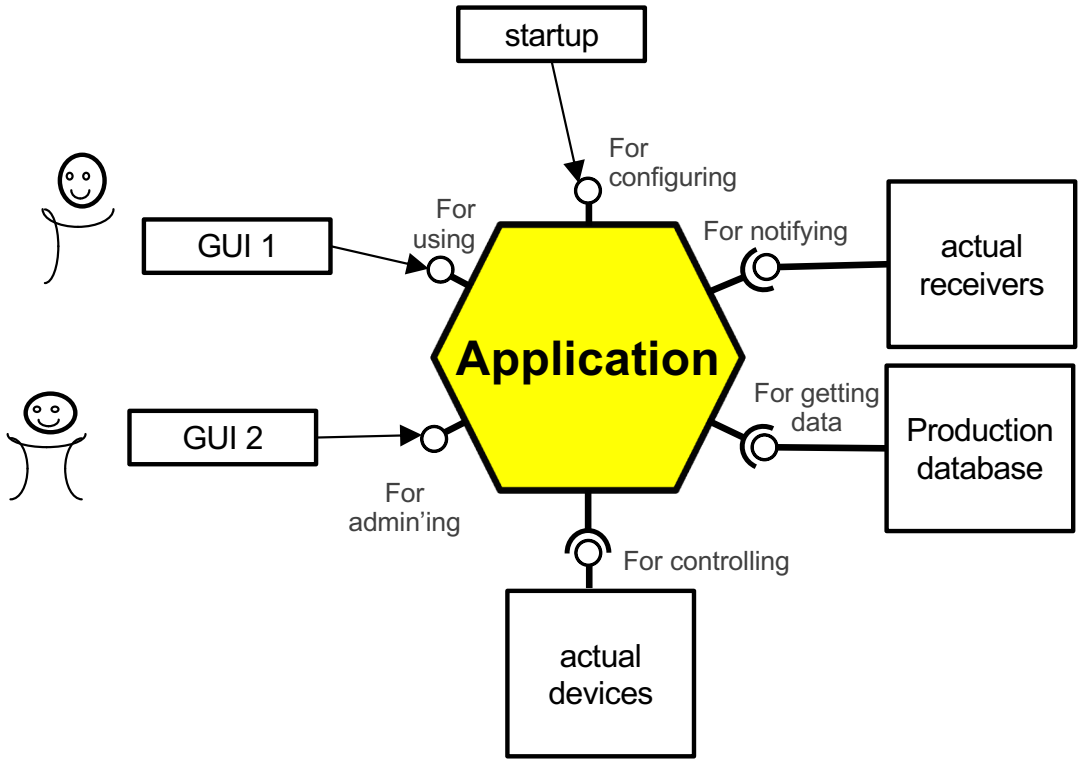
Each port can have multiple function calls.



