Spring Cleaning

Implementing the Clean Architecture with SpringBoot

DevNexus 2022 ❖ M. Jeff Wilson ❖ @mjeffw

* Agile Coach
* Certified SAFe® Program Consultant (SPC)
* Senior/Lead Developer/Architect
* DevOps + Test Automation SME

geek.macbeer@gmail.com
@mjeffw
https://linkedin.com/in/mjeffwilson

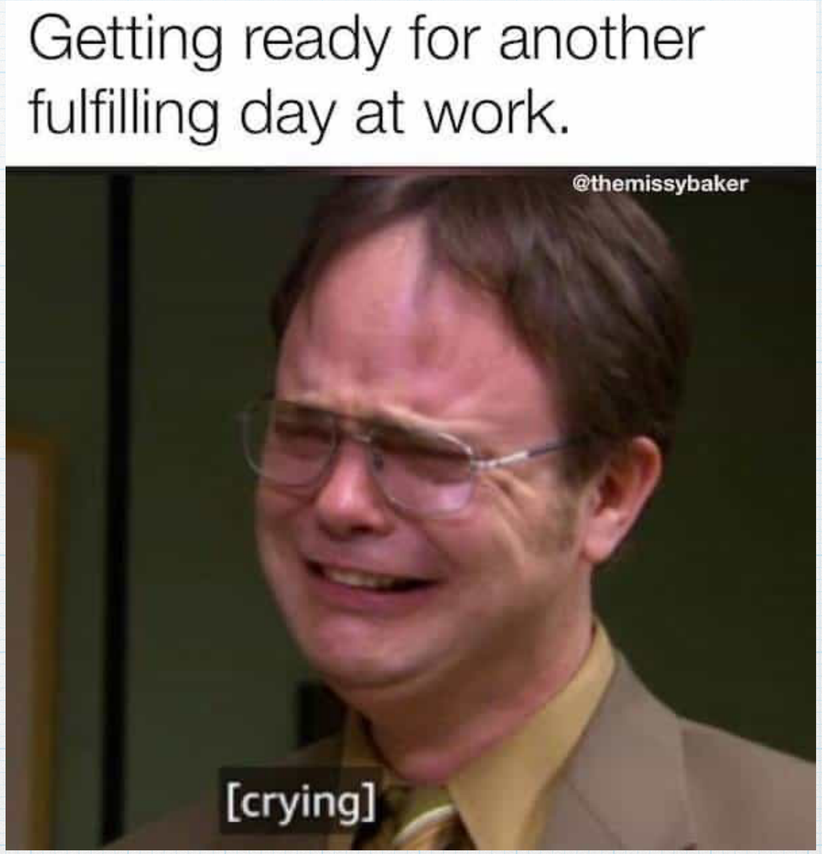
Coaching at the intersection of code and agile.

My Current Context

- Large enterprise IT department
  - App teams both employee and outsourced
  - Nontrivial requirements
  - Schedule and budget constraints
    - Funded for new functionality
    - Refactoring and documentation discouraged

Culture is What You Do

- Focus on adding functionality, not technical quality
- Documentation is low priority
- “It works” becomes good enough
- No bandwidth to review every PR or proposed change
- Architecture decisions are tribal knowledge



The Big Ball of Mud

- Leaky abstractions and fuzzy layers
- Specific responsibilities spread haphazardly across the code
- Organization slowly disintegrates
- Multiple, overlapping, libraries introduced
- Naming conventions drift or disappear

"Architecture is about the important stuff. Whatever that is."

—Ralph Johnson

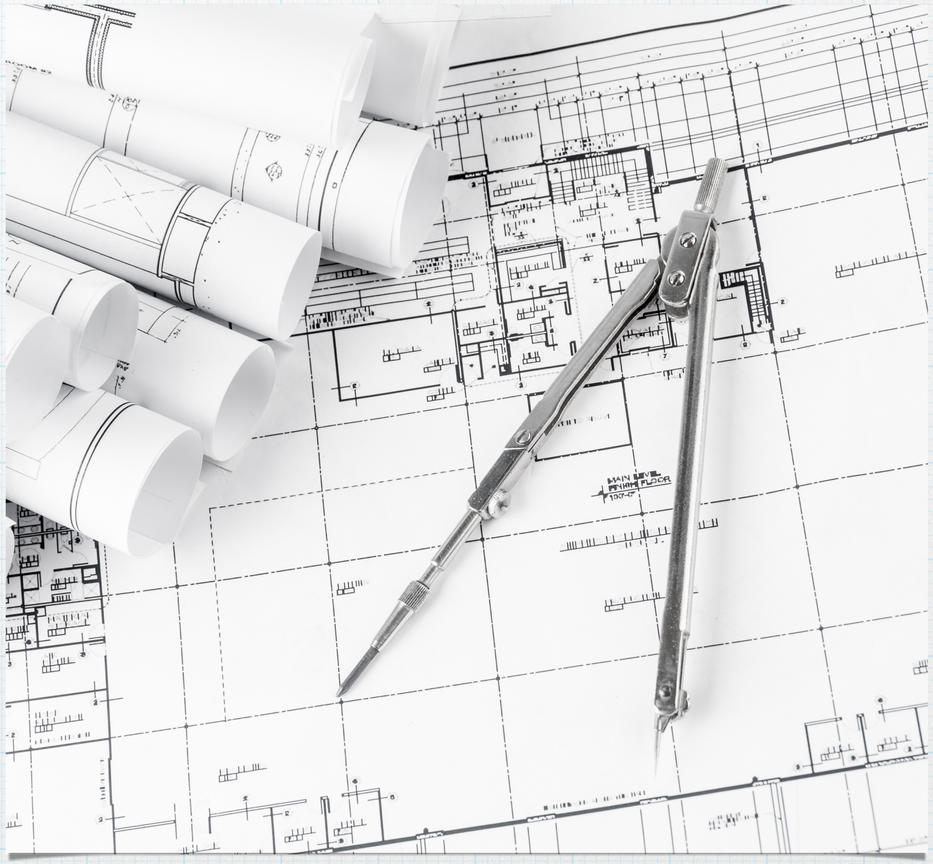
Architecture

"Architecture is the set of design patterns, organization, and conventions that improve the comprehensibility and changeability of the application.

"The goal of these decisions are to make it easy and safe for developers to identify the part of the application that needs to change, and to make that change predictably."

Architecture includes...

- Programming language(s)
- Source code directory layout
- Naming conventions
- Dependency strategy
- Framework(s)
- Design Patterns or metaphors

