Software Architecture

Hexagonal Architecture

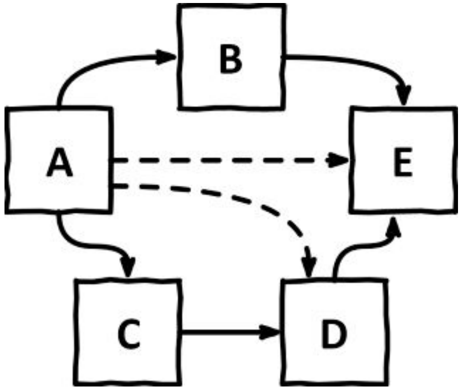
We’ll take advantage of SOLID principles…

- Single Responsibility Principle (SPR)
- Open Closed Principle (OCP)
- Interface segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

SRP: Single Responsibility Principle

A component should have only one reason to change

If a component has only one reason to change, we don't have to worry about this component at all if we change the software for any other reason, because we know that it will still work as expected

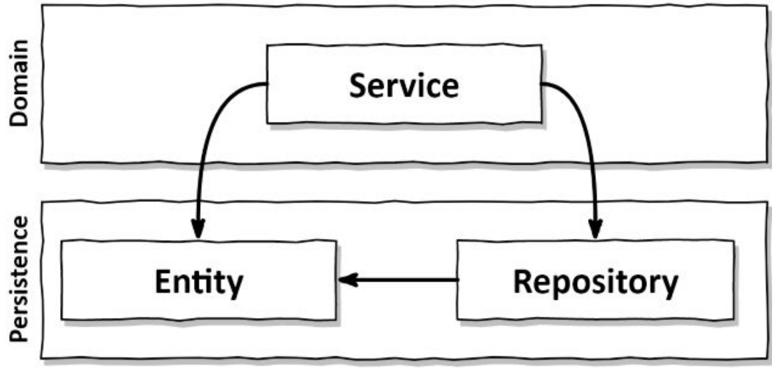


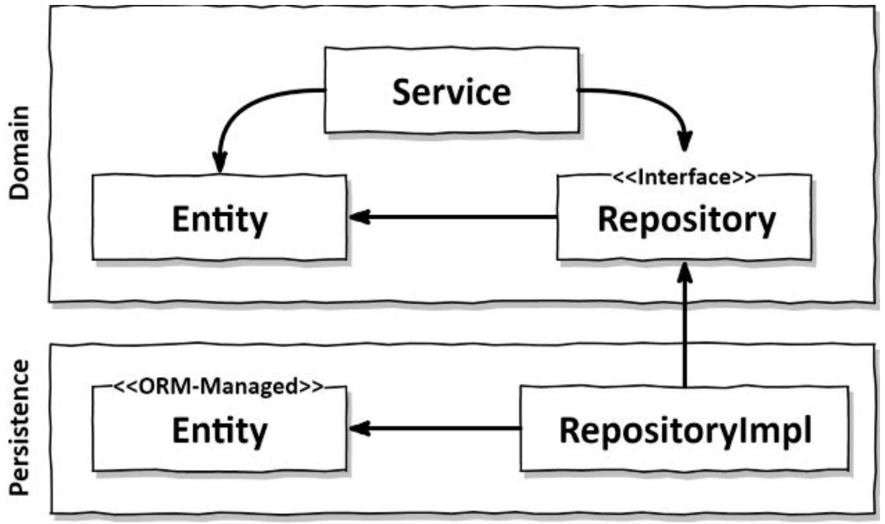
Dependency Inversion Principle

In a layered architecture, the cross-layer dependencies always point downward to the next layer. When we apply the Single Responsibility Principle on a high level, we notice that the upper layers have more reasons to change than the lower layers.

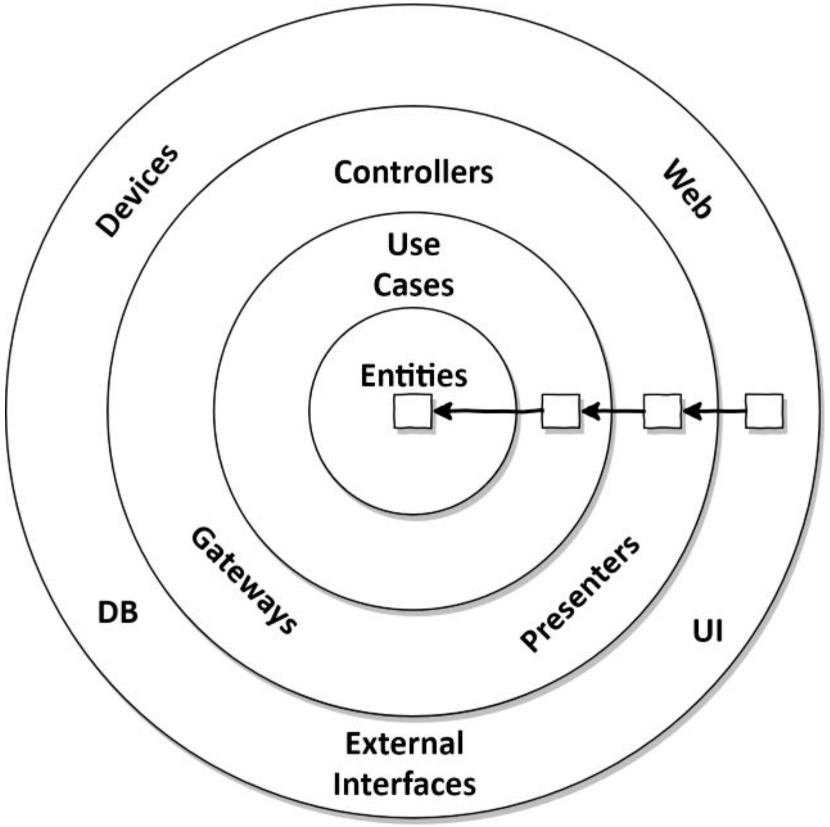
DIP: “We can turn around (invert) the direction of any dependency within our codebase”

Dependency Inversion Principle



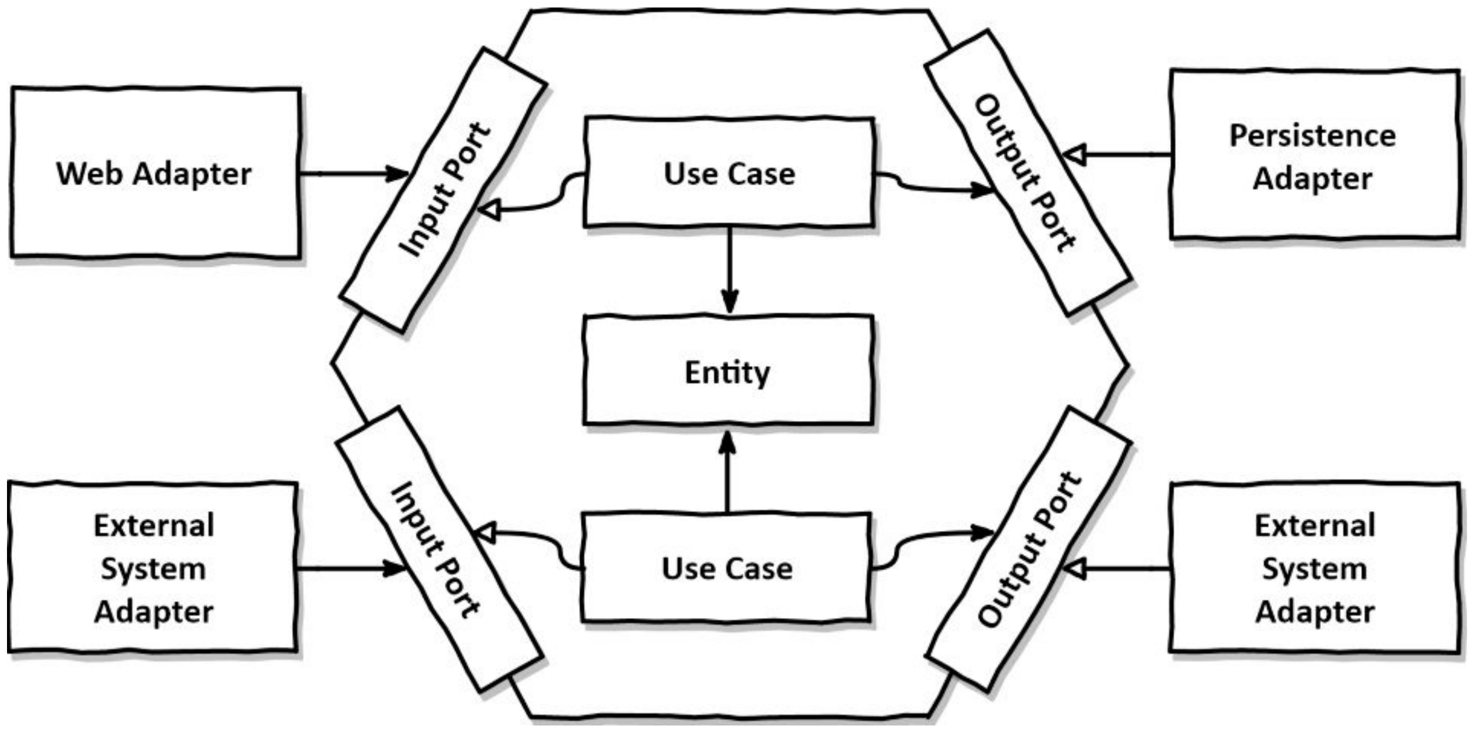


Clean Architecture



- Clean Architecture comes at a cost: we have to maintain a model of our application’s entities in each of the layers.
- This decoupling is exactly what we wanted to achieve to free the domain code from the details (frameworks, …)

Hexagonal Architecture: clean arch. made concrete



Implementing the Hexagonal Architecture

Organizing Code

Example: Buckpal

- Accounts
- Money
- Deposits
- Withdrawals
- Money transfers
- ...