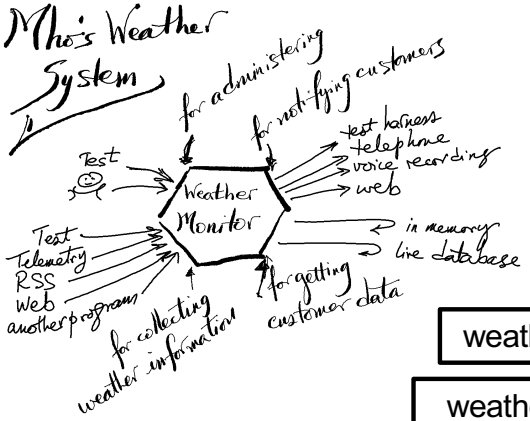
Hooking up the component for testing

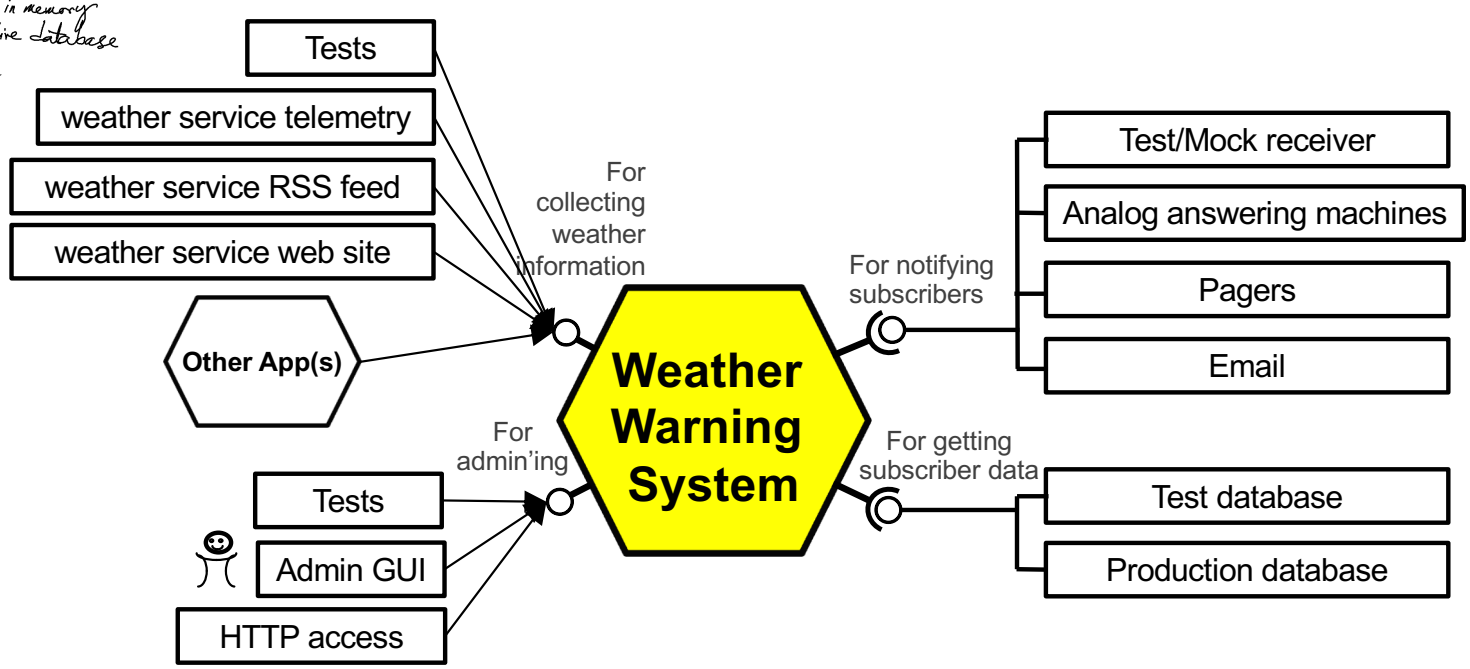


Hooking up the component for production

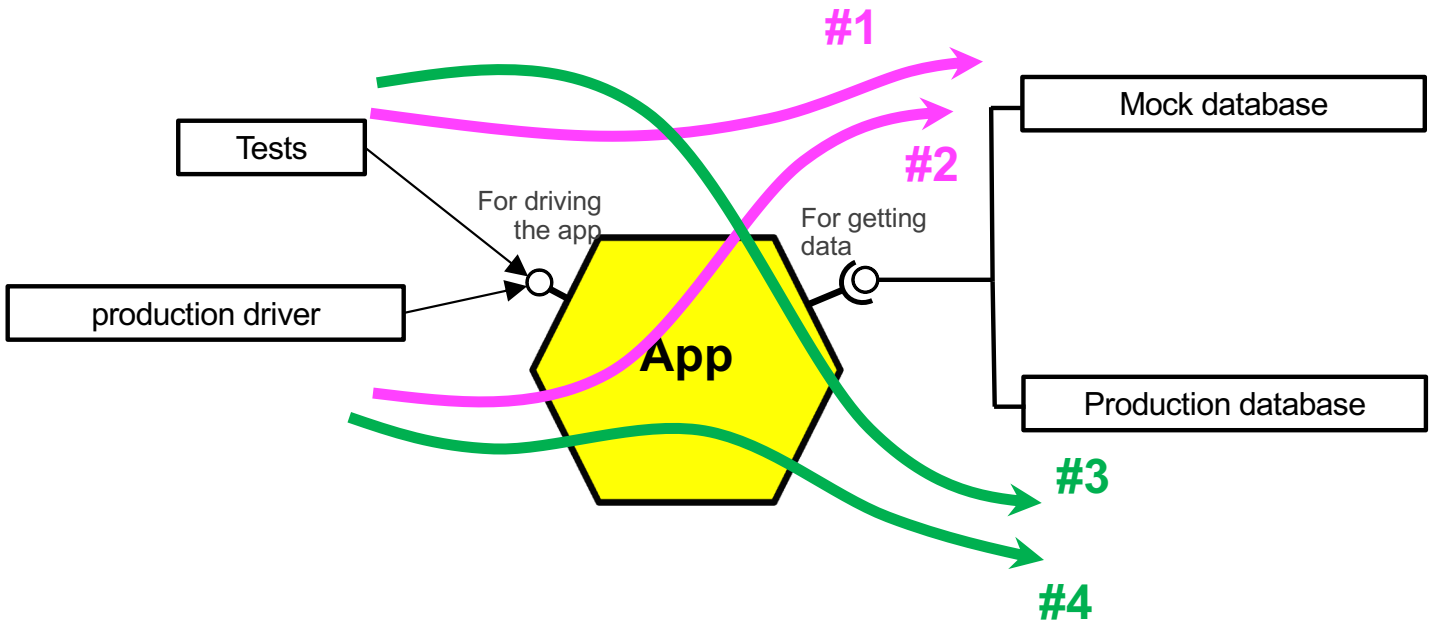


Mho's weather warning system as components

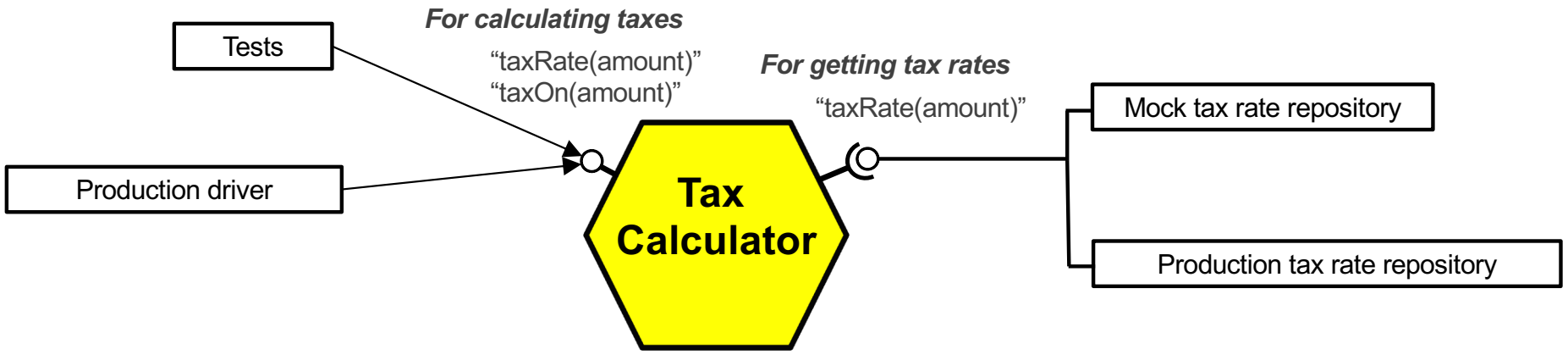




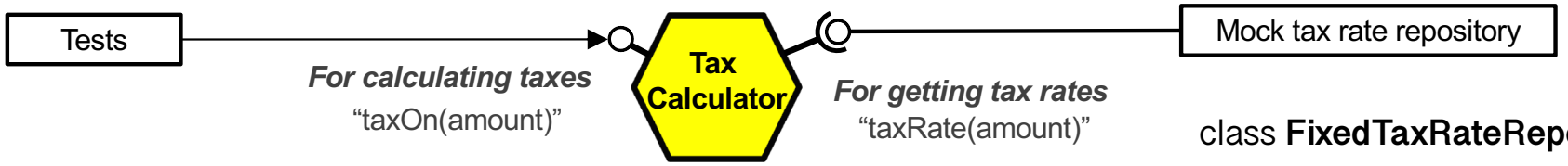
Development Sequence:
tests + mocks first



Simplest example: tax calculator



Code for Tax Calculator (Ruby)




```
class TaxCalculator
  def initialize( tax_rate_repository )
    @tax_rate_repository = tax_rate_repository
  end

  def tax_on( amount )
    amount * @tax_rate_repository.tax_rate( amount )
  end
end
```

```
class FixedTaxRateRepository
  def tax_rate( amount )
    0.15
  end
end
```

(Note: no interface declarations)

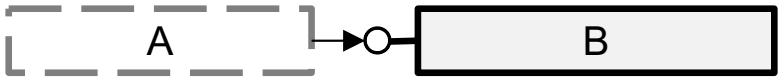
```
tax_rate_repository = FixedTaxRateRepository.new
my_calculator = TaxCalculator.new( tax_rate_repository )
puts my_calculator.tax_rate( 100 )
```

This is the work of the "configurator" (aka Composition Root)

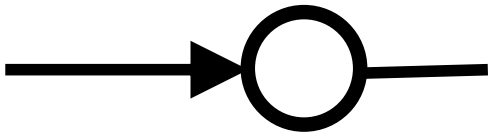
This is using the system at the port

Detour for type-checked languages: *Provided* & *Required* interfaces

B provides services.
A uses them.