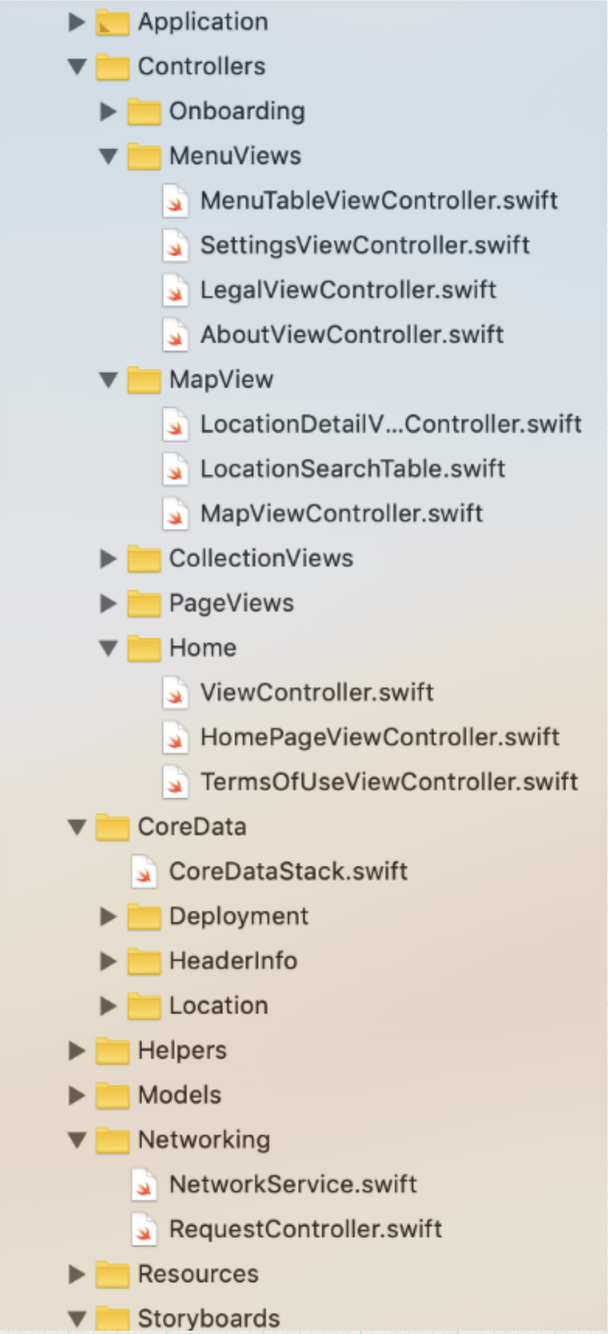


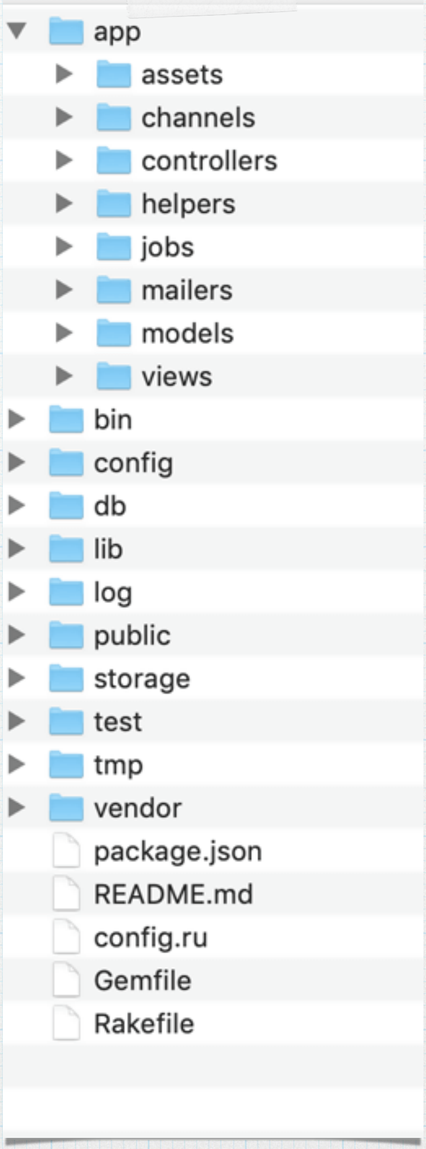
"The job of architect is to defer decisions as long as possible, to make it possible to write most of the code before making that decision."

"A good architecture maximizes the number of decisions not made."

—Robert Martin

But what does it do?





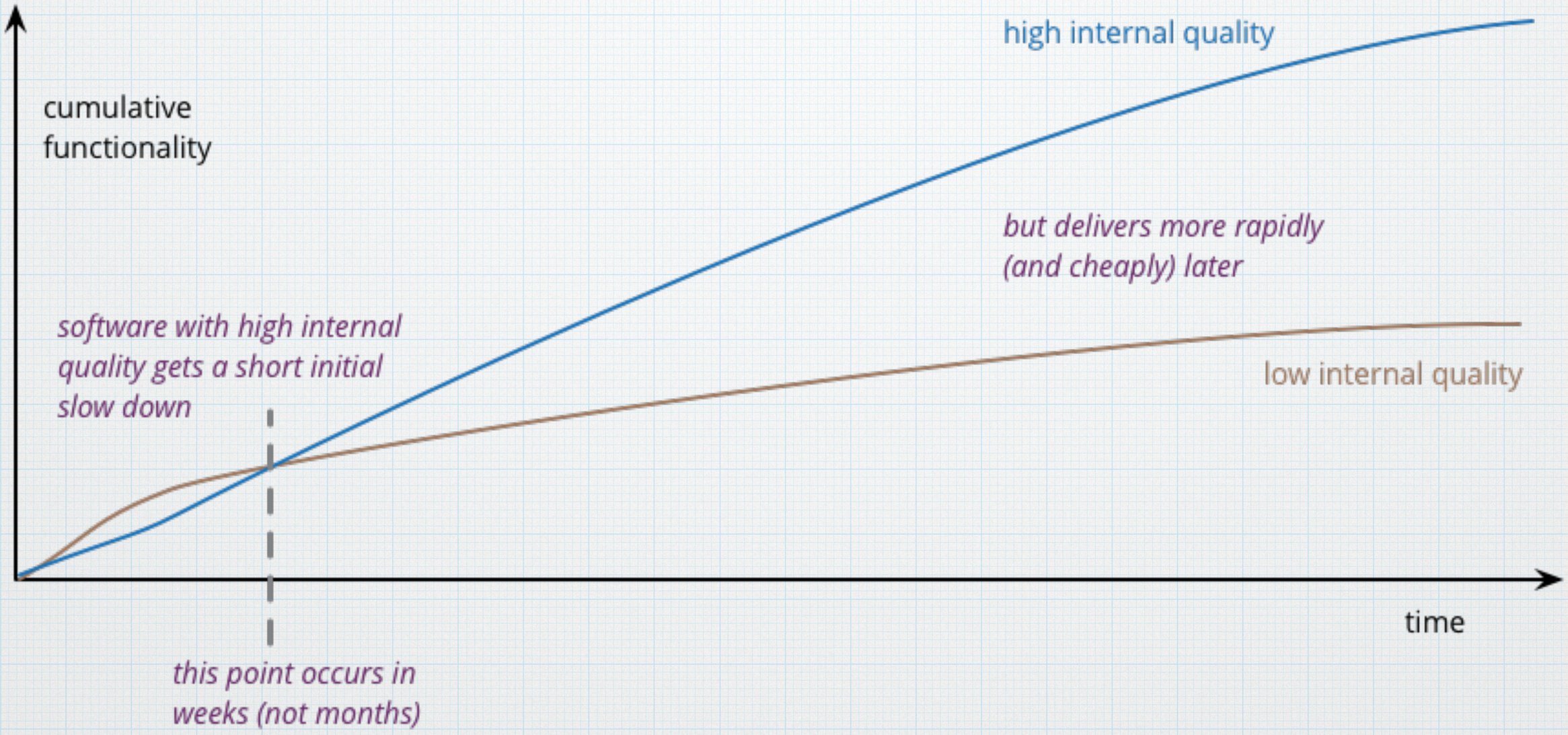
Start with the business problem(s) to solve

What makes code “good”?

* Single-responsibility: Every class should have a single responsibility.
* Open-closed: Software should be open for extension, closed for modification.
* Liskov substitution: An interface’s implementation can change without the caller caring or knowing.
* Interface segregation: Many client-specific interfaces are better than one general-purpose interface.
* Dependency inversion: Depend upon abstractions, not concretions.

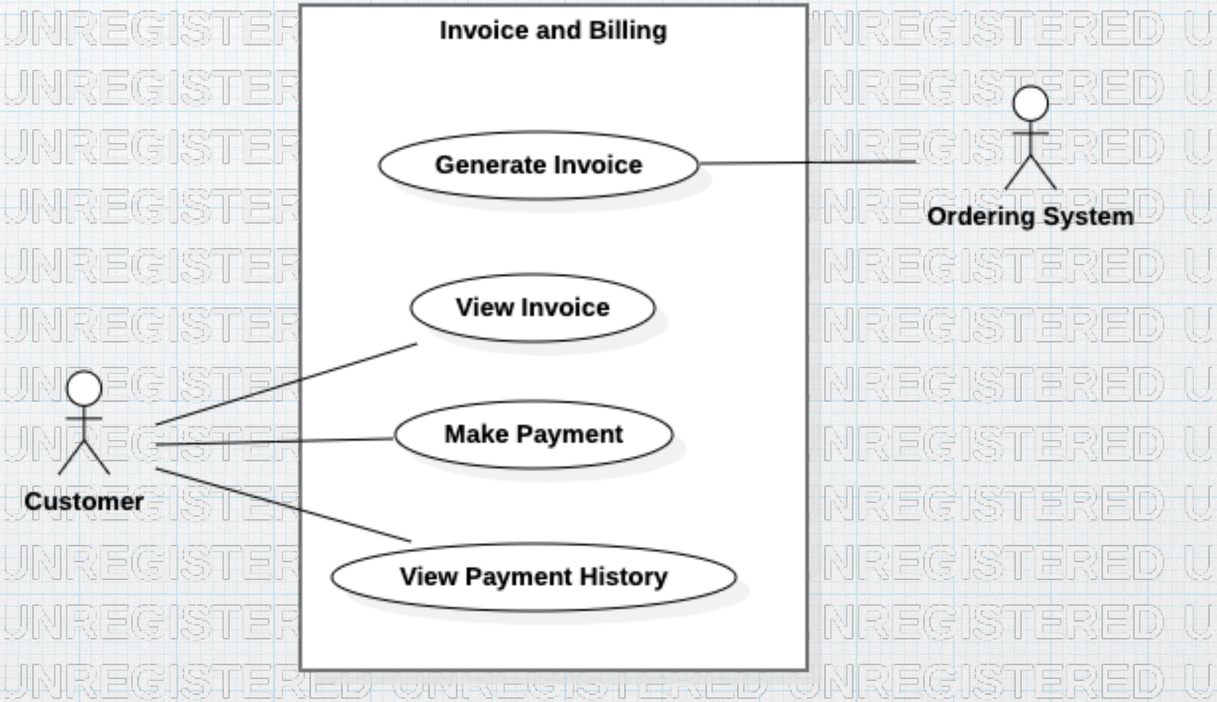
What makes an architecture "good"?