First Try: Organizing by Layer

```
buckpal
├── domain
|   ├── Account
|   ├── Activity
|   ├── AccountRepository
|   └── AccountService
├── persistence
|   └── AccountRepositoryImpl
└── web
    └── AccountController
```

Problems:

1. We have no package boundary between functional slices or features of our application
2. We can’t see which use cases our application provides
3. We can’t see our target architecture within the package structure. The incoming and outgoing ports are hidden in the code

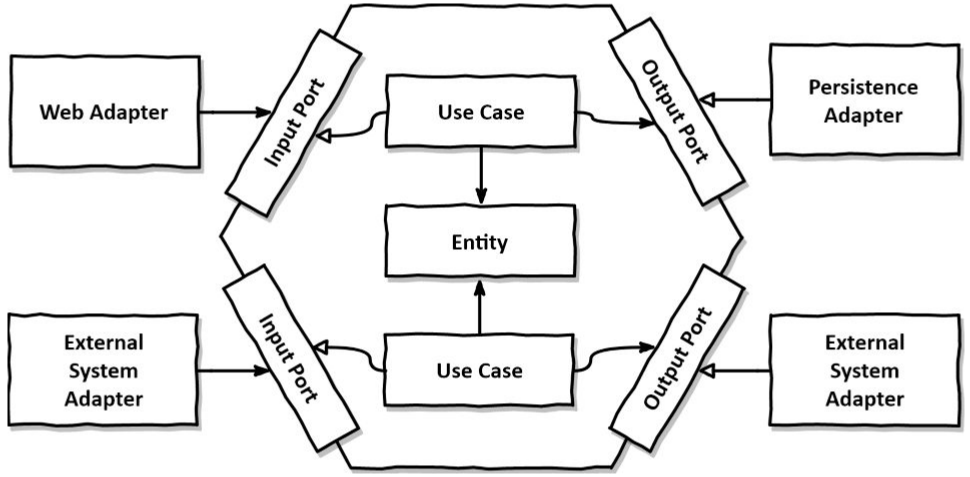
Second Try: Organizing by Feature

```
buckpal
└── account
    ├── Account
    ├── AccountController
    ├── AccountRepository
    ├── AccountRepositoryImpl
    └── SendMoneyService
```

- Each new group of features will get a new high-level package next to account and we can enforce package boundaries between the features by using package-private visibility for the classes that should not be accessed from the outside.
- We have also renamed AccountService to SendMoneyService to narrow its responsibility
- The package-by-feature approach makes our architecture even less visible. Domain and persistence code can easily become coupled

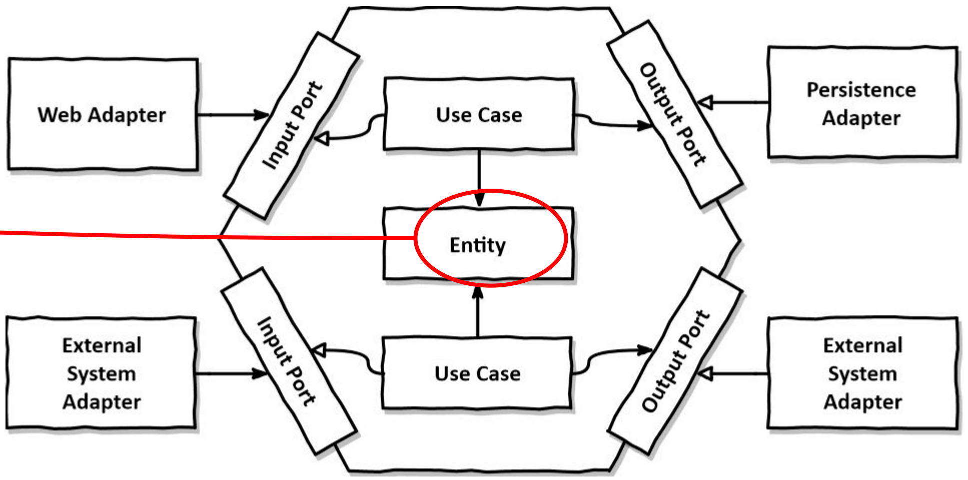
An Architecturally Expressive Package Structure

```
buckpal
 └── account
    ├── adapter
    |  ├── in
    |  |   └── web
    |  |       └── AccountController
    |  ├── out
    |  |   └── persistence
    |  |       ├── AccountPersistenceAdapter
    |  |       └── SpringDataAccountRepository
    ├── domain
    |   ├── Account
    |   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            |   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort”
```



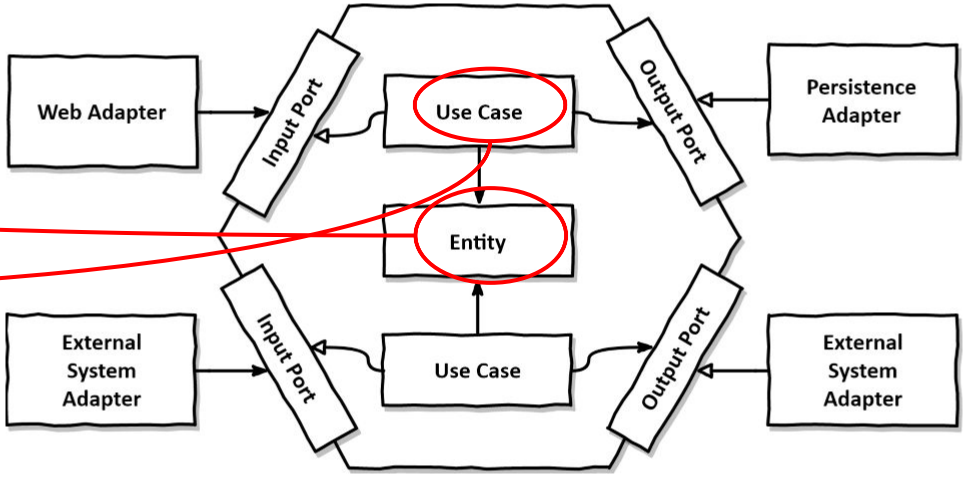
An Architecturally Expressive Package Structure

```
buckpal
└── account
    ├── adapter
    |   ├── in
    |   |   └── web
    |   |       └── AccountController
    |   ├── out
    |   |   └── persistence
    |   |       ├── AccountPersistenceAdapter
    |   |       └── SpringDataAccountRepository
    ├── domain
    |   ├── Account
    |   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            |   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort”
```



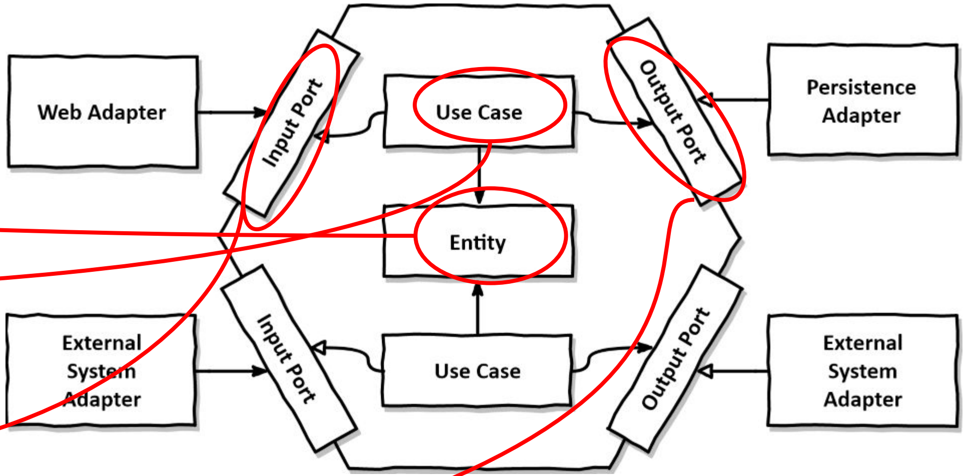
An Architecturally Expressive Package Structure

```
buckpal
└── account
    ├── adapter
    |   ├── in
    |   |   └── web
    |   |       └── AccountController
    |   ├── out
    |   |   └── persistence
    |   |       ├── AccountPersistenceAdapter
    |   |       └── SpringDataAccountRepository
    ├── domain
    |   ├── Account
    |   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            |   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort”
```



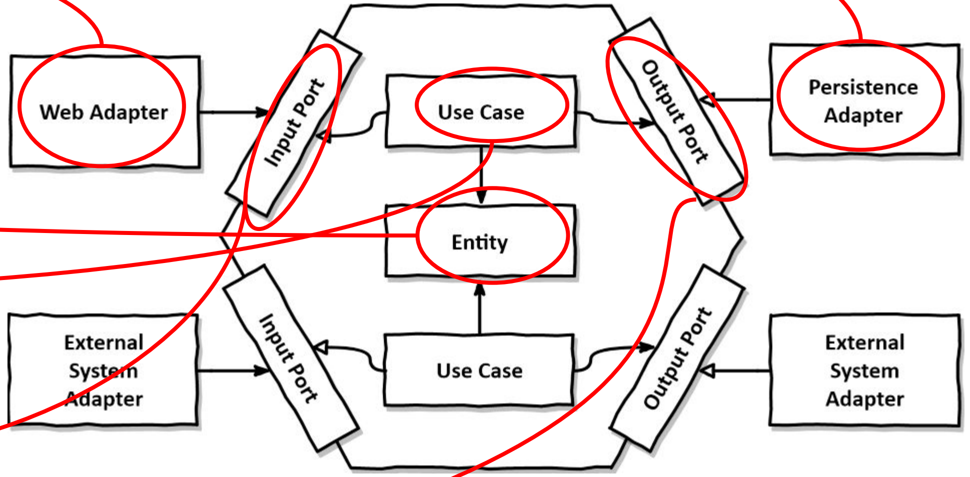
An Architecturally Expressive Package Structure

```
buckpal
└── account
    ├── adapter
    |   ├── in
    |   |   └── web
    |   |       └── AccountController
    |   ├── out
    |   |   └── persistence
    |   |       ├── AccountPersistenceAdapter
    |   |       └── SpringDataAccountRepository
    ├── domain
    |   ├── Account
    |   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            |   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort”
```



An Architecturally Expressive Package Structure

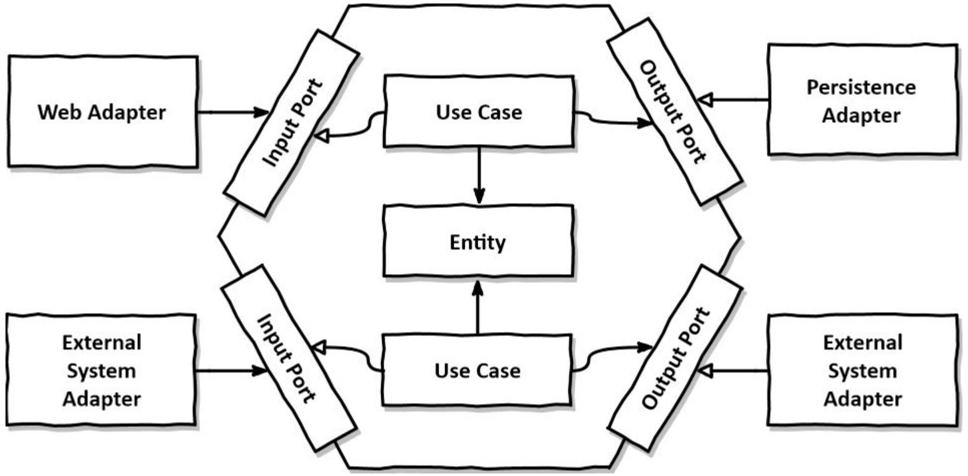
```
buckpal
└── account
    ├── adapter
    │   ├── in
    │   │   └── web
    │   │       └── AccountController
    │   ├── out
    │   │   └── persistence
    │   │       ├── AccountPersistenceAdapter
    │   │       └── SpringDataAccountRepository
    ├── domain
    │   ├── Account
    │   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            │   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort”
```