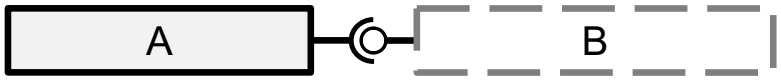


B "implements" the interface.

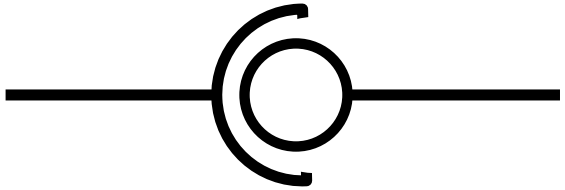


Useful for defining a public API

A "requires" this interface.
B implements it.



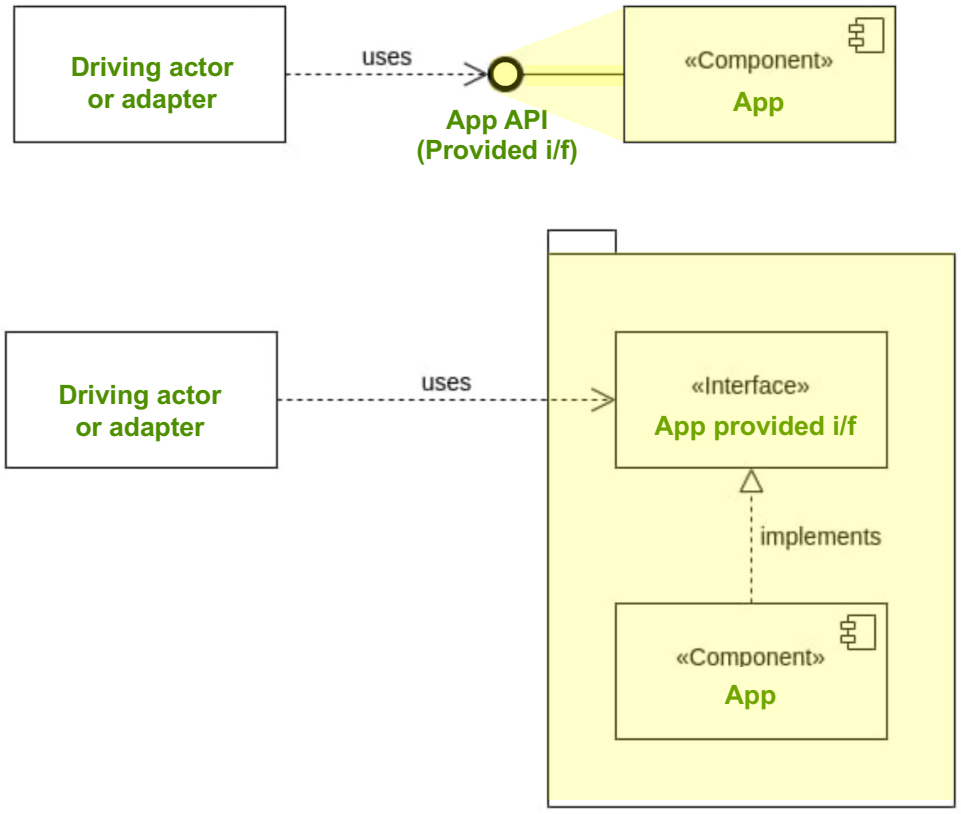
A owns the interface definition.



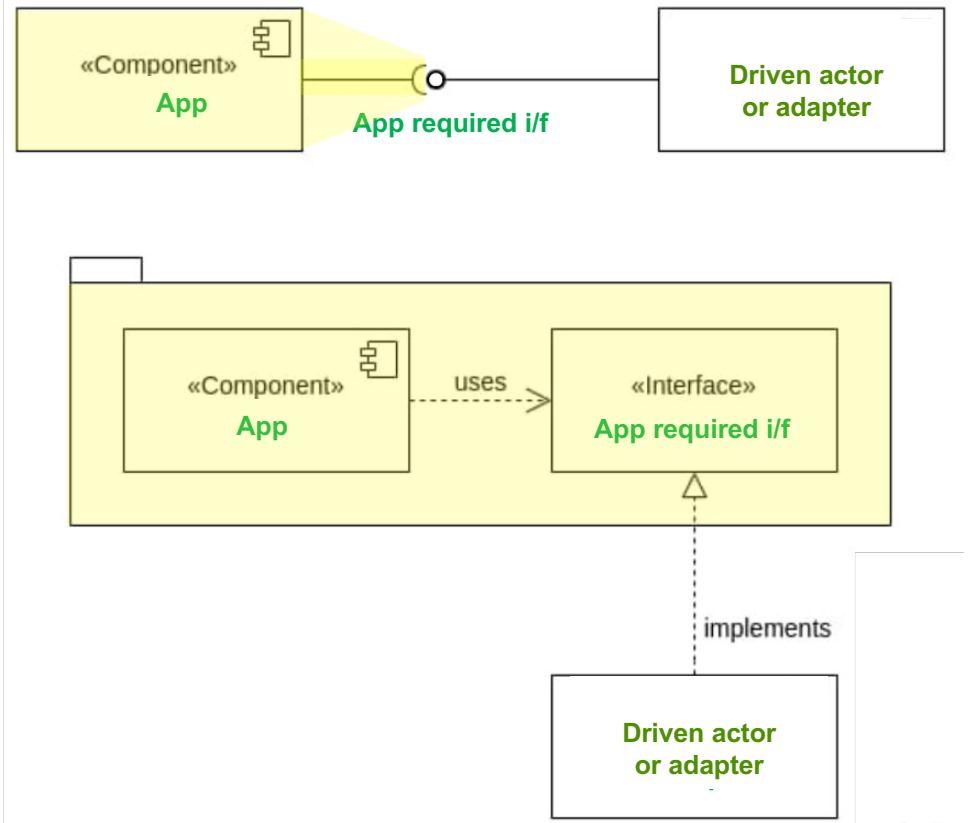
Useful for decoupling a component from its receivers

