Clean Architecture with Django

Rethinking basic assumptions

Paul Wolf

paul.wolf@yew.io

https://github.com/paul-wolf

Can we build our Django applications using the principles of a Clean Architecture?

Clean Architecture: A Craftsman's Guide to Software Structure and Design

by Robert C. Martin

Typical Django Applications

- *Web* applications
- They have a relational database and views on that data with some transactional logic in between
- However powerful the other features are, they are pretty much a set of conveniences: admin, user and authentication framework, data migrations, middleware, management commands, etc.

Why Clean Architecture (CA)?

CA is intended to reduce the cost of change

- Easier and faster to build new features
- More independence of solution components meaning less refactoring if you change something
- Improved testability leading to better quality software
- Less cost overhead for software development

Arrow of Dependency

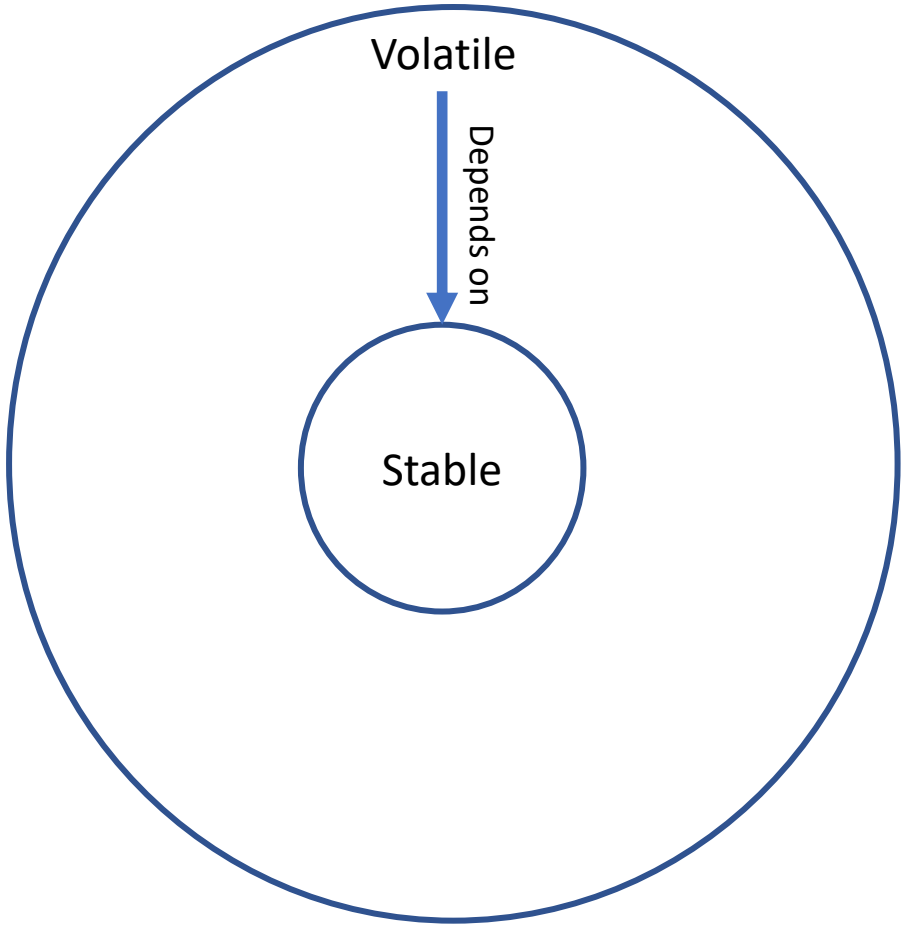
"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend on details. Details should depend on abstractions."

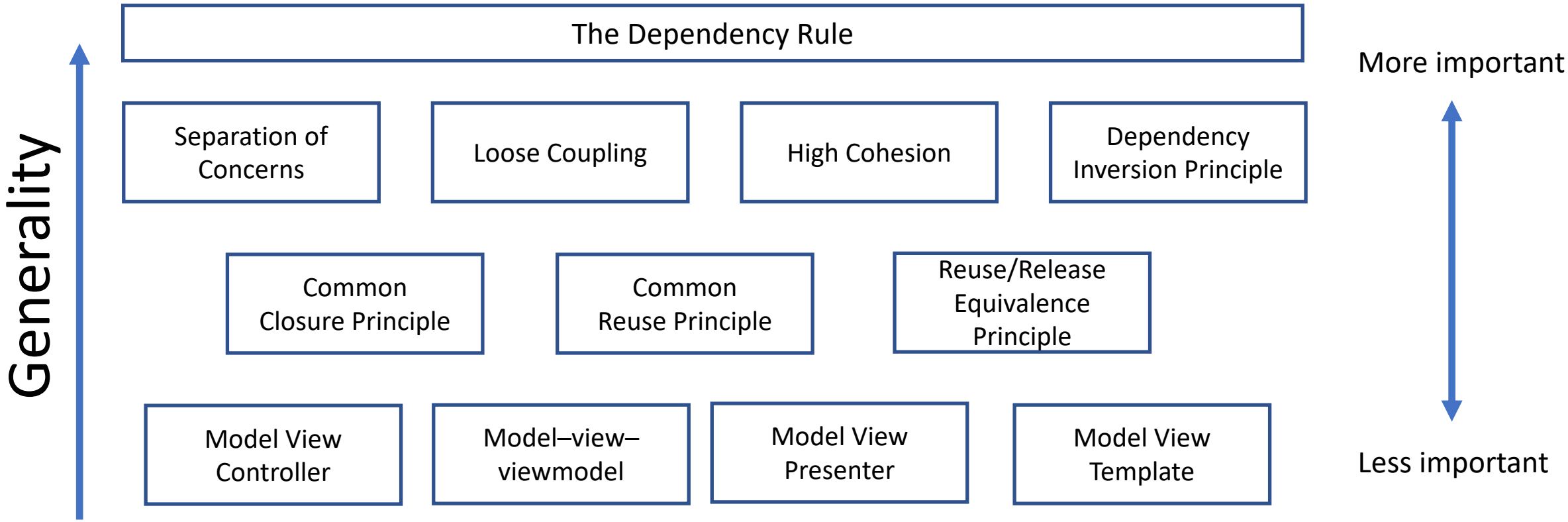
R.Martin