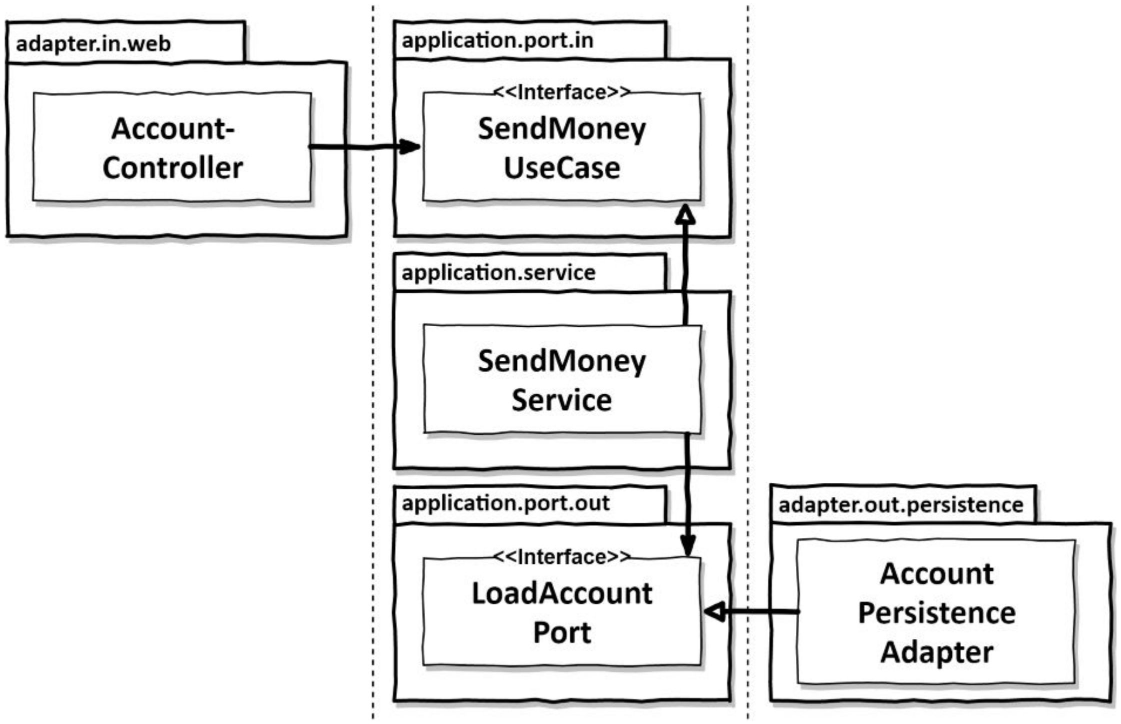
An Architecturally Expressive Package Structure

```
1 buckpal
2 └── account
3    ├── adapter
4    |  ├── in
5    |  |   └── web
6    |  |      └── AccountController
7    |  ├── out
8    |  |   └── persistence
9    |  |      ├── AccountPersistenceAdapter
10   |  |      └── SpringDataAccountRepository
11   ├── domain
12   |  ├── Account
13   |  └── Activity
14   └── application
15      └── SendMoneyService
16      └── port
17         ├── in
18         |  └── SendMoneyUseCase
19         └── out
20            ├── LoadAccountPort
21            └── UpdateAccountStatePort”
```

Allows working in parallel !!!



An Architecturally Expressive Package Structure



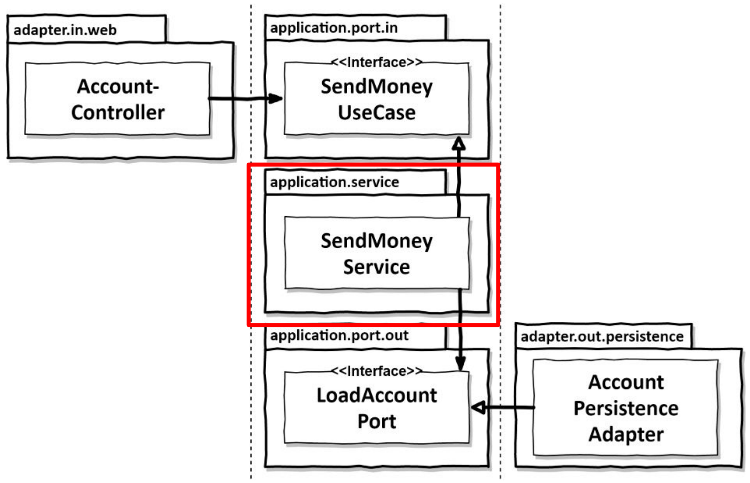
An Architecturally Expressive Package Structure

- This package structure helps reducing the so-called “architecture/code gap” or “model/code gap”. This expressive package structure promotes active thinking about the architecture
- **Adapter packages**: All classes they contain may be package private since they are not called by the outside world except over port interfaces, which live within the application package. So, no accidental dependencies from the application layer to the adapter classes.
- **Application and domain packages**: The ports must be public because they must be accessible to the adapters by design. The domain classes must be public to be accessible by the services and, potentially, by the adapters. The services don’t need to be public, because they can be hidden behind the incoming port interfaces.

Dependency Injection

Who provides the application with the actual objects that implement the port interfaces?

Dependency Injection: a neutral component that has a dependency to all layers



Let us get our hands dirty

https://github.com/thombergs/buckpal

https://reflectoring.io/

Implementing Domain

Domain: Account

Show Account in:

Io.reflectoring.buckpal.domain

https://github.com/thombergs/buckpal/blob/master/src/main/java/io/reflectoring/buckpal/application/domain/model/Account.java

Implementing Use Cases

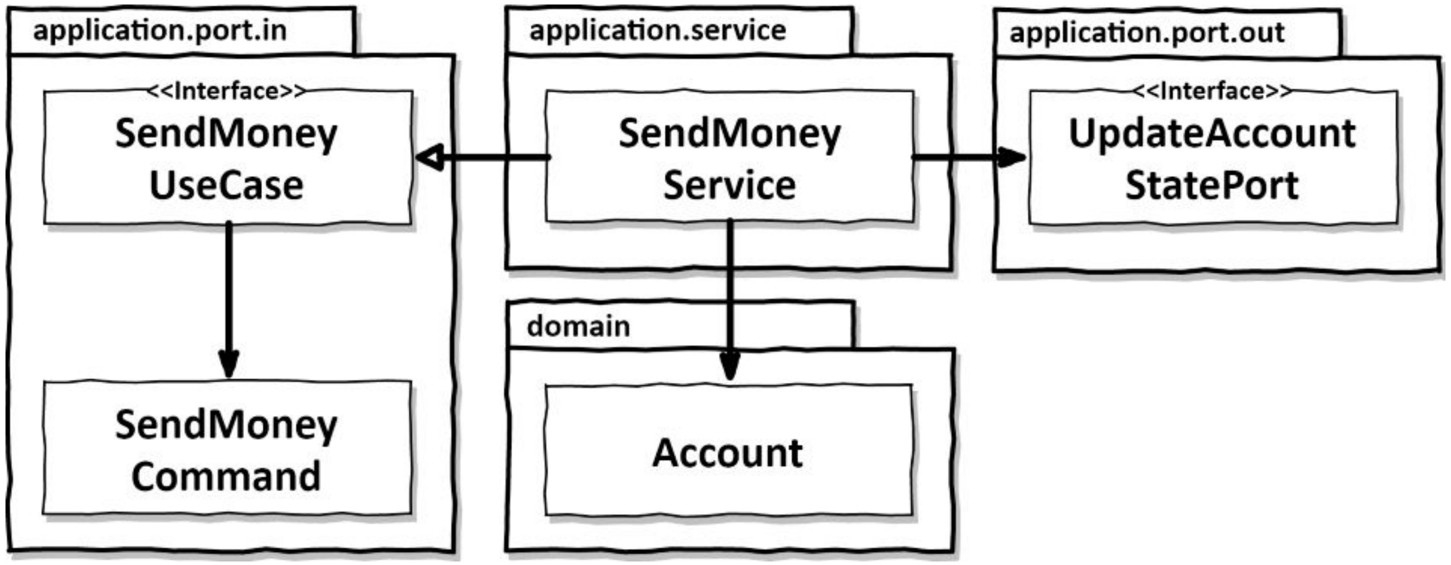
Send Money Use Case

1. Take input
2. Validate business rules
3. Manipulate model state
4. Return output

Create a separate class file for each use case to avoid having broad services.

Note that a service is a domain controller: validates business rules

Send Money Use Case



Send Money Use Case

```java
package buckpal.application.service;

@RequiredArgsConstructor
@Transactional
public class SendMoneyService implements SendMoneyUseCase {

  private final LoadAccountPort loadAccountPort;
  private final AccountLock accountLock;
  private final UpdateAccountStatePort updateAccountStatePort;

  @Override
  public boolean sendMoney(SendMoneyCommand command) {
    // TODO: validate business rules
    // TODO: manipulate model state
    // TODO: return output
  }
}
```

Send Money Use Case: **input**

- The input must be validated
- The input must be as specific to the use case as possible
- Do not share inputs (and input ports) with other use cases → that would couple them

Send Money Use Case: **who validates the input?**

The use case, the different adapters …

definitely the application layer → the input port

Show:
https://github.com/thombergs/buckpal/blob/master/src/main/java/io/reflectoring/buckpal/application/port/in/SendMoneyCommand.java

Note that:

Class immutable (record)

Create different ports for different use cases (keep them uncoupled)

Use cases: do NOT share input

Example: Register Account & Update Account Details

Send Money Use Case: **validate business rules**

- Business rules belong to use case logic
- A business rules requires accessing the current domain state while input validation does not
- Example:
  “the source account must not be overdrawn” → business rule
  “the transfer amount must be greater than zero” → input validation

Send Money Use Case: **where to validate business rules?**

Preferably in the domain

```java
package buckpal.domain;

public class Account {

  // ...

  public boolean withdraw(Money money, AccountId targetAccountId) {
    if (!mayWithdraw(money)) {
      return false;
    }
    // ...
  }
}
```

Send Money Use Case: **where to validate business rules**?

When not possible in the use case service/controller

```java
package buckpal.application.service;

@RequiredArgsConstructor
@Transactional
public class SendMoneyService implements SendMoneyUseCase {

  // ...

  @Override
  public boolean sendMoney(SendMoneyCommand command) {
    requireAccountExists(command.getSourceAccountId());
    requireAccountExists(command.getTargetAccountId());
    ...
  }
}
```

- If rules broken the service throws a dedicated exception
- The adapter interfacing with the user deals with the exception (showing it to the user for example)

Send Money Use Case: **where to validate business rules**?