The source-code dependencies

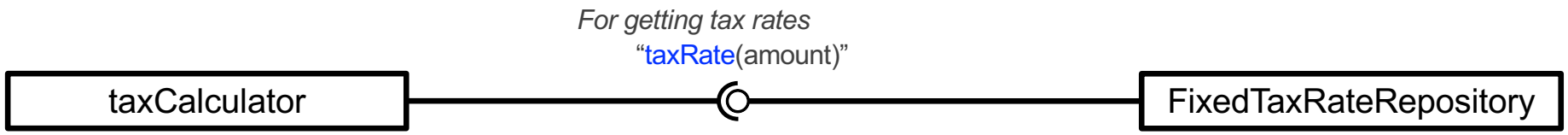
Driving ports: The app owns the interface



Driven ports: The app owns the interface



What a required interface looks like in code (Java)



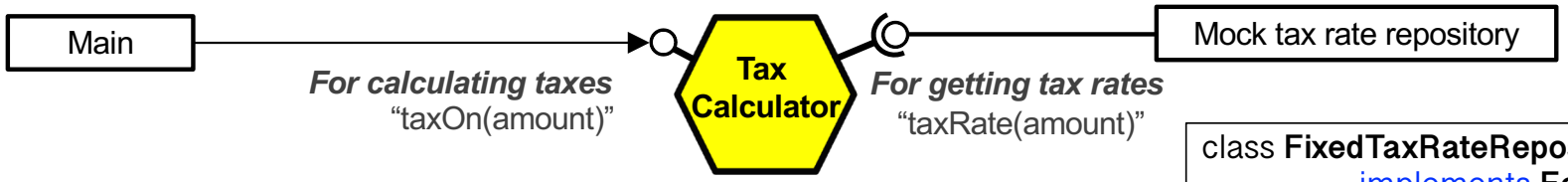

```
interface ForGettingTaxRates {
    double taxRate(double amount); }
```

Make sure the required interface's definition belongs to the calculator, not to the repository!

```
class TaxCalculator {
    private ForGettingTaxRates taxRateRepository;
    public TaxCalculator(ForGettingTaxRates taxRateRepository) {
        this.taxRateRepository = taxRateRepository;
    }
    public double taxOn(double amount) {
        return amount * taxRateRepository. taxRate( amount );
    }
}
```

```
class FixedTaxRateRepository
            implements ForGettingTaxRates {
    public double taxRate(double amount) {
            return 0.15; }
}
```


Code for Tax Calculator (Java)




```
interface ForCalculatingTaxes {
    double taxOn(double amount);
}
interface ForGettingTaxRates {
    double taxRate(double amount);
}

class TaxCalculator implements ForCalculatingTaxes {
    private ForGettingTaxRates taxRateRepository;

    public TaxCalculator(ForGettingTaxRates taxRateRepository) {
        this.taxRateRepository = taxRateRepository;
    }
    public double taxOn(double amount) {
        return amount *  taxRateRepository. taxRate( amount );
    }
}
```

```
class FixedTaxRateRepository
        implements ForGettingTaxRates {
    public double taxRate(double amount) {
        return 0.15;
    }
}
```

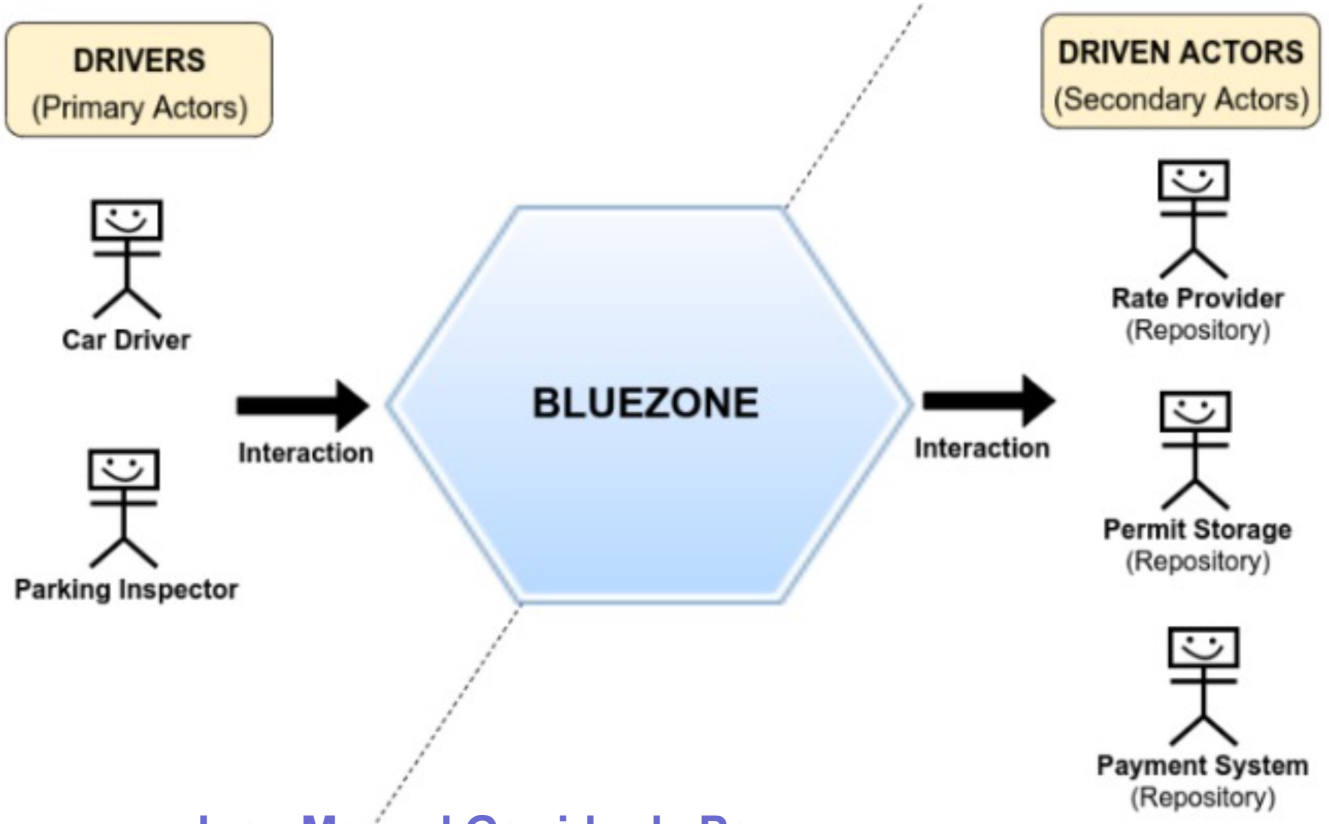
```
class Main {
    public static void main(String[] args) {
        ForGettingTaxRates taxRateRepository = new FixedTaxRateRepository();
        TaxCalculator myCalculator = new TaxCalculator( taxRateRepository );
        System.out.println( myCalculator.taxOn( 100 ) );
    }
```

the "configurator"

using the system at the port

A more complex example: Juan's "Blue Zone"