* Suitability - Does it enable the application to easily meet its requirements?
* Comprehensibility - High level, abstraction, encapsulation, etc. (SRP, ISP, DIP)
* Extensibility - How easy it is to add or modify functionality? (OCP, LSP, DIP)
* Testability - How easy is it to prove correctness? (LSP, ISP, DIP)
* Replaceability - How easy is it to replace a component or framework? (SRP, LSP, ISP, DIP).


cumulative
functionality
high internal quality
but delivers more rapidly
(and cheaply) later
software with high internal
quality gets a short initial
slow down
low internal quality
time
this point occurs in
weeks (not months)

What use cases do we need?

"A use case is a list of actions or event steps typically defining the interactions between a role and a system to achieve a goal."
-Wikipedia



First steps...

* Create a class for every use case

```
domain.usecase
  MakePaymentUsecase
  ViewInvoiceUsecase
  ...
```

Make Payment Use Case Analysis

Input:

* Invoice Number (`java.lang.String`), Payment (`domain.common.Money`)
* Payment Information
  * Payment Type (CC, PayPal, Bank, etc..)
  * Payment Parameters (`java.util.Properties`)

Output:

* Payment Confirmation

Invoice:

* Associated to a Customer, contains a list of LineItem

- `domain.common`
  - `PaymentInfo`
  - `PaymentType`
  - `Money`
  - `PaymentConfirmation`
- `domain.entity`
  - `Customer`
  - `Invoice`
  - `LineItem`
- `domain.usecase`
  - `MakePaymentUsecase`
  - `ViewInvoiceUsecase`

The Domain

Use cases and entities are the heart of our application.

The "common" classes support the business logic by moving data into and out of the domain.

Keep all of these classes "pure":

* No frameworks
* No database code
* No UI code

```
domain.common
    PaymentInfo
    PaymentType
    Money
    PaymentConfirmation
domain.entity
    Customer
    Invoice
    LineItem
domain.usecase
    MakePaymentUsecase
    ViewInvoiceUsecase
```

At this point we can select a use case and flesh it out with code.

We can use TDD to do this, or we can write a use case and then add tests.

We want full coverage of every business rule — it's the most important code in the application.

Make Payment Algorithm

1. Look up the invoice
2. If nothing is due, return immediately
3. Convert payment to USD
4. Request electronic payment
5. If payment is successful:
    1. Update and save the invoice
    2. Inform the customer of the payment details
6. Else: return failure to the customer

```
domain.common
    DomainException
    InvoiceNotFoundException
    Money
    PaymentConfirmation
    PaymentInfo
    PaymentReason
    PaymentType
domain.entity
    Customer
    Invoice
    LineItem
domain.port
    CurrencyExchange
    Invoices
    PaymentService
domain.usecase
    MakePaymentUsecase
    ViewInvoiceUsecase
```

Make Payment Algorithm

1. Look up the invoice
2. If nothing is due, return immediately
3. Convert payment to USD
4. Request electronic payment
5. If payment is successful:
    1. Update and save the invoice
    2. Inform the customer of the payment details
6. Else: return failure to the customer

CurrencyExchangeService.java

```java
package us.hypermediocrity.springclean.domain.port;

import java.util.Currency;

public interface CurrencyExchangeService {
  Money convert(Money amount, Currency usd);
}
```

PaymentService.java

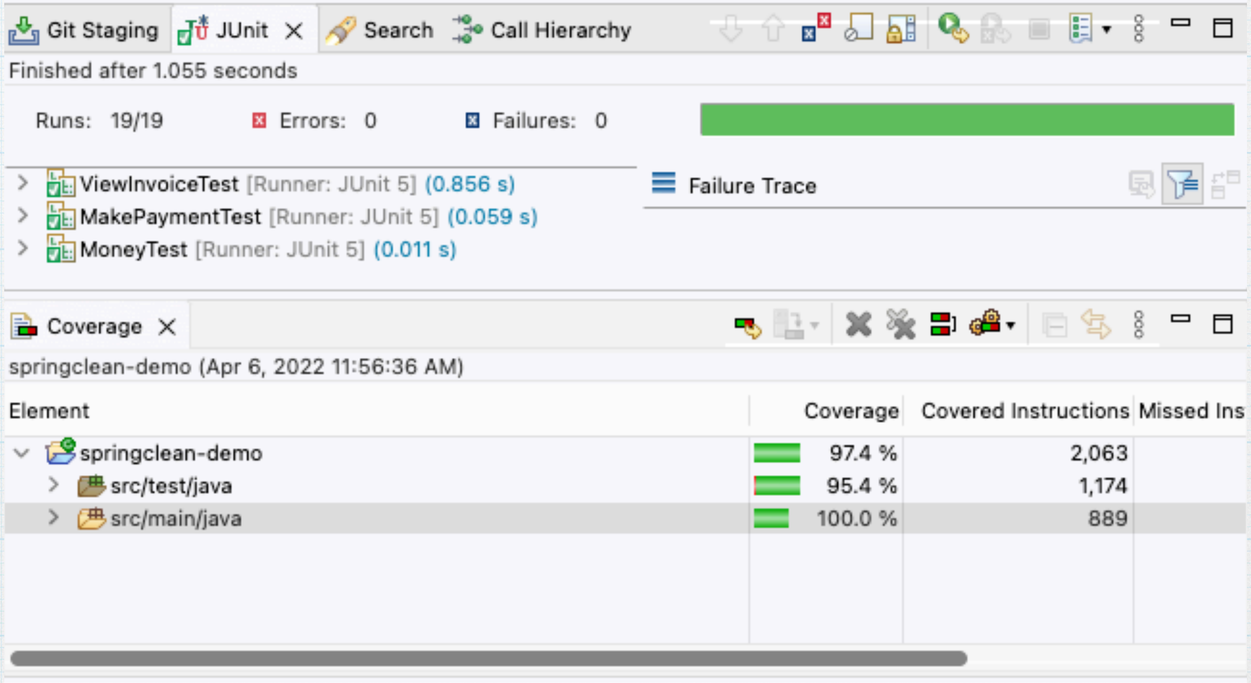
```java
package us.hypermediocrity.springclean.domain.port;

import us.hypermediocrity.springclean.domain.common.Money;
import us.hypermediocrity.springclean.domain.common.PaymentInfo;

public interface PaymentService {
  PaymentResponse request(Money adjusted, PaymentInfo paymentInfo);
}
```

Invoices.java

```java
package us.hypermediocrity.springclean.domain.port;

import java.util.Date;
import java.util.List;
import java.util.Optional;

import us.hypermediocrity.springclean.domain.entity.Invoice;

public interface Invoices {
  Optional<Invoice> getInvoice(String invoiceId);

  void update(Invoice invoice);

  List<Invoice> findInvoicesDueBefore(Date date);

  List<Invoice> findOpenInvoicesByCustomer(String customerId);

  void create(Invoice invoice);
}
```