Rich vs Anemic Domain Model

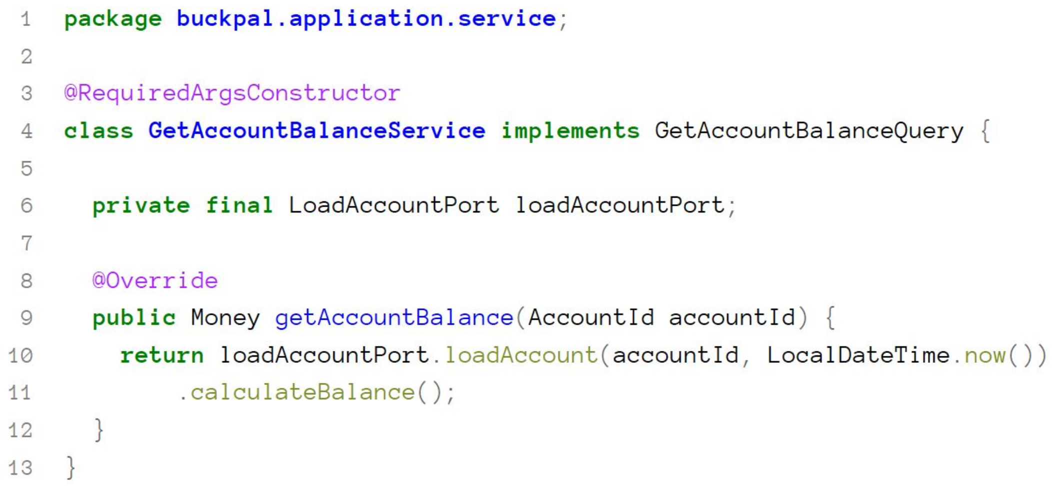
The hexagonal architecture is agnostic with respect types of models:
choose whatever fits your needs

Send Money Use Case: **output**

- As specific to the use case as possible. Only the data needed for the caller to work
- Send Money Use Case:
  - Return boolean indicating whether it’s done
  - Return the account (or both accounts)
- Return different output for different use cases → shared output tend to couple use cases
- You may want to avoid using the domain entities as output

Read-only Use Case

- We will separate (real) use cases from read-only ones which are simple queries for data



```java
package buckpal.application.service;

@RequiredArgsConstructor
class GetAccountBalanceService implements GetAccountBalanceQuery {

  private final LoadAccountPort loadAccountPort;

  @Override
  public Money getAccountBalance(AccountId accountId) {
    return loadAccountPort.loadAccount(accountId, LocalDateTime.now())
        .calculateBalance();
  }
}
```

- If we use the same model across layers we could let the client call the out port directly (since the use case is doing nothing)

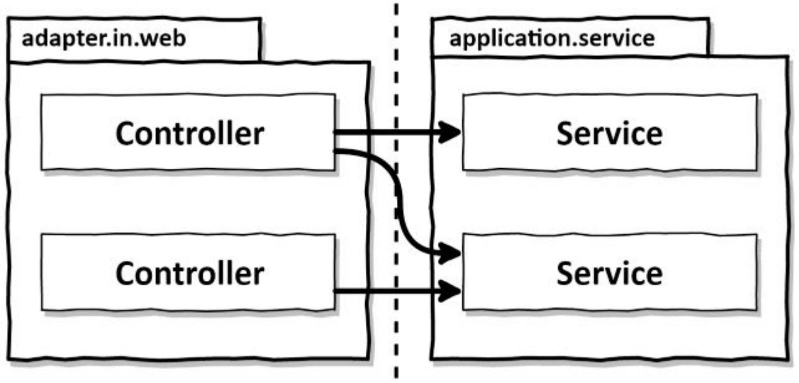
What do we win?

- More code): we don’t share models between use cases → create different models and map these models with domain entities

- Use case-specific models allow
  - For a crisp understanding of a use case, making it easier to maintain in the long run
  - Multiple developers to work on different use cases in parallel

Implementing a Web Adapter

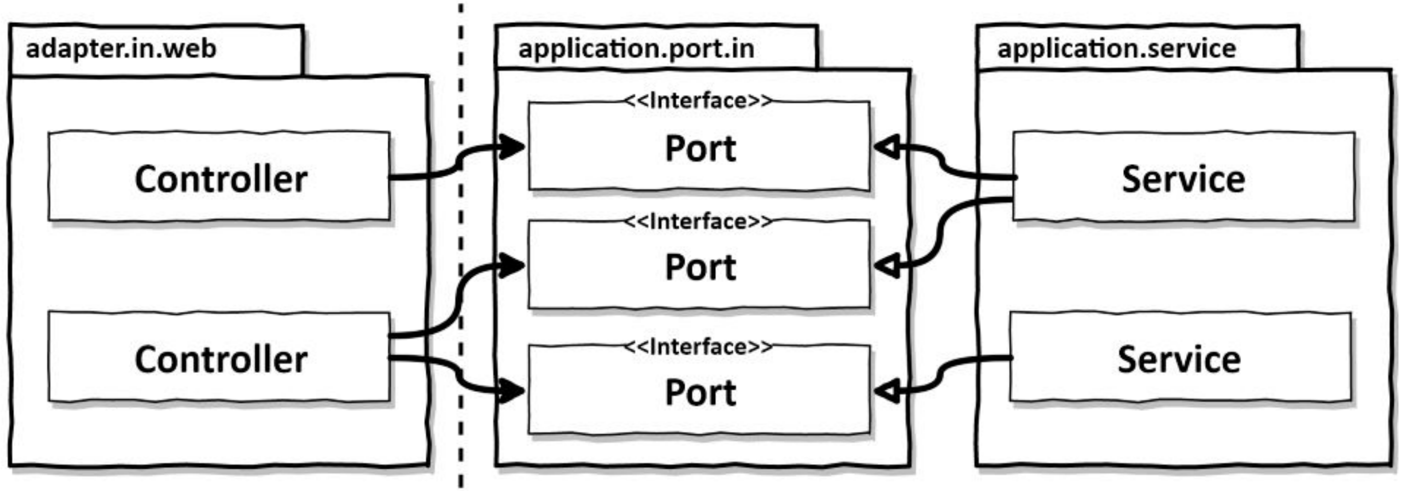
What we’ve done till now



Not bad at all!! (see the arrows)

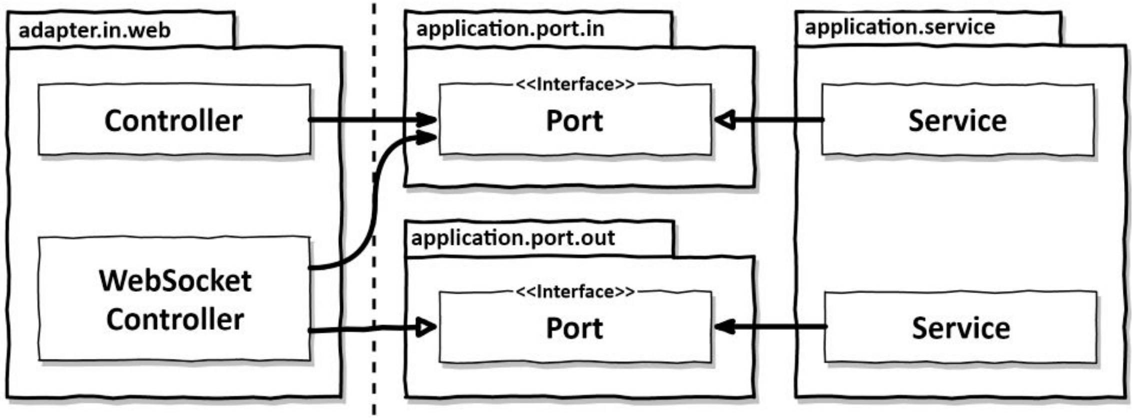
However we don’t see the specific communication between adapters and services

Web Adapter



- Why don’t we just skip these in ports interfaces? Really, really tempting!!!

“Reactive” Web Adapter



The WebSocketController is an incoming and outgoing adapter at the same time. In this case we definitely need an out port implemented by the WebSocketController and called by the application service

Web Adapter: Responsibilities

1. Map HTTP request to Java objects
2. Perform authorization checks
3. Validate input
4. Map input to the input model of the use case
5. Call the use case
6. Map output of the use case back to HTTP
7. Return HTTP response

Web Adapter: prefer small rather than large

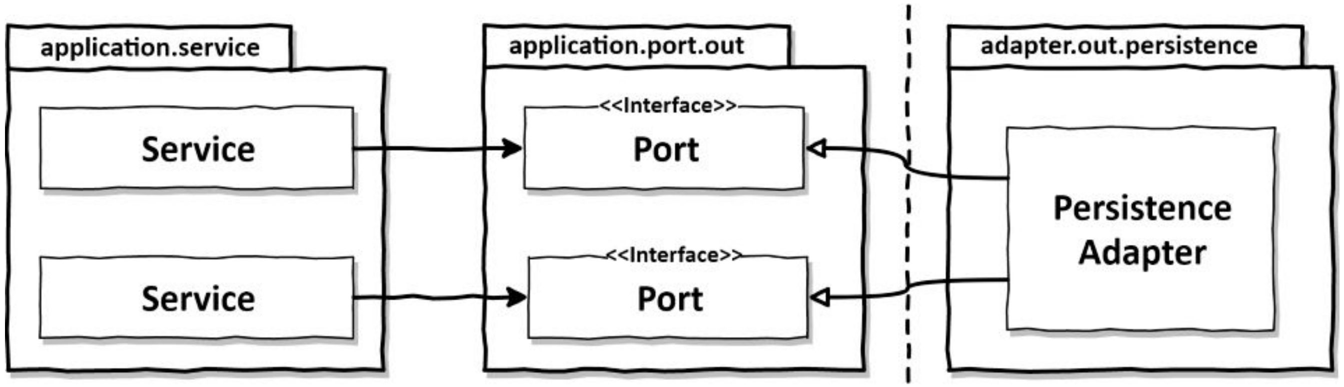
- Few lines of code per class → easier to maintain and to find
- Small class code means small test code → easier to maintain and to find
- Avoids re-use of data structure code. Example:
  - Account’s Id: is it included in objects in creation use cases?
  - Account one-to-many Users: do we include users in the account object?
- Allows parallel work

Web Adapter

```java
package buckpal.adapter.web;

@RestController
@RequiredArgsConstructor
public class SendMoneyController {

  private final SendMoneyUseCase sendMoneyUseCase;

  @PostMapping(path = "/accounts/sendMoney/{sourceAccountId}/{targetAccountId}/{amou\
nt}")
  void sendMoney(
      @PathVariable("sourceAccountId") Long sourceAccountId,
      @PathVariable("targetAccountId") Long targetAccountId,
      @PathVariable("amount") Long amount) {

    SendMoneyCommand command = new SendMoneyCommand(
        new AccountId(sourceAccountId),
        new AccountId(targetAccountId),
        Money.of(amount));

    sendMoneyUseCase.sendMoney(command);
  }

}
```

Implementing a Persistence Adapter

Persistence Adapter: dependency inversion

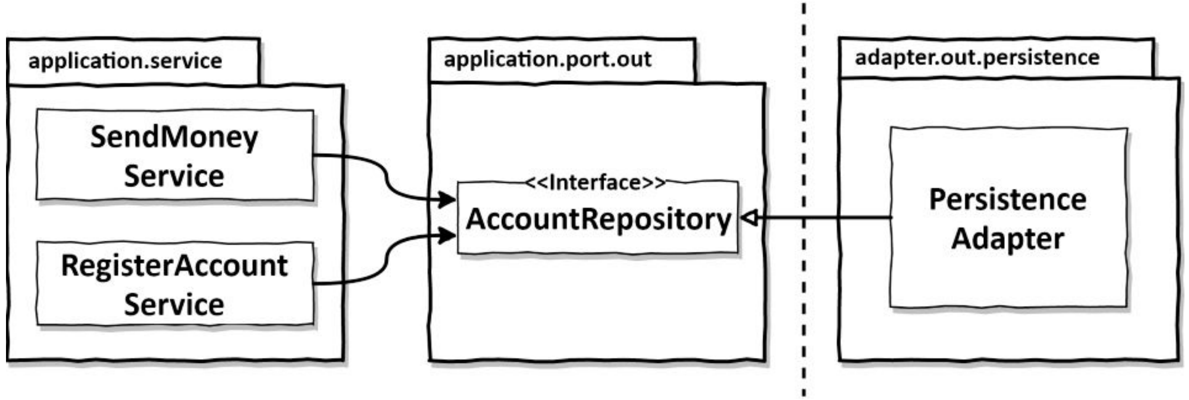


Persistence Adapter: Responsibilities

1. Take input
2. Map input into database format
3. Send input to the database
4. Map database output into application format
5. Return output

The important part is that the **input and output models** to/from the persistence adapter lie within the **application core** and not within the persistence adapter itself, so that changes in the persistence adapter don’t affect the core.