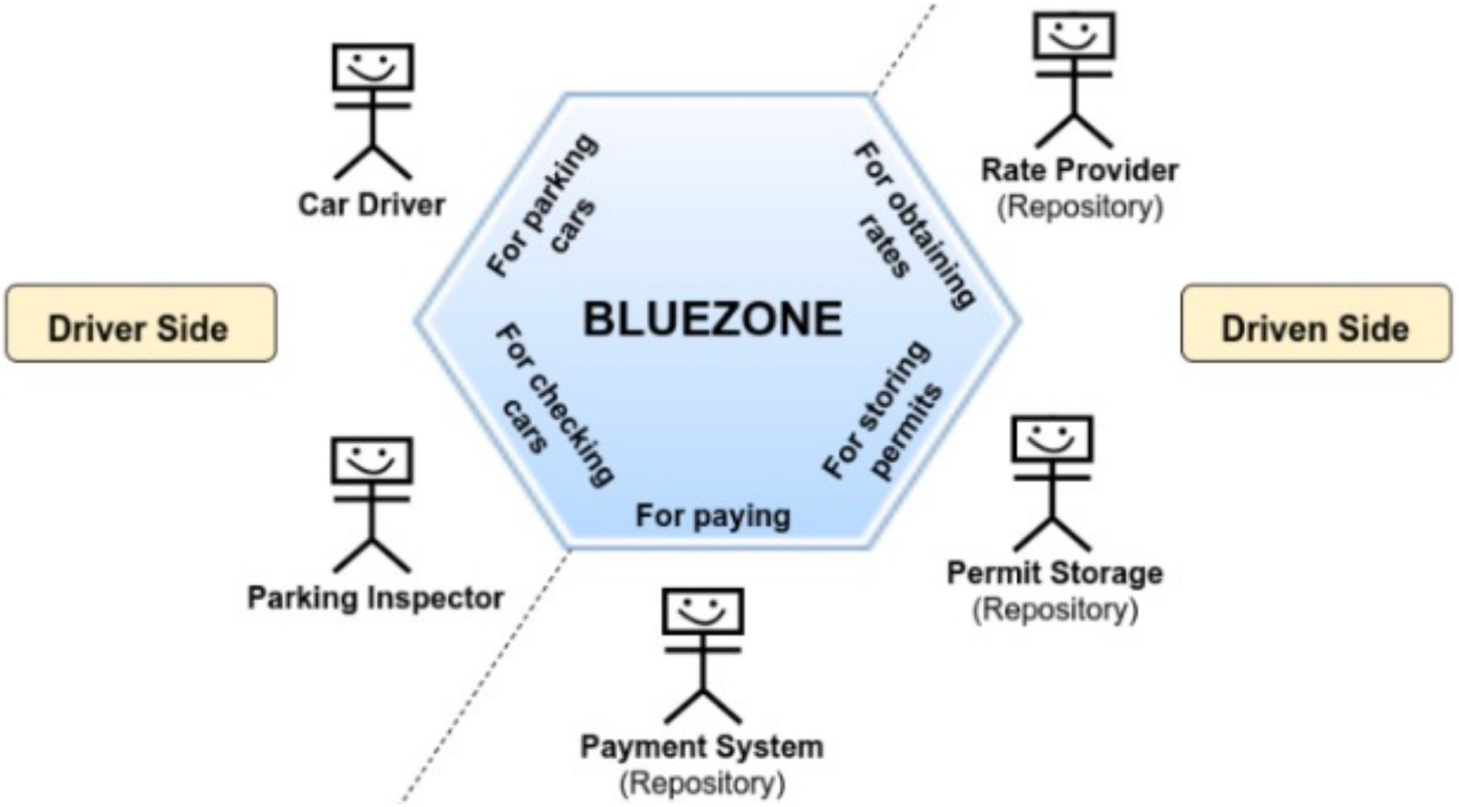
BlueZone allows car drivers to pay remotely for parking cars at zones in a city, instead of paying with coins using parking meters