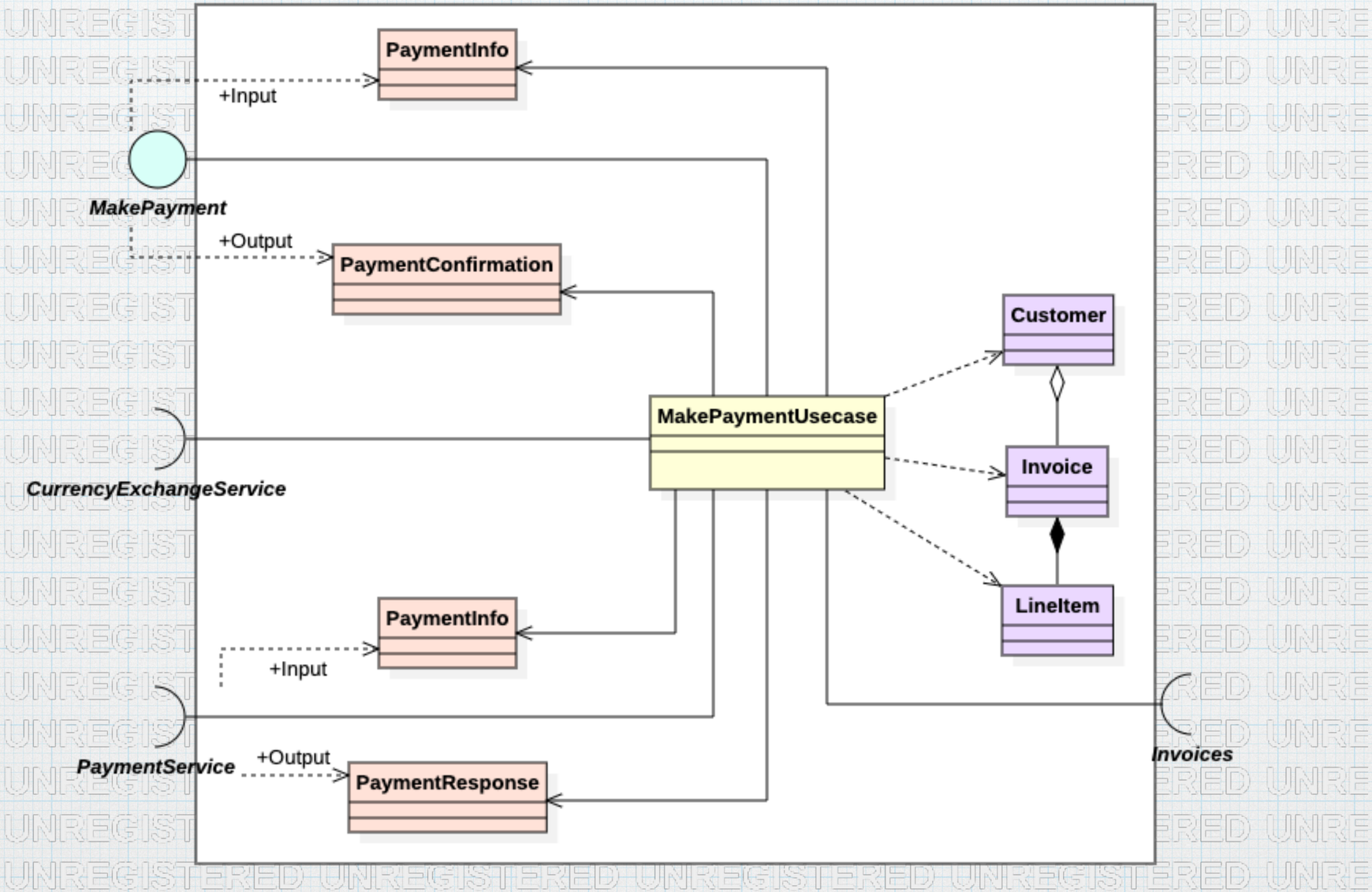
Get the domain under test

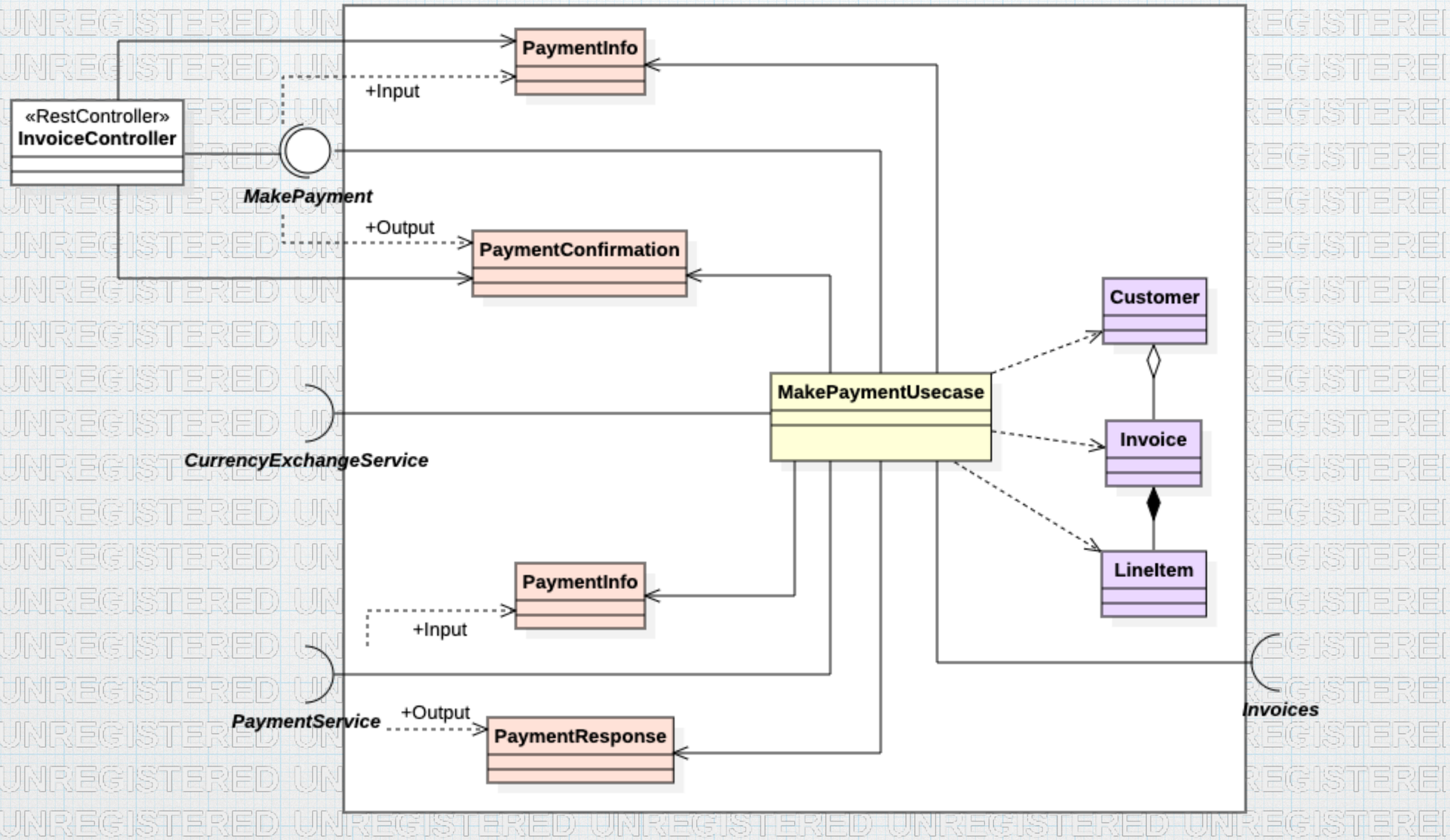
Because there are no external dependencies, frameworks, etc, we can get very high unit test code coverage and those tests run very fast.