Persistence Adapter: Slicing Port Interfaces

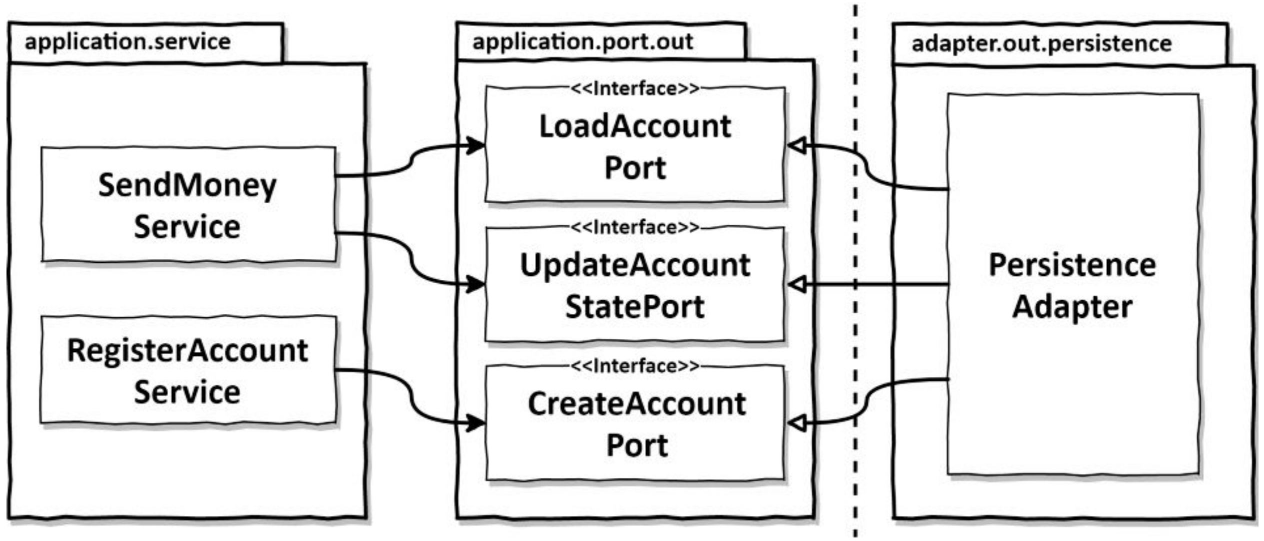


Dependencies on methods you don’t use can cause problems and confusion

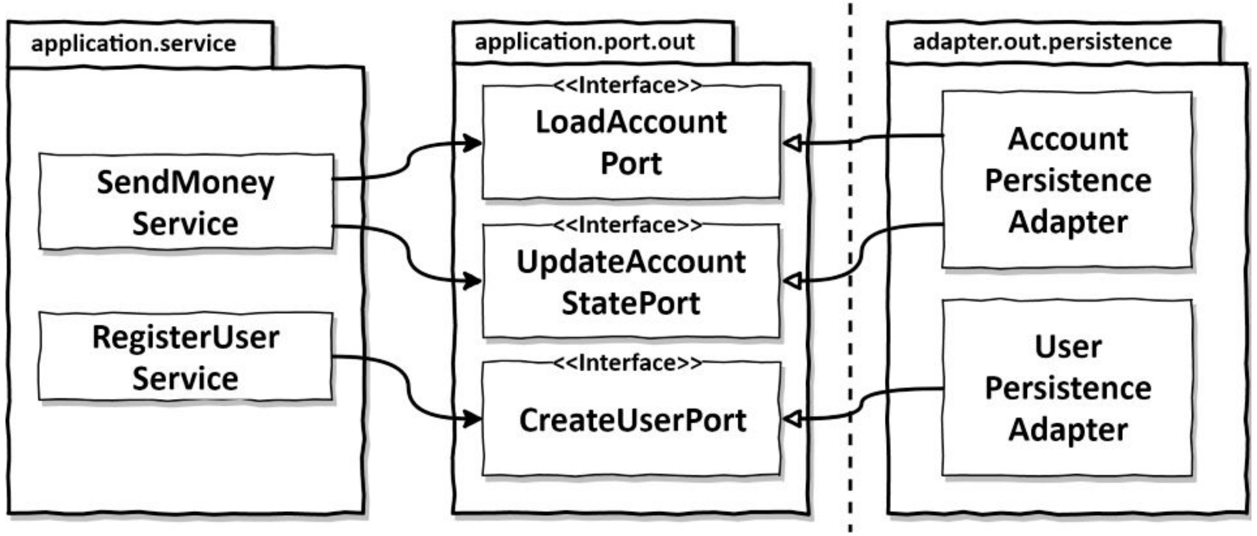
“**Depending on something that carries baggage that you don’t need can cause you troubles that you didn’t expect**”, Robert C. Martin

Persistence Adapter: Slicing Port Interfaces

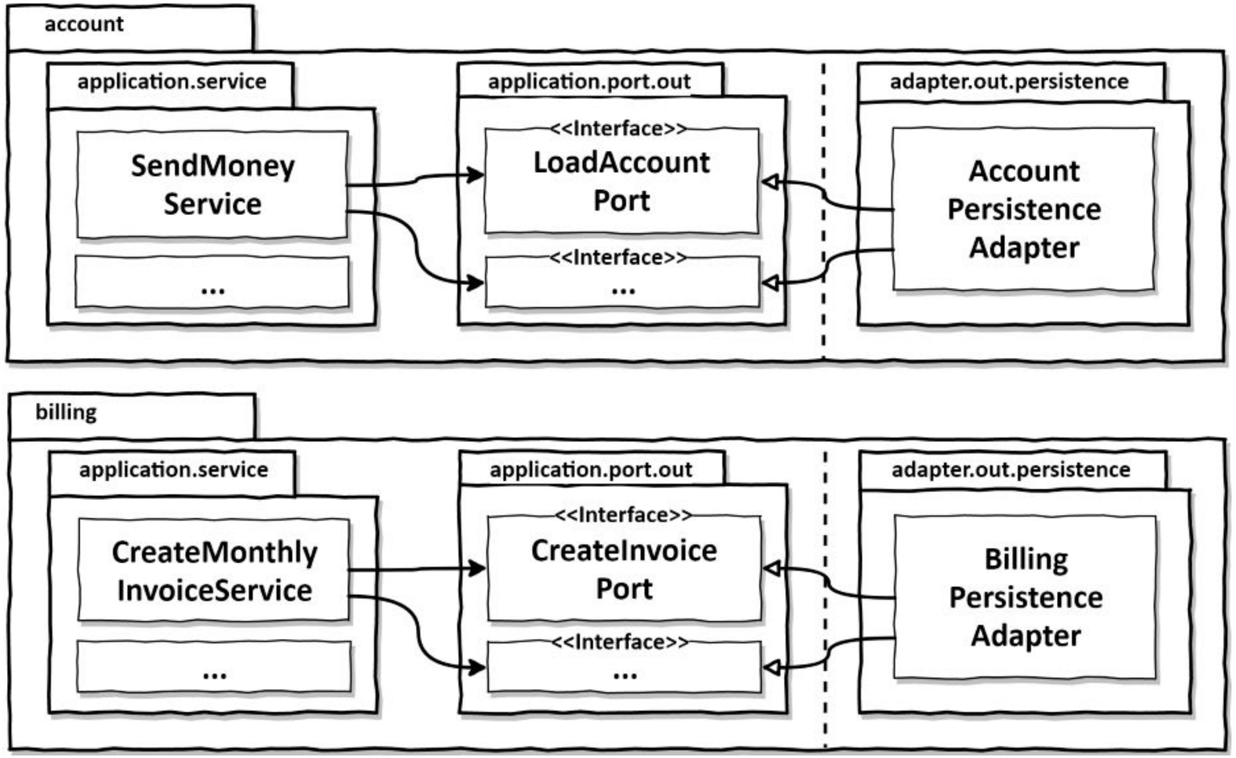
Interface Segregation Principle: that broad interfaces should be split into specific ones so that clients only know the methods they need



Persistence Adapter: Slicing Persistence Adapters



Bounded Contexts: slicing applications



Persistence Adapter: code example

Out port:
https://github.com/thombergs/buckpal/tree/master/src/main/java/io/reflectoring/buckpal/application/port/out