Juan Manuel Garrido de Paz:
https://github.com/jmgarridopaz/bluezone

Juan's "Blue Zone" example

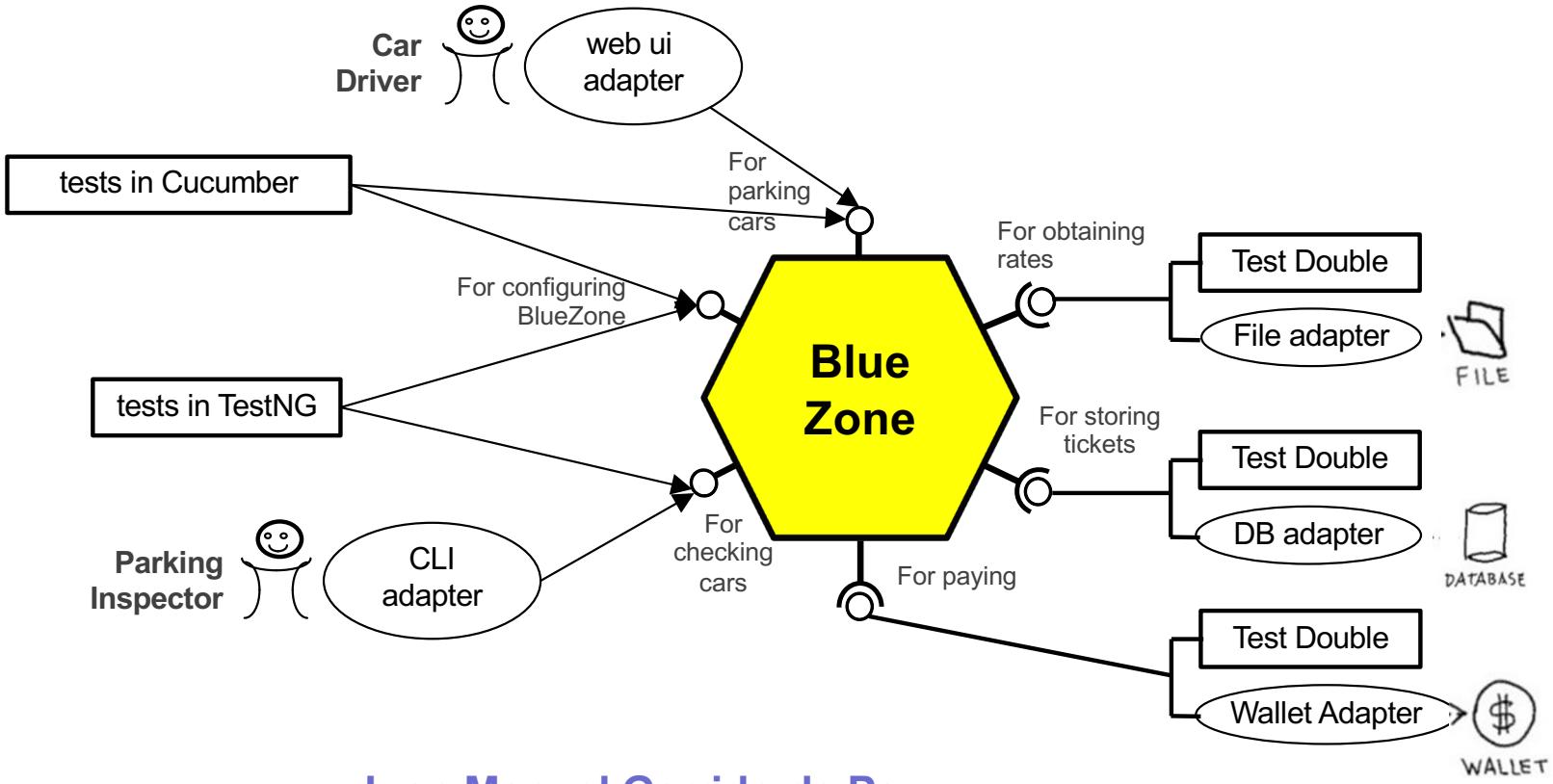
BlueZone allows car drivers to pay remotely for parking cars at zones in a city, instead of paying with coins using parking meters



Juan Manuel Garrido de Paz:
https://github.com/jmgarridopaz/bluezone

Juan's "Blue Zone" example

BlueZone allows car drivers to pay remotely for parking cars at zones in a city, instead of paying with coins using parking meters



Juan Manuel Garrido de Paz:
https://github.com/jmgarridopaz/bluezone

Juan's "Blue Zone" example

https://github.com/jmgarridopaz/bluezone

Ports

- Driven Ports
  - ForPaying.java
  - ForObtainingRates.java
  - ForStoringTickets.java
- Driving Ports
  - ForCheckingCars.java
  - ForConfiguringApp.java
  - ForParkingCars.java

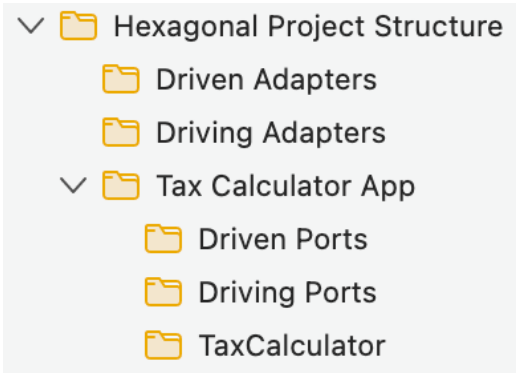
```java
/**
 * DRIVEN PORT
 */
public interface ForObtainingRates {
    public Set<Rate> findAll();
    public Rate findByName (String rateName );
    public void addRate ( Rate rate );
    public boolean exists ( String rateName );
    public void empty();
}
```

Adapters

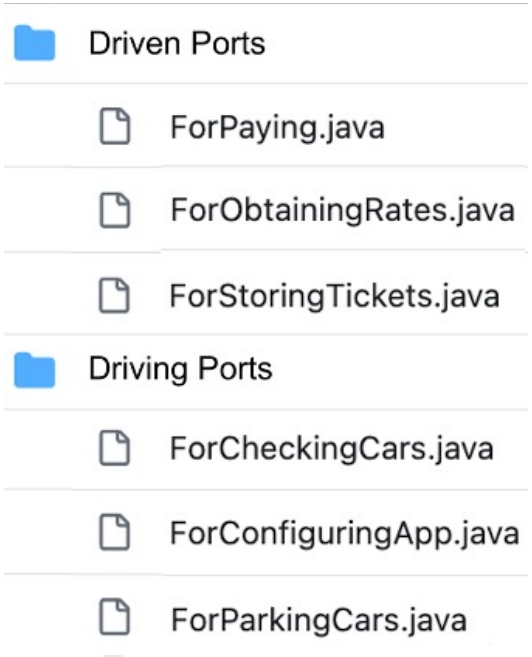
- Driven Adapters
  - bluezone-adapter-forpaying-spy
  - bluezone-adapter-forobtainingrates-stub
  - bluezone-adapter-forstoringtickets-fake
- Driving Adapters
  - bluezone-driver-forcheckingcars-test
  - bluezone-adapter-forparkingcars-webui
  - bluezone-driver-forparkingcars-test

```java
/**
 * Driven adapter that implements "forobtainingrates" port
   with a stub test double.
 */
@Adapter(name="test-double")
public class StubRateProviderAdapter implements ForObtainingRates {
        private Set<Rate> rates;
```