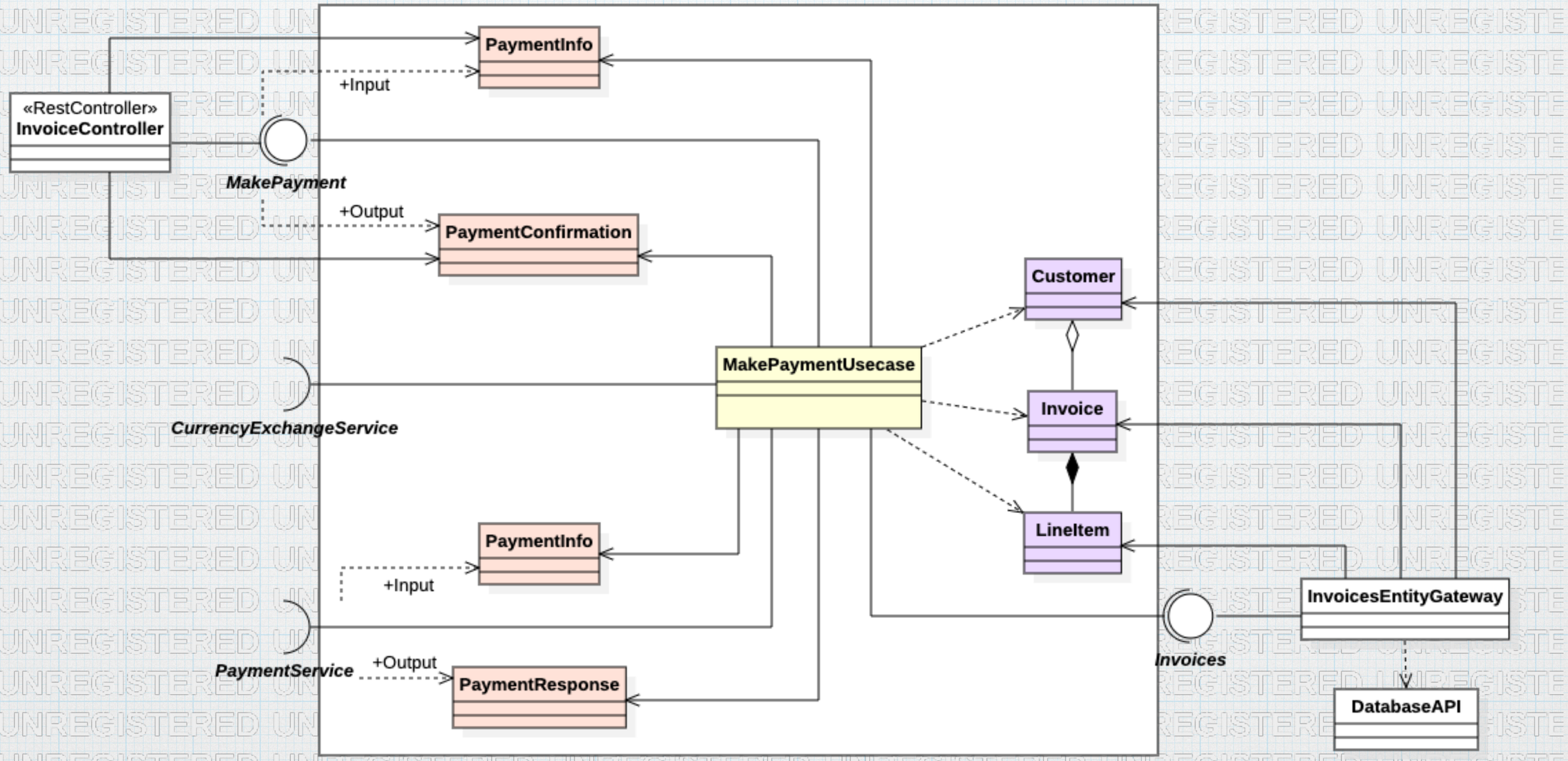


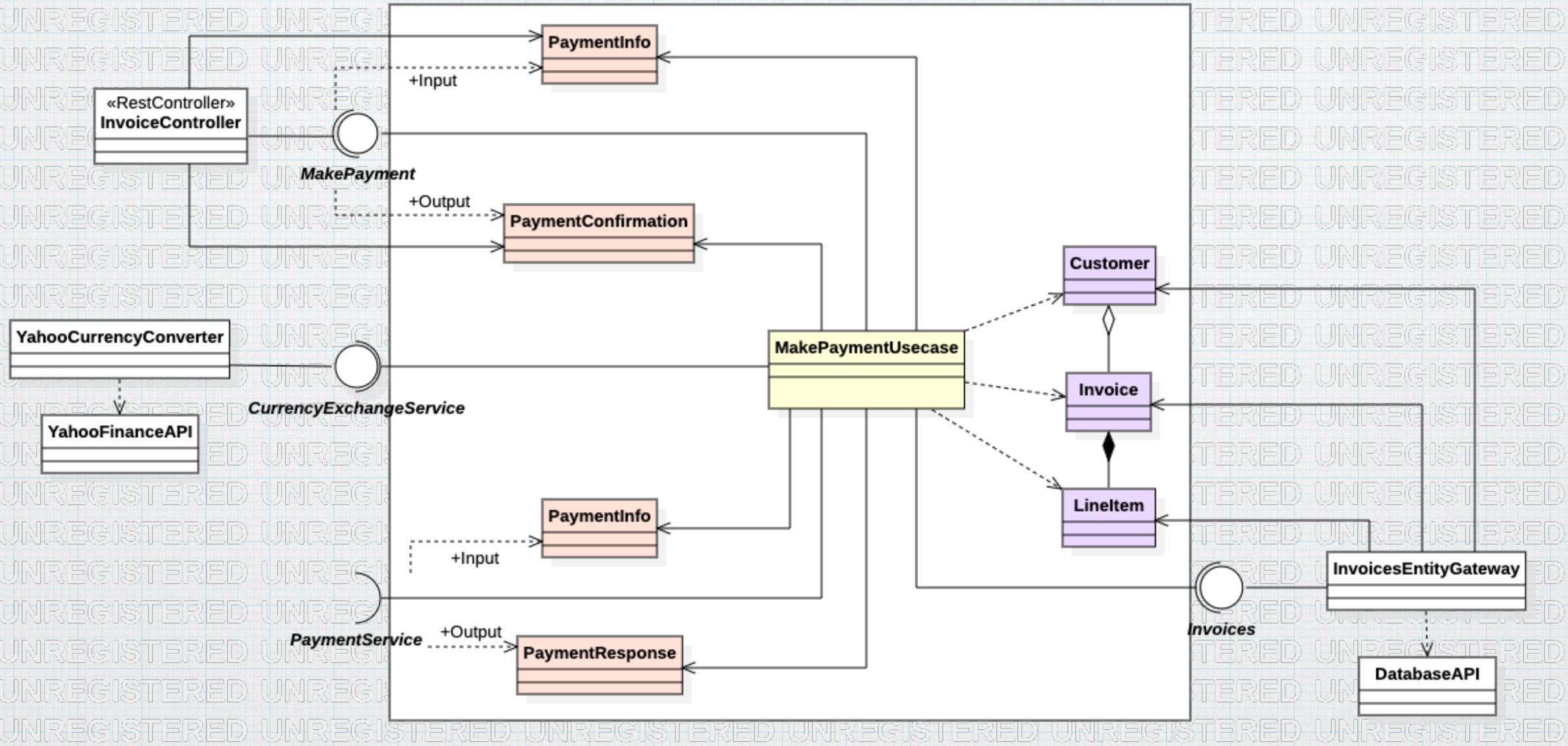
Finishing the domain

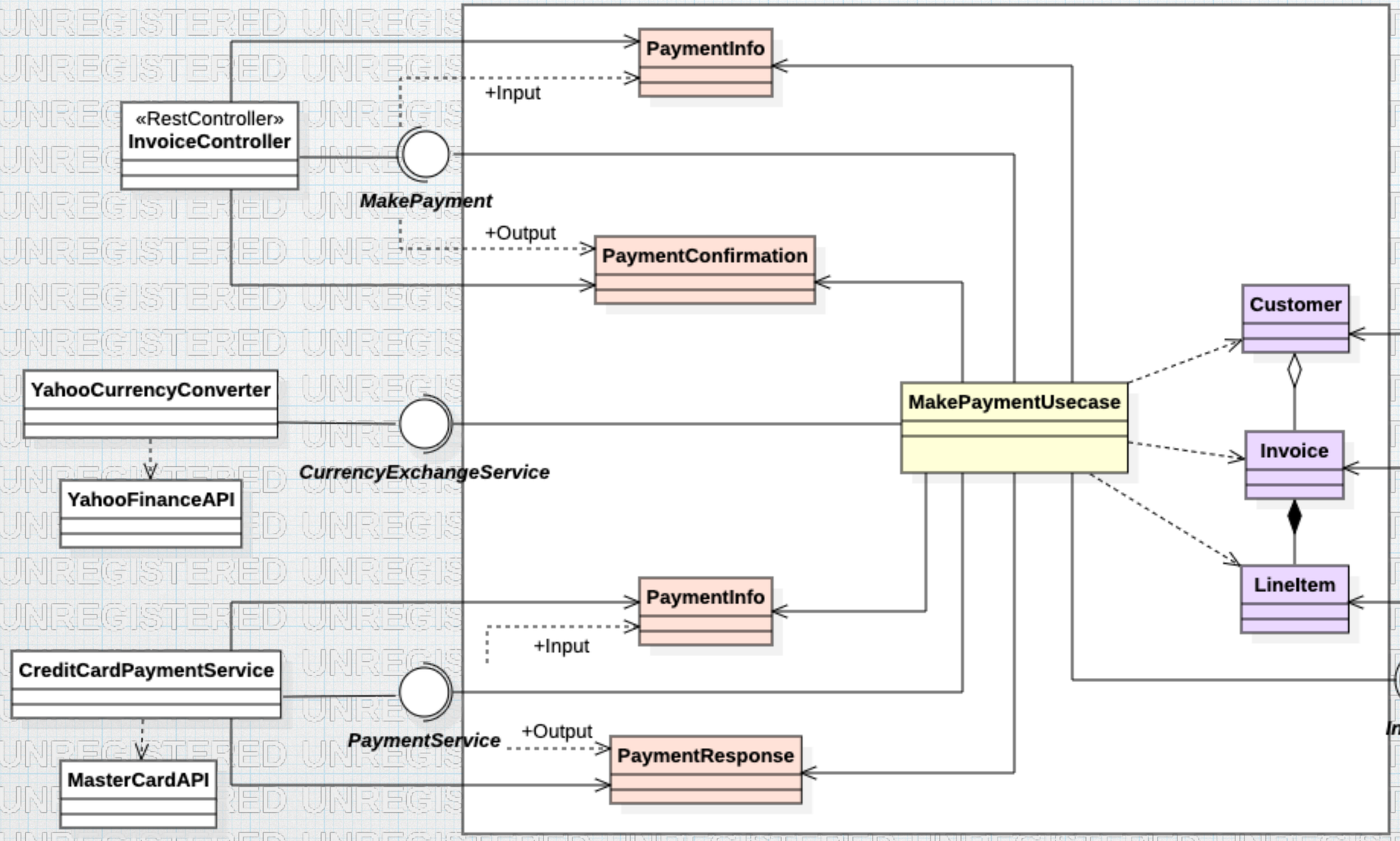
- Expose the use cases via interfaces for others to call.
- Expose the ports for others to extend.
- Isolate this code so that nobody can accidentally change or effect the logic.
  - ArchUnit, Jigsaw Modules, and/or Maven Modules