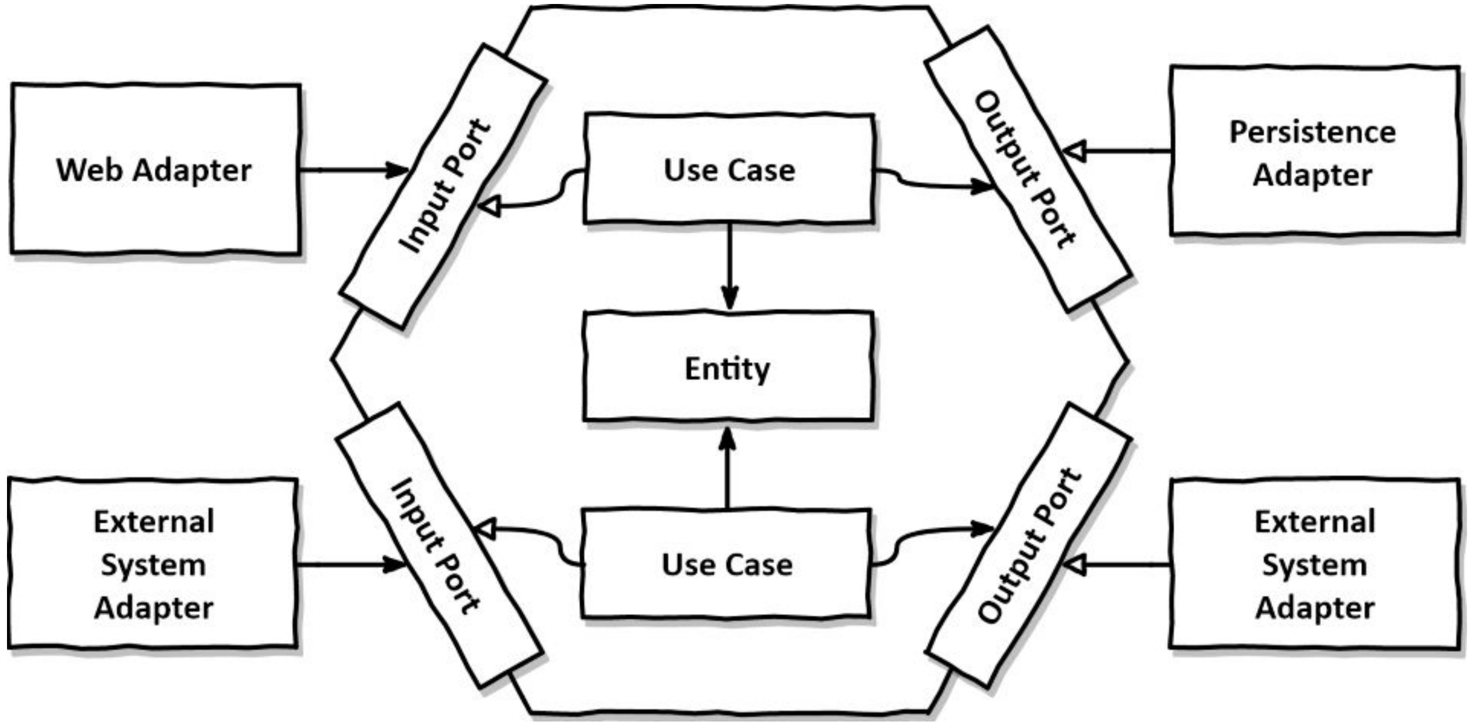
Adapter:
https://github.com/thombergs/buckpal/tree/master/src/main/java/io/reflectoring/buckpal/adapter/out/persistence

Better to have specific persistence models (meaning more code and work) to avoid compromises of the domain model to the persistence layer

Transactions are handled at the services (@Transactional)

Mapping Between Boundaries: Strategies

Hexagonal Architecture: clean arch. made concrete



Mapping vs. No Mapping

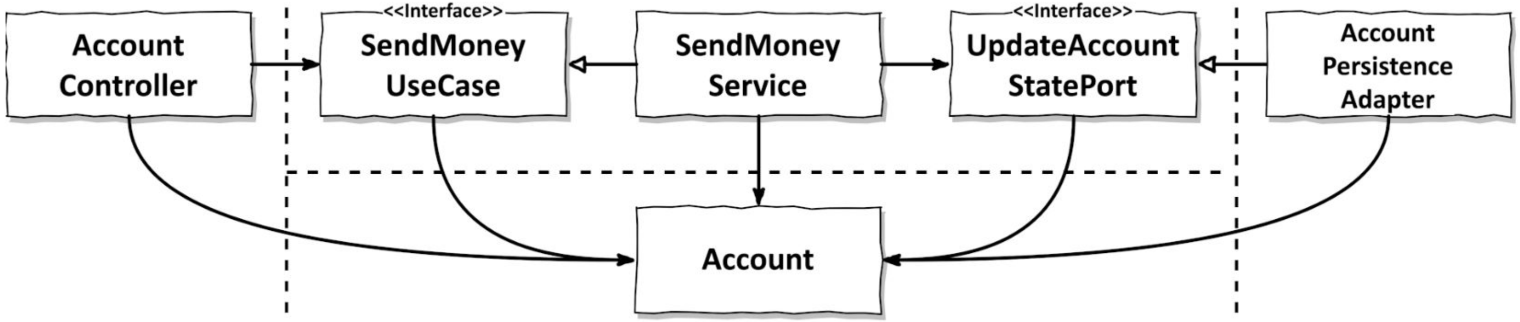
Pro-Mapping Developer:

> If we don’t map between layers, we have to use the same model in both layers which means that the layers will be **tightly coupled**!

Contra-Mapping Developer:

> But if we do map between layers, we produce a lot of **boilerplate** code which is overkill for many use cases, since they’re only doing CRUD and have the same model across layers anyways!

The “No Mapping” Strategy



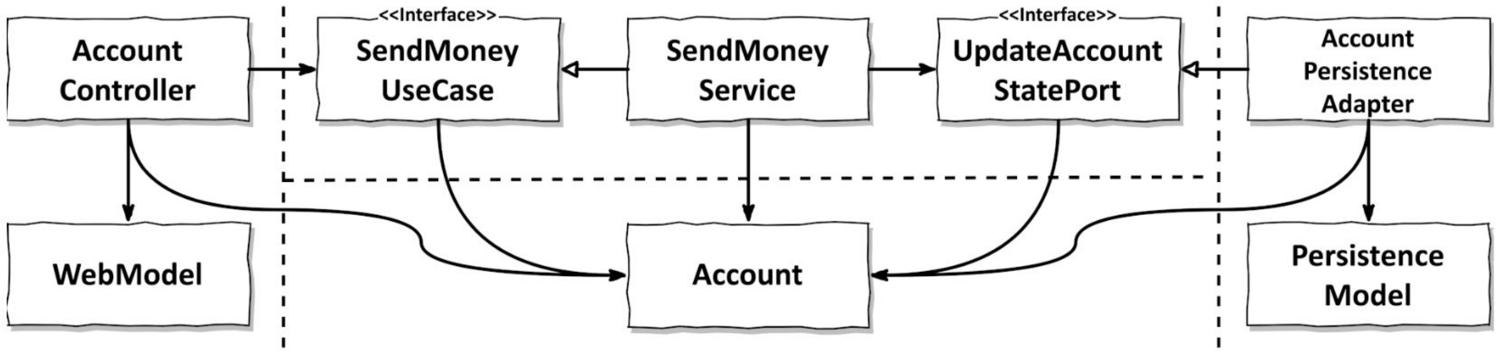
- Account “polluted” with annotations from the JSON and the JPA libraries: SRP violation!!!
- Each layer might require certain custom fields on the Account class. This might lead to a fragmented domain model with certain fields only relevant in one layer

The “No Mapping” Strategy

Simple CRUD use case: Do we need to map the same fields from the web model into the domain model and from the domain model into the persistence model?

- **NO**: As long as all layers need exactly the same information in exactly the same structure
- Many use cases can start their life as simple CRUD use cases. Later, they might grow into a full-fledged business use case with a rich behavior and validations which justify a more expensive mapping strategy

Two-Way Mapping

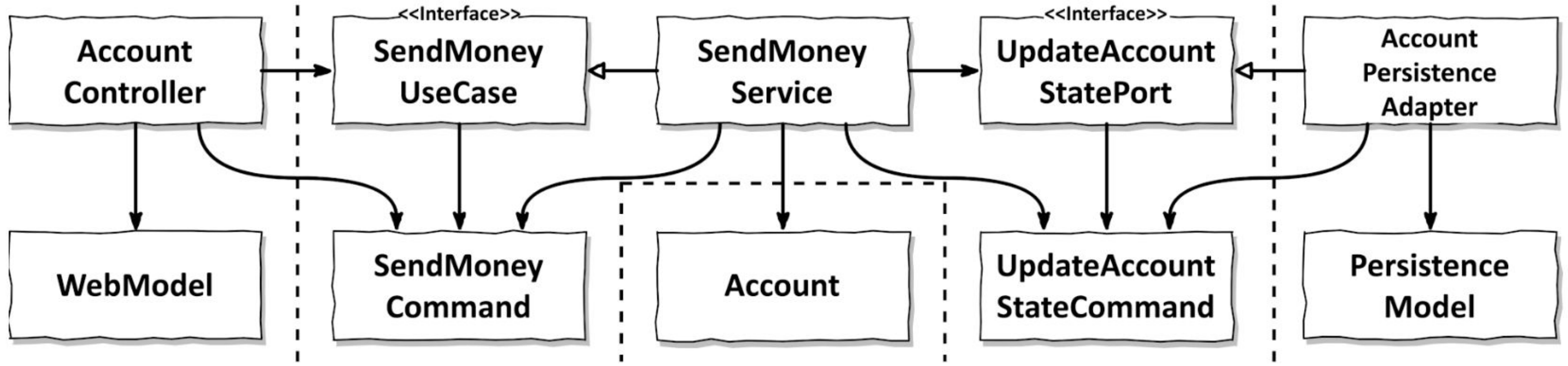


- Each layer can modify its own model without affecting the other layers (as long as the contents are unchanged)
- Clean domain model that is not dirtied by web or persistence concerns
- The mapping responsibilities are clear: the outer layers/adapters map into the model of the inner layers and back.

Two-Way Mapping: drawbacks

- Lots of boilerplate code which in turn is difficult to test (specially when using mapping libraries)
- The domain model is used to communicate across layer boundaries. The incoming ports and outgoing ports use domain objects as input parameters and return values.
  - This makes them vulnerable to changes that are triggered by the needs of the outer layers
  - It’s desirable for the domain model only to evolve due to the needs of the domain logic.

The “Full” Mapping Strategy

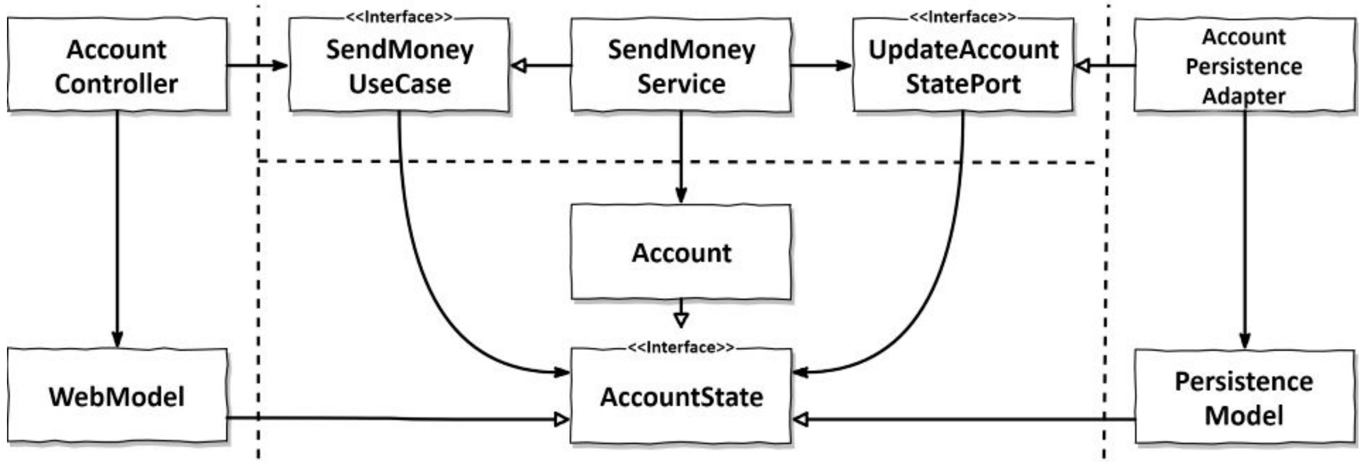


- This strategy introduces a separate input and output model **per operation**. We use a model specific to each operation. We can call those models “commands”, “requests”, or similar.
- These commands (models) make the interface to our application very explicit
- More mapping overhead but mappers are simpler to code due their simplicity

The “Full” Mapping Strategy: when to use it?