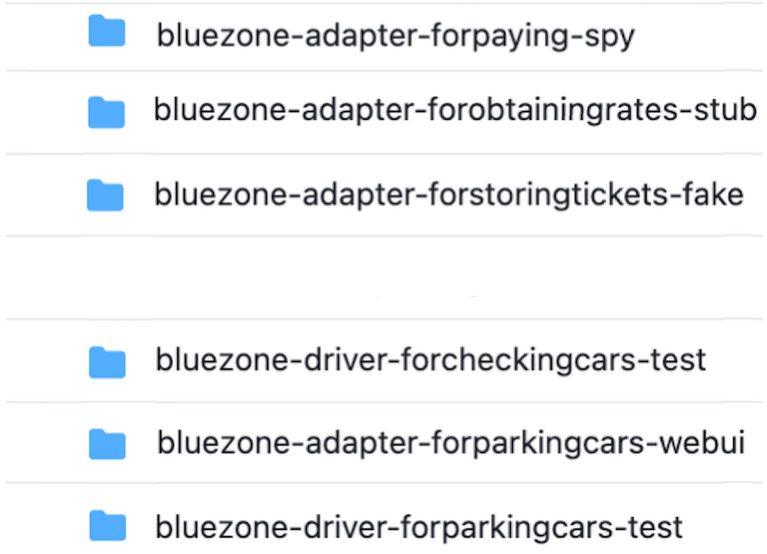

The folders: Port declarations and Adapters



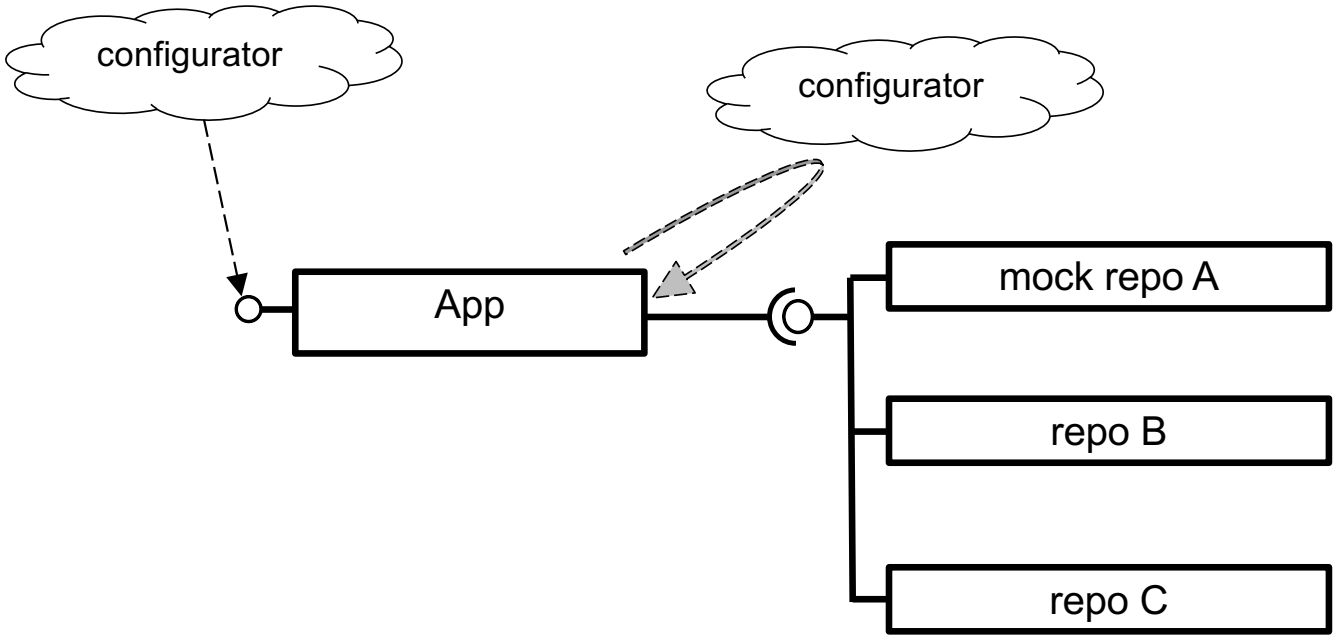
Port declaration folders in the app



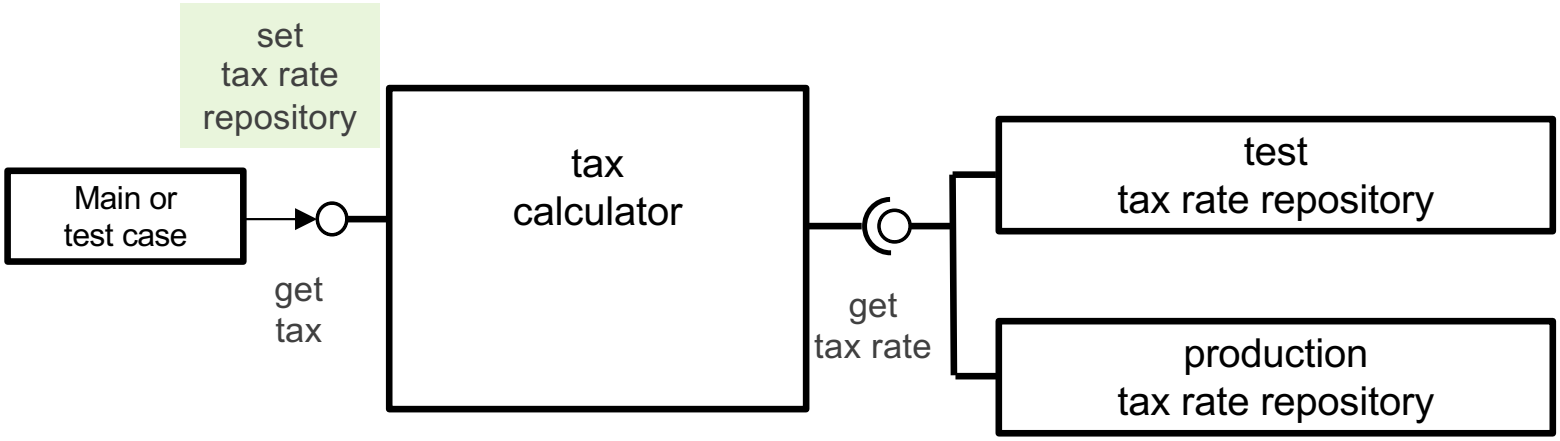
Adapter folders outside the app



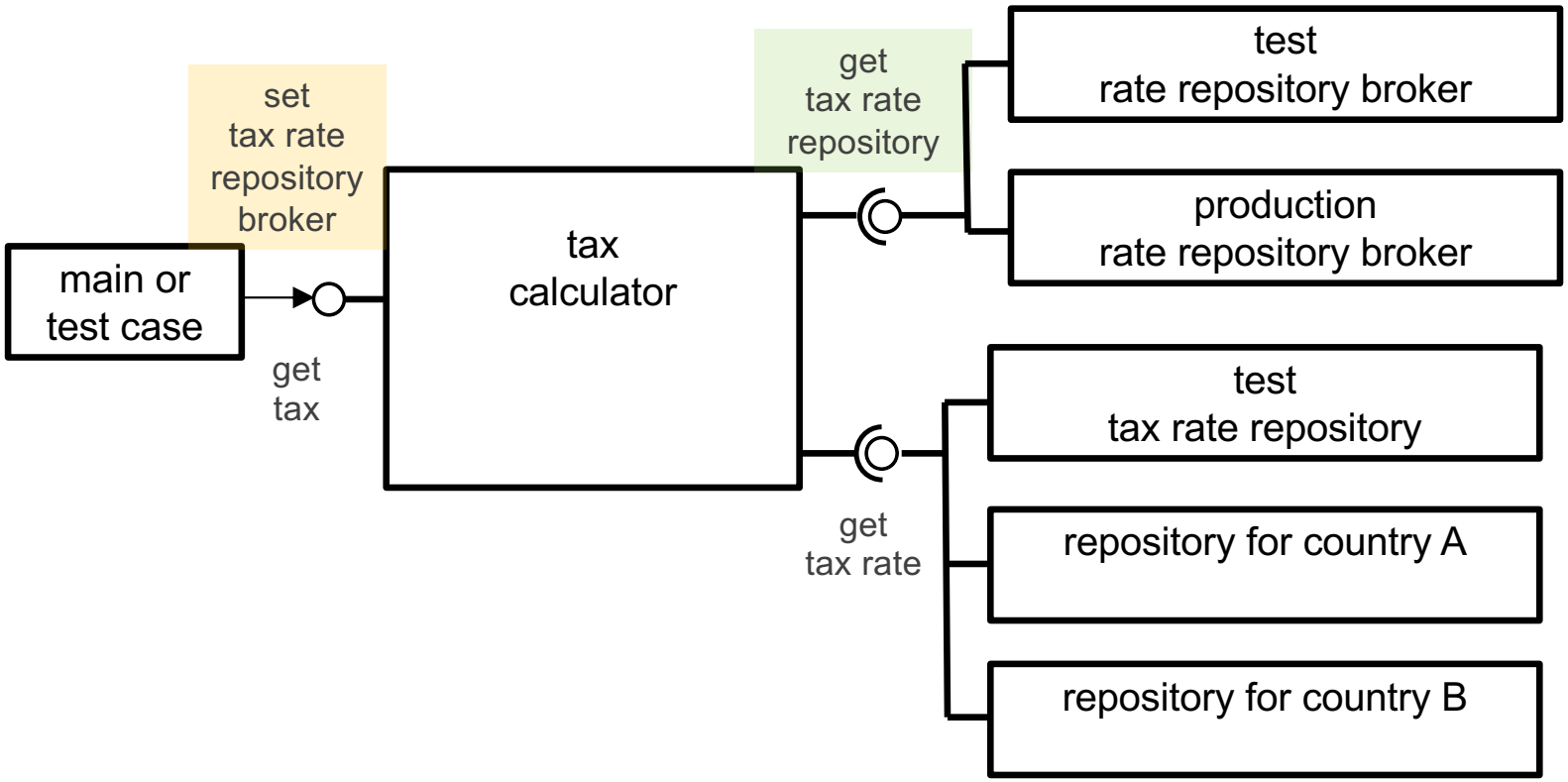
How do we design the configurator?



Configurator design #1: setter method
(Dependency Injection)



Configurator design #2: repository broker
(Dependency Lookup)



Benefits

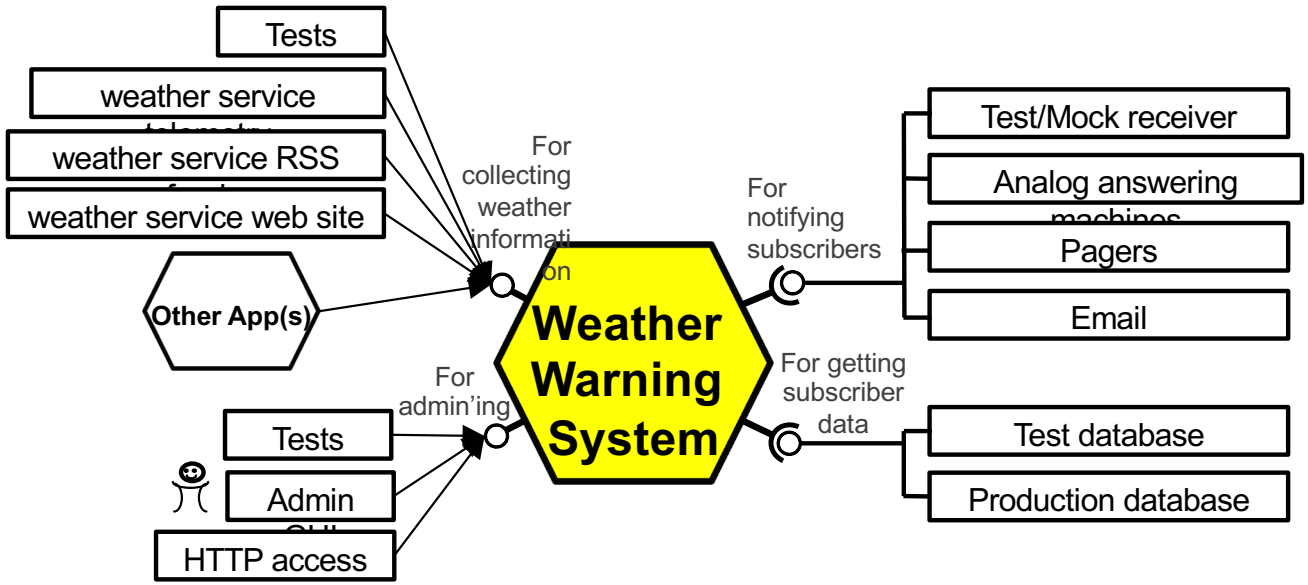
1. You get to set the app's driven actors during execution -- at initialization, over a period of years as technologies shift, or in real time.
2. You get to replace production connections with test harnesses, and back again, without changing the source code.
3. You get to avoid having to change the source code and then rebuild the system every time you make these shifts.
4. You can prevent leaks of business logic into the UI or data services, and vice versa, prevent leaks of UI or data service logic into the business logic.

Costs

1. You must add an instance var to hold each driven actor, or get it every time.
2. You must add a constructor parameter or a setter function for each driven actor, or a call to the configurator to get it.
3. You must design and add a configurator.
4. *(Type-checked languages)* You must declare the *"required"* interfaces.
5. *(Type-checked languages)* You must add folder structure for the port declarations.

Ports & Adapters Pattern
(aka Hexagonal Architecture)

Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, upgrade to new technology, and link applications together.



With many thanks to Juan Manuel Garrido de Paz for continual close reading, exactness, and code.