PaymentInfo
+Input
MakePayment
+Output
PaymentConfirmation
Customer
MakePaymentUsecase
Invoice
CurrencyExchangeService
LineItem
PaymentInfo
+Input
PaymentService
+Output
PaymentResponse
Invoices


PaymentInfo
+Input
«RestController»
InvoiceController
MakePayment
+Output
PaymentConfirmation
Customer
MakePaymentUsecase
Invoice
CurrencyExchangeService
LineItem
PaymentInfo
+Input
PaymentService
+Output
PaymentResponse
Invoices


PaymentInfo
+Input
«RestController»
InvoiceController
MakePayment
+Output
PaymentConfirmation
Customer
MakePaymentUsecase
Invoice
CurrencyExchangeService
LineItem
PaymentInfo
+Input
InvoicesEntityGateway
Invoices
PaymentService
+Output
PaymentResponse
DatabaseAPI


PaymentInfo
+Input
«RestController»
InvoiceController
MakePayment
+Output
PaymentConfirmation
Customer
YahooCurrencyConverter
MakePaymentUsecase
Invoice
CurrencyExchangeService
YahooFinanceAPI
LineItem
PaymentInfo
+Input
InvoicesEntityGateway
Invoices
PaymentService
+Output
PaymentResponse
DatabaseAPI



UNREGISTERED

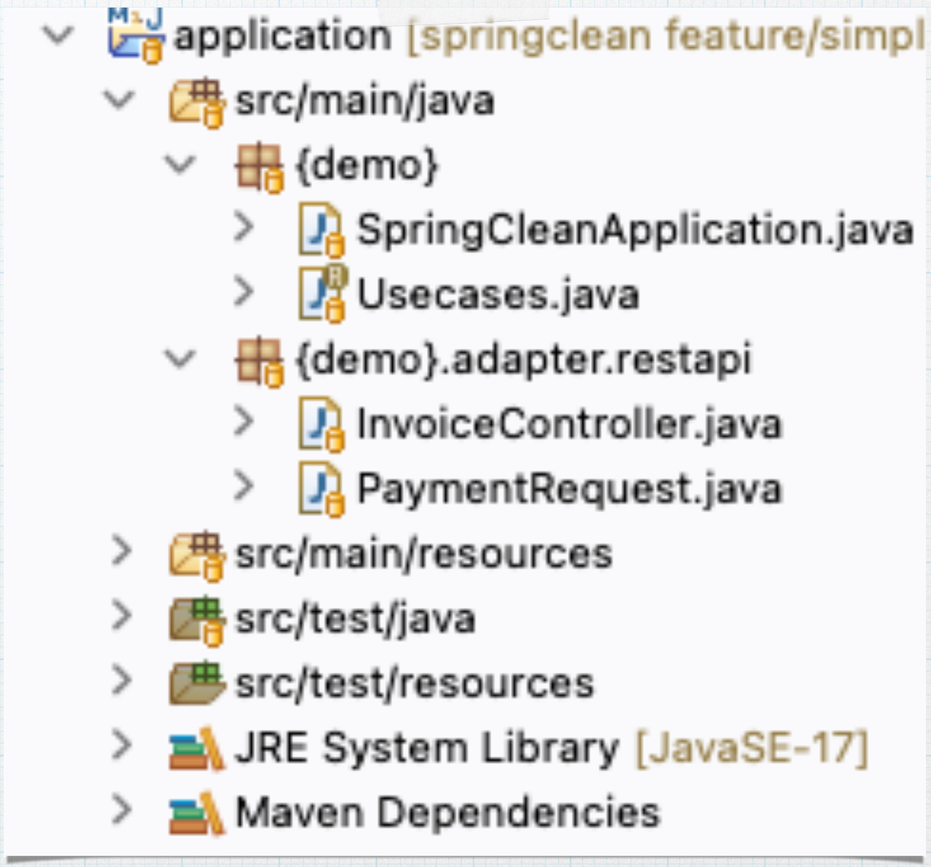
Integrate with SpringBoot

- Create a maven multi-module project:
  - A submodule for the domain (domain)
  - A SpringBoot submodule (application)
  - Configure a factory to return each use case with its dependencies satisfied

```java
public record Usecases(ViewInvoice viewInvoice, MakePayment makePayment) {}

@SpringBootApplication
public class SpringCleanApplication {
 public static void main(String[] args) {
  SpringApplication.run(SpringCleanApplication.class, args);
 }

 @Bean public Usecases usecases() {
  return new Usecases(viewInvoice(), makePayment()); }

 public MakePayment makePayment() {
  return new MakePaymentUsecase(invoices(), currencyExchange(), router()); }

 public ViewInvoice viewInvoice() {
  return new ViewInvoiceUsecase(invoices(), customers(), currencyExchange()); }

 @Bean public Invoices invoices() { return new InvoicesAdapter(); }
 @Bean public Customers customers() { return new CustomersAdapter(); }
 @Bean public CurrencyExchangeService currencyExchange() {
  return new CurrencyConverterAdapter(); }

 @Bean public PaymentService router() {
  return new PmtRouter(paypal(), bank(), card());
 }
}
```

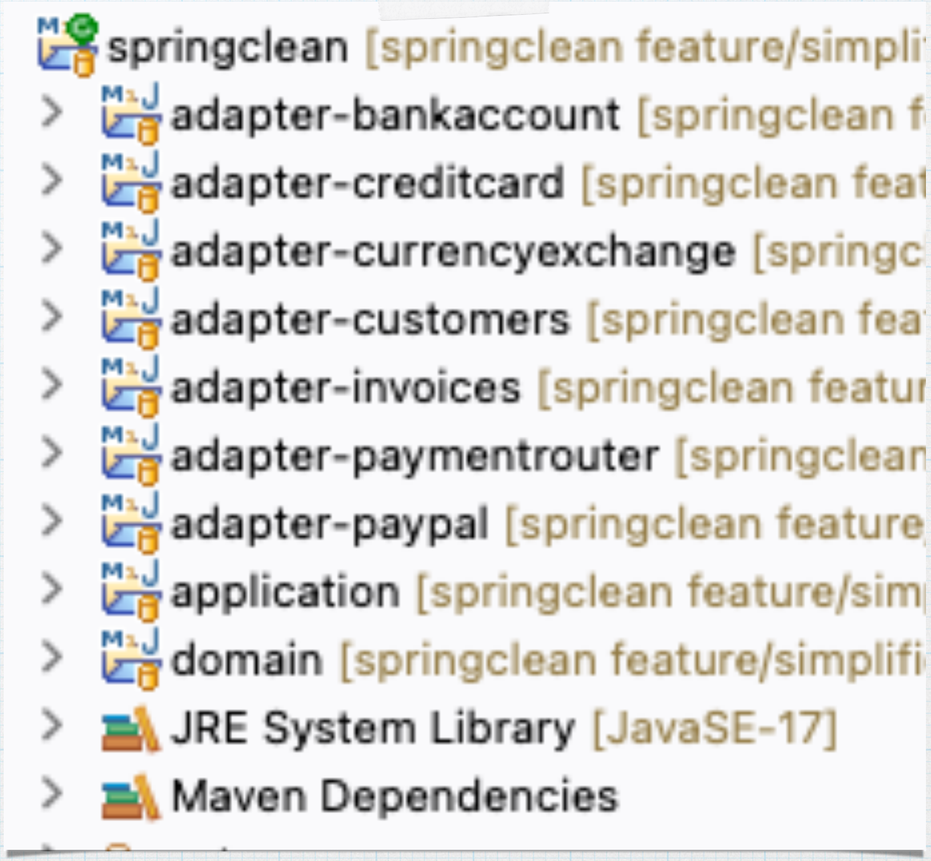
Decide how to implement ports

* "Primary" ports (interfaces that call the use cases) I defined as a part of my SpringBoot application

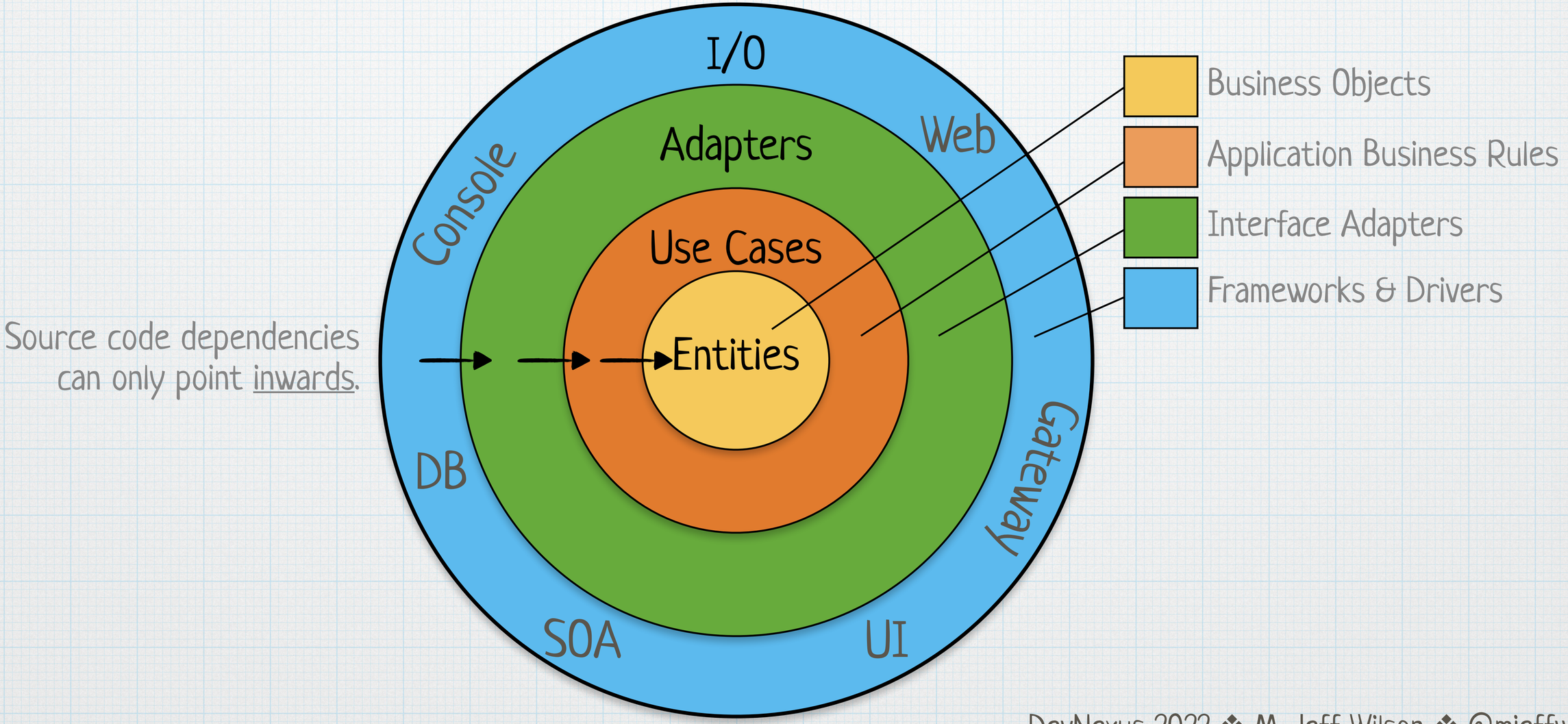


Decide how to implement ports

* "Secondary" ports — interfaces that are called by a use case — I defined in its own maven submodule.



The Resulting (Clean) Architecture



Clean Architecture Rules

* The Domain must not depend on any frameworks, including Spring and SpringBoot itself.
* The Domain must not depend on code outside of the Domain package.
* Domain Entities have no other dependencies even to other code inside the Domain package.
* Use Cases depend upon Entities and Ports, and their associated data payload.
* With one exception, the data passed between the domain and the adapters are always "primitives" (int, double, boolean, String, Date, Money, etc.) or simple DTOs or Java 17 records. They just carry data, and do not have any significant behavior or methods.
    * The exception is a special port/adapter that retrieves Entities from some form of persistent storage. These are called Entity Gateways.
* Adapters do not depend upon each other.

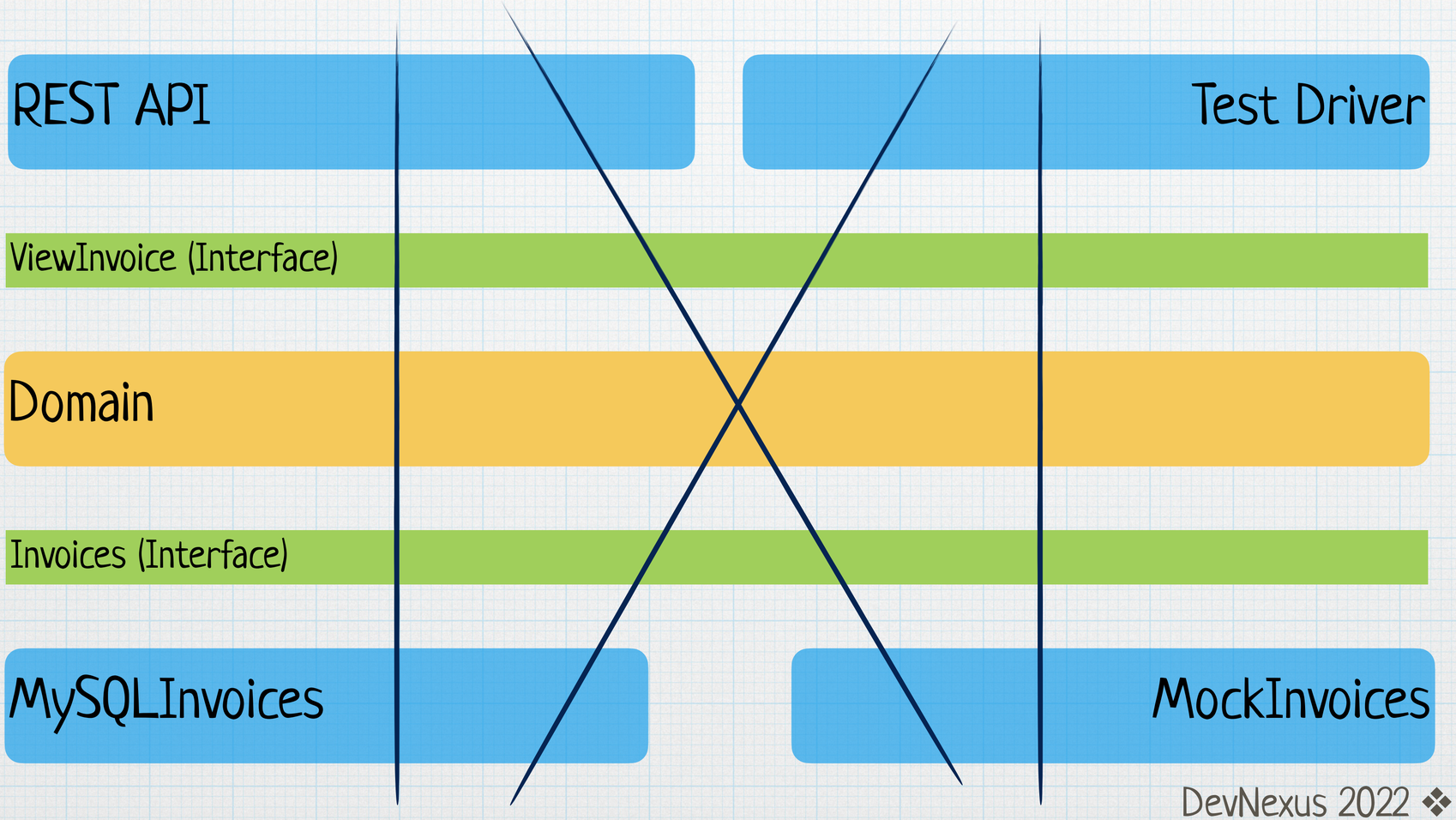
What are the tradeoffs?

* Data translation between domain and adapters
* Since adapters cannot depend on each other, there may be missed opportunities for reuse.
* Cannot take advantage of frameworks and libraries in the domain.
* Potential context switching — if the adapters and domains are in completely separate projects.

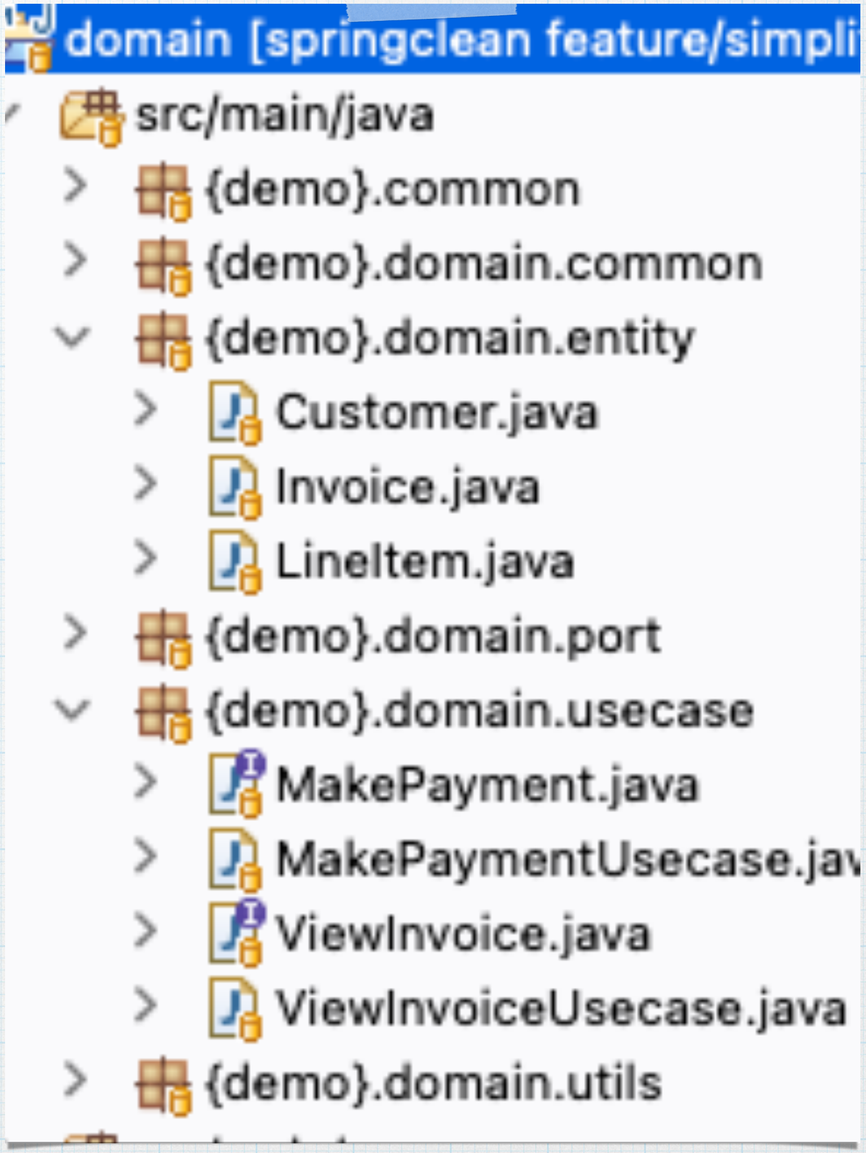
Maintaining the Architecture

```
@AnalyzeClasses(packages = "us.hypermediocrity.springclean")
class SpringCleanArchitectureTest {
  // No cycles between packages are allowed.
  @ArchTest static final ArchRule noCycles = slices().matching("..springclean.(*)..").should().beFreeOfCycles();

  // Package [common] contains types or utils that other classes can use, but depend on nothing.
  @ArchTest static final ArchRule commonDependsOnNothing = noClasses().that().resideInAPackage("..springclean.common..")
      .should().dependOnClassesThat().resideOutsideOfPackages("java..", "..springclean.common..");

  // No class in [domain] should depend on anything outside of those packages.
  @ArchTest static final ArchRule domainOnlyDependsOnCommon = noClasses().that().resideInAPackage("..springclean.domain..")
      .should().dependOnClassesThat().resideOutsideOfPackages("java..", "..springclean.common..", "..springclean.domain..");

  // Adapters should not depend on each other, just [domain].
  @ArchTest static final ArchRule adaptersCannotDependOnEachOther = slices().matching("..springclean.adapter.(*)..")
      .should().notDependOnEachOther();

  // Package [domain.entity] depends only on [common].
  @ArchTest static final ArchRule domainEntityOnlyDependsOnDomainCommon = noClasses().that()
      .resideInAPackage("..springclean.domain.entity..").should().dependOnClassesThat()
      .resideOutsideOfPackages("java..", "..springclean.common..", “..springclean.domain.entity..");

  // Package [domain.port] depends only on [domain.entity].
  @ArchTest static final ArchRule domainPortOnlyDependsOnEntity = noClasses().that().resideInAPackage("..springclean.domain.port..")
      .should().dependOnClassesThat().resideOutsideOfPackages(“java..", "..springclean.common..", “..springclean.domain.entity..",
        "..springclean.domain.port..");
}
```

Testability

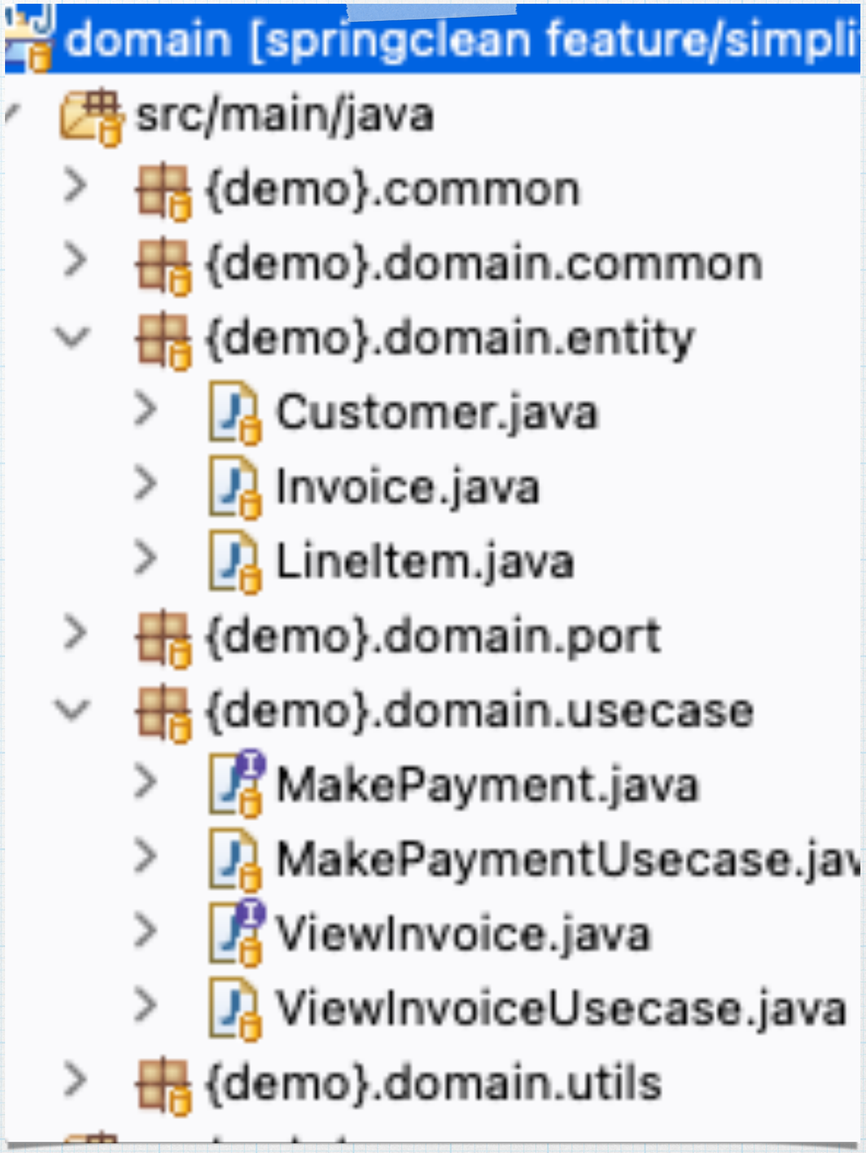


But what does it do?



Your first user story...

"As CFO, I want to give my customers a 10% discount on future orders if they pay within 10 days of the invoice date, so that we get their money earlier."



Source code: https://github.com/mjeffw/springclean

References:

* https://alistair.cockburn.us/hexagonal-architecture/
* https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Questions?

geek.macbeer@gmail.com
@mjeffw
https://linkedin.com/in/mjeffwilson