- Use it between incoming adapters and the application layer to clearly demarcate the state-modifying use cases of the application.
- You could restrict this kind of mapping to the input model of operations and simply use a domain object as the output model. Example: the sendMoneyUseCase gets a command (specific model) but returns a Account object (a model from the domain)
- You could also avoid using it between applications and persistence layer

The “One-Way” Mapping Strategy



The models in all layers implement the same interface that encapsulates the state of the domain model by providing **getter** methods on the relevant attributes. The modifying behaviour is NOT exposed.

The “One-Way” Mapping Strategy

- If application layer wants to pass a domain object to the outer layers, it can do so without mapping, since the domain object implements the state interface expected by the incoming and outgoing ports.
- The outer layers decide if they can work with the interface or they map them to their own model
- The objects we pass from the outer layer to the application layer are mapped into a real domain model to get access to their behaviour
- If a layer receives an object from another layer, we map it into something the layer can work with. Thus, each layer only maps **one way**.
- This strategy plays out its strength best if the models across the layers are similar. For read-only operations

When to use which Mapping Strategy? (it depends)

- Probably use more than one
- Change strategy when conditions change
- Have agreed upon and precise rules to choose strategy

Example rules:
Modifying use case: “full mapping” strategy is the first choice between the web - application layers, in order to decouple the use cases from one another. This gives clear per-use-case validation rules and we don’t have to deal with fields we don’t need in a certain use case.

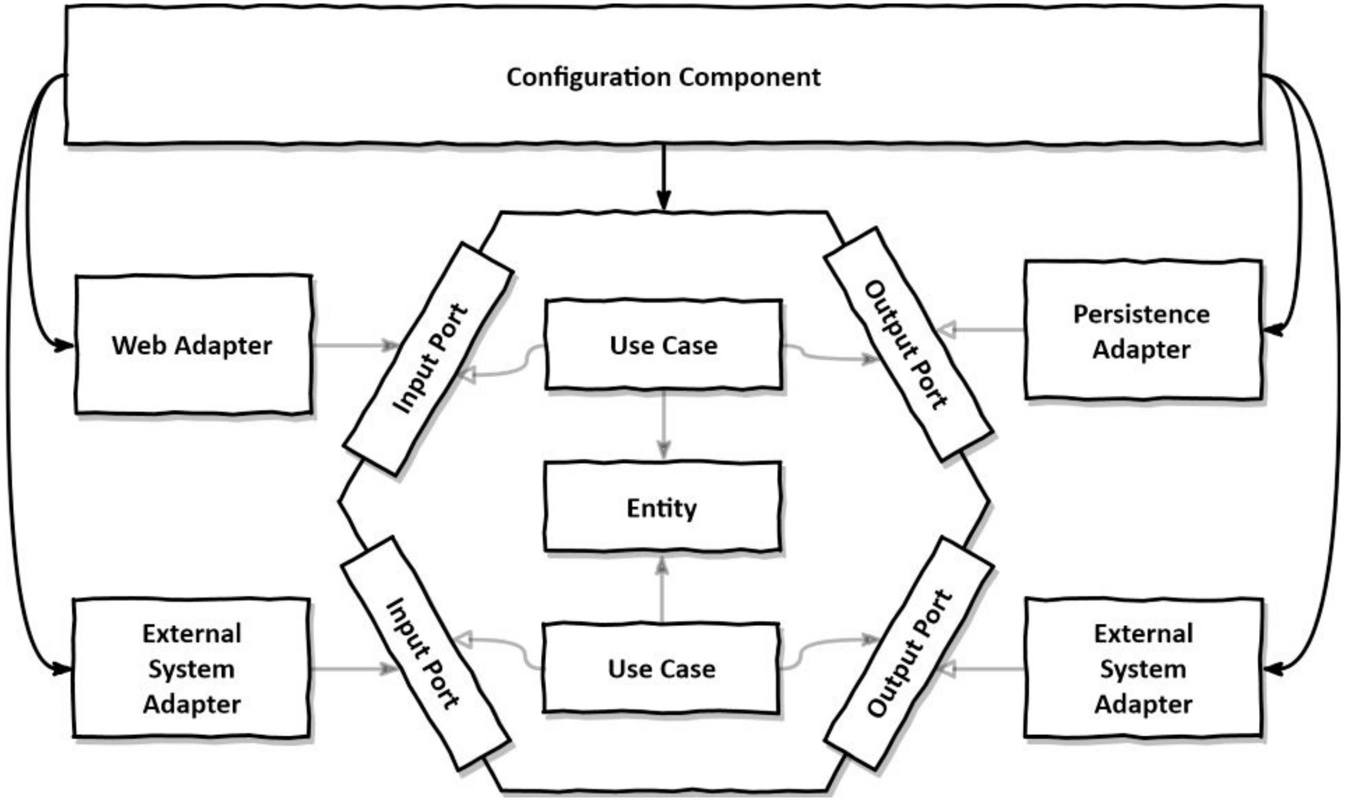
Modifying use case: “no mapping” strategy is the first choice between the application - persistence layer to quickly evolve the code without mapping overhead. As soon as we have to deal with persistence issues in the application layer, however, we move to a “two-way” mapping strategy to keep persistence issues in the persistence layer.

Assembling the Application

Assembling the Application

- If a use case needs to call a persistence adapter and just instantiates it itself, we have created a code dependency in the wrong direction. See service and out port: if a service creates an adapter object, it’d have an arrow in the wrong direction
- We need a neutral **configuration component** that will be coupled with ALL application’s layers.

Configuration Component



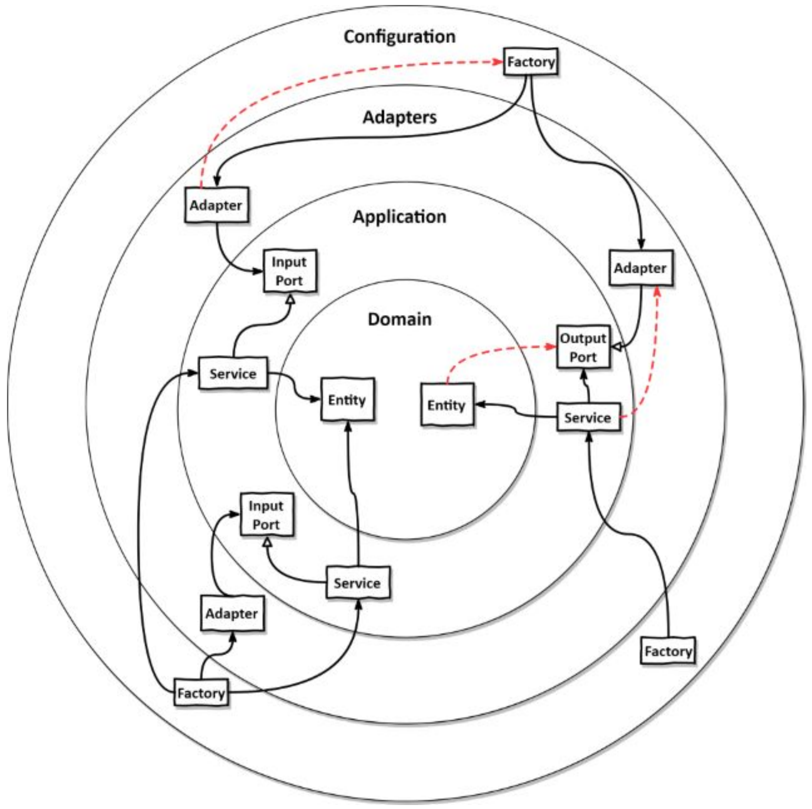
Configuration Component: responsibilities

- Create web adapter instances
- Ensure that HTTP requests are actually routed to the web adapters
- Create use case instances
- Provide web adapters with use case instances
- Create persistence adapter instances
- Provide use cases with persistence adapter instances
- Ensure that the persistence adapters can actually access the database
- Access certain sources of configuration parameters, like configuration files or command line parameters

Yes, you are right! Configuration component violates SRP!!!

Enforcing Architecture Boundaries

Enforcing Architecture Boundaries

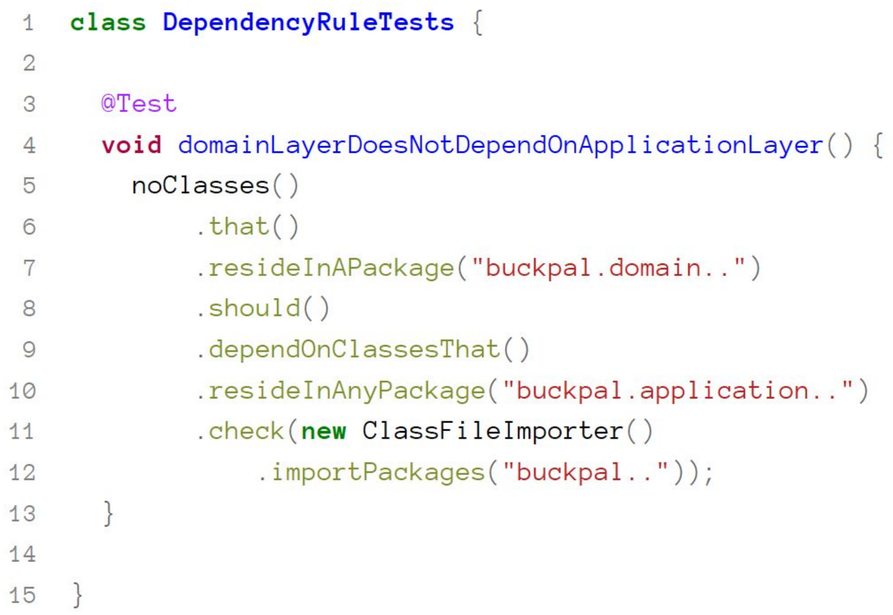


Visibility Modifiers: package default visibility

```
buckpal
└── account
    ├── adapter
    |   ├── in
    |   |   └── web
    |   |       └── o AccountController
    |   ├── out
    |   |   └── persistence
    |   |       ├── o AccountPersistenceAdapter
    |   |       └── o SpringDataAccountRepository
    ├── domain
    |   ├── + Account
    |   └── + Activity
    └── application
        └── o SendMoneyService
        └── port
            ├── in
            |   └── + SendMoneyUseCase
            └── out
                ├── + LoadAccountPort
                └── + UpdateAccountStatePort
```

Package visibility is not enough when subpackages need to be created

Post-compile Checks: ArchUnit



```java
class DependencyRuleTests {

  @Test
  void domainLayerDoesNotDependOnApplicationLayer() {
    noClasses()
        .that()
        .resideInAPackage("buckpal.domain..")
        .should()
        .dependOnClassesThat()
        .resideInAnyPackage("buckpal.application..")
        .check(new ClassFileImporter()
            .importPackages("buckpal.."));
  }

}
```

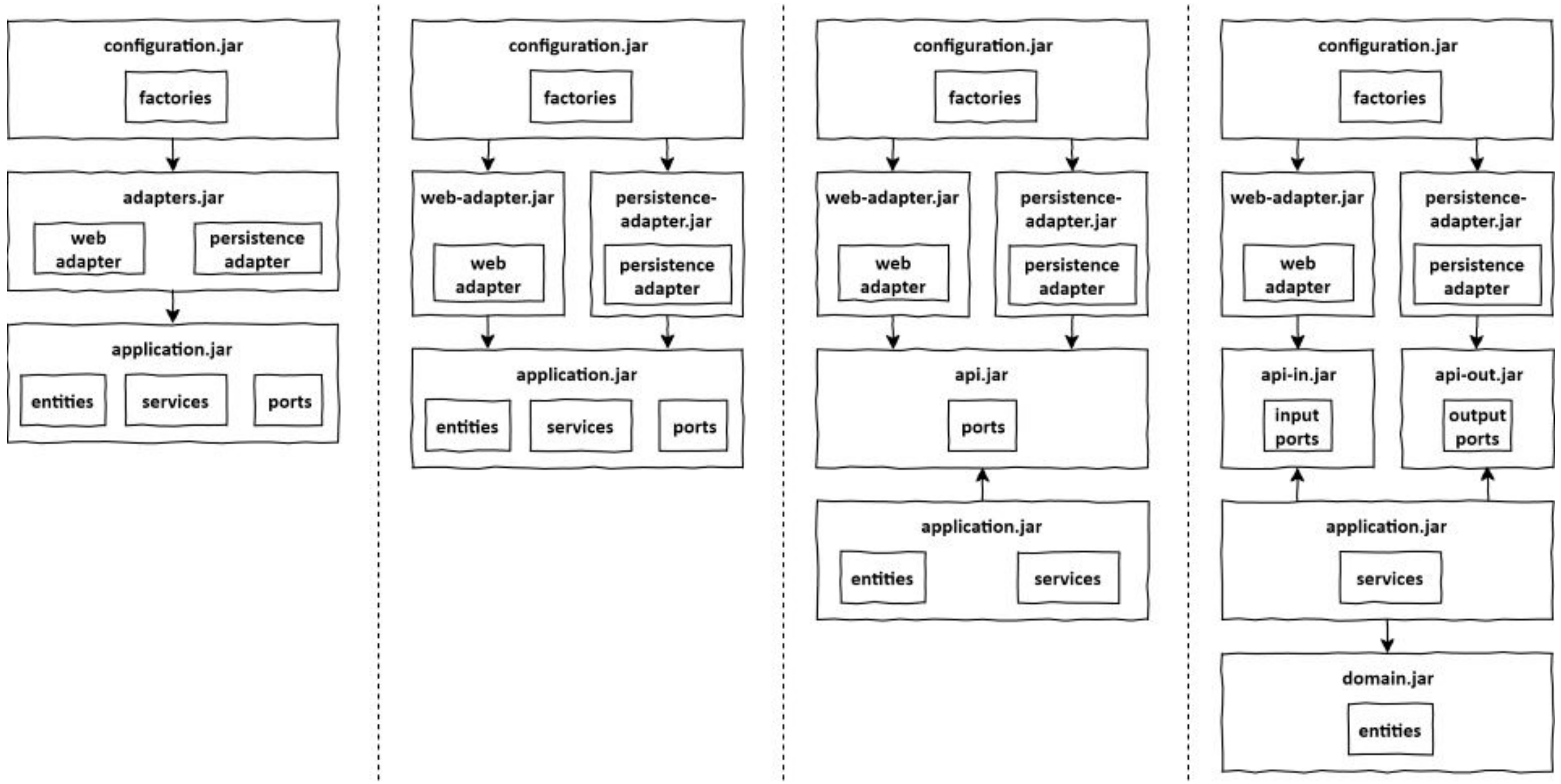
- Check during runtime (testing) using the ArchUnit library
- This kind of tests are difficult to maintain: name, package structure changes

Build Artifacts: modules

- Use automatic build tools (maven, gradle) that allow having complex module structures
- Each module compile into a jar file
- Building is about dependency checking!!!
- We can create a different build module for each application/architecture layer and specify only those dependencies allowed by the architecture

Let the compiler complain!!!

Build Artifacts: modules



Build Artifacts: modules

- The finer we cut, however, the more mapping we have to do between those modules, enforcing one of the mapping strategies
- Build tools hate circular dependencies :-)
- Modules allow separate compilation and testing. If adapters and application are in the same module, and we are in the middle of changing adapters (with compilation errors), we won’t be able to run (and build) application tests due to errors that do not affect the application layer
- But of course, we’ll need to create and keep the compilation scripts

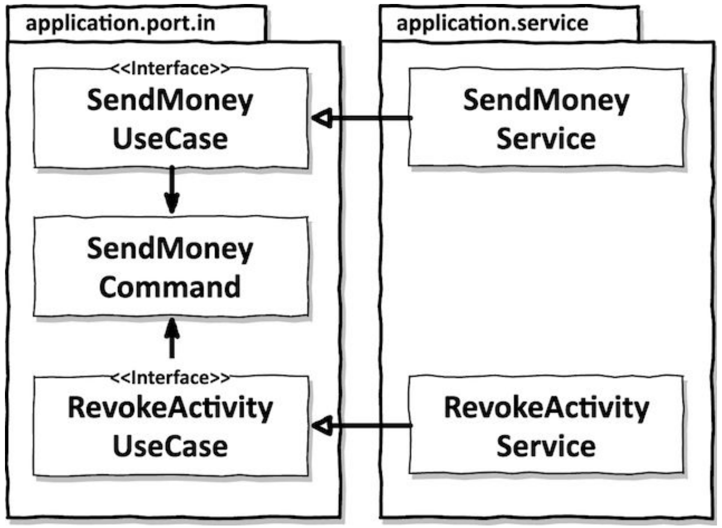
Take Shortcuts Consciously

Take Shortcuts Consciously

As soon as something looks run-down, damaged, [insert negative adjective here], or generally untended, the human brain feels that it’s OK to make it more run-down, damaged, or [insert negative adjective here].

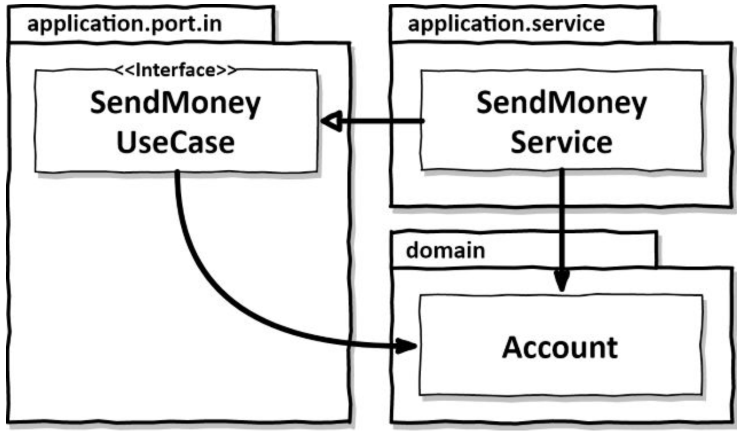
- When working on a low-quality codebase, the threshold to add more low-quality code is low
- When working on a codebase with a lot of coding violations, the threshold to add another coding violation is low
- When working on a codebase with a lot of shortcuts, the threshold to add another shortcut is low
- …

Shortcut 1: Sharing I/O Models Among Use Cases



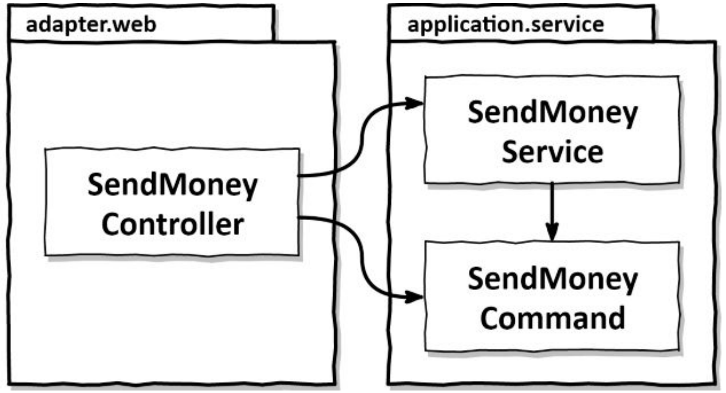
- Use cases become coupled through the model. If SendMoney needs to change the Command, RevokeAct will be affected.~~SPR~~
- OK **only if** use cases are functionally bound: *we actually want both use cases to be affected if we change a certain detail*
- When multiple use cases share i/o models, it's worthwhile to regularly ask the question whether the use cases should **evolve separately** from each other. As soon as the answer becomes a "yes", it's time to separate the input and output models

Shortcut 2: Using Domain Entities as I/O Models



- Entities have more reasons to change: use cases may force changes in models. ~~SPR~~
- OK **only if** use cases are CRUD
- As soon as a use case evolves to a more complicated one with domain logic, use a dedicated i/o model
- In Agile most use cases begin being CRUD and evolve to more complex ones

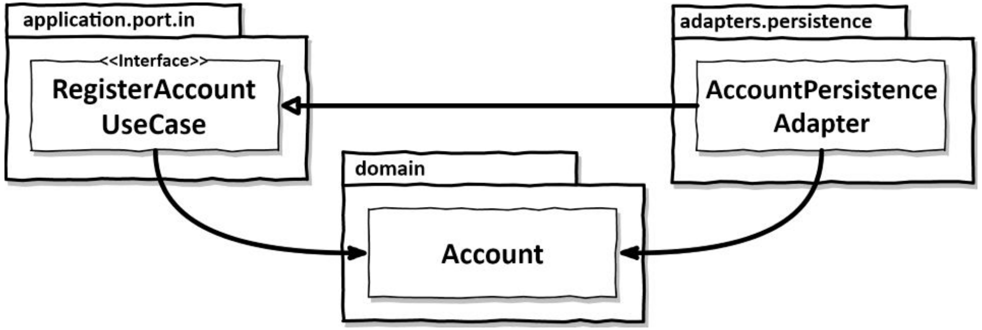
Shortcut 3: Skipping Incoming Ports



- Incoming ports are not necessary to invert the dependency adapter → application
- Incoming ports define the entry points to our application (sort of documentation) and help enforce the architecture
- OK **only if** the application is really small or only has a single incoming adapter

Shortcut 4: Skipping Application Services

- It is very tempting to do this for simple CRUD use cases, since in this case an application service usually only forwards a create, update or delete request to the persistence adapter, without adding any domain logic. Instead of forwarding, we can let the persistence adapter implement the use case directly.
- Ok **only if** use case is CRUD
- Prefer this than shortcut 3. Like shortcut 2, add the abstraction layer (application service) as soon as use case acquires domain logic


application.port.in
<<Interface>>
RegisterAccount UseCase
adapters.persistence
AccountPersistence Adapter
domain
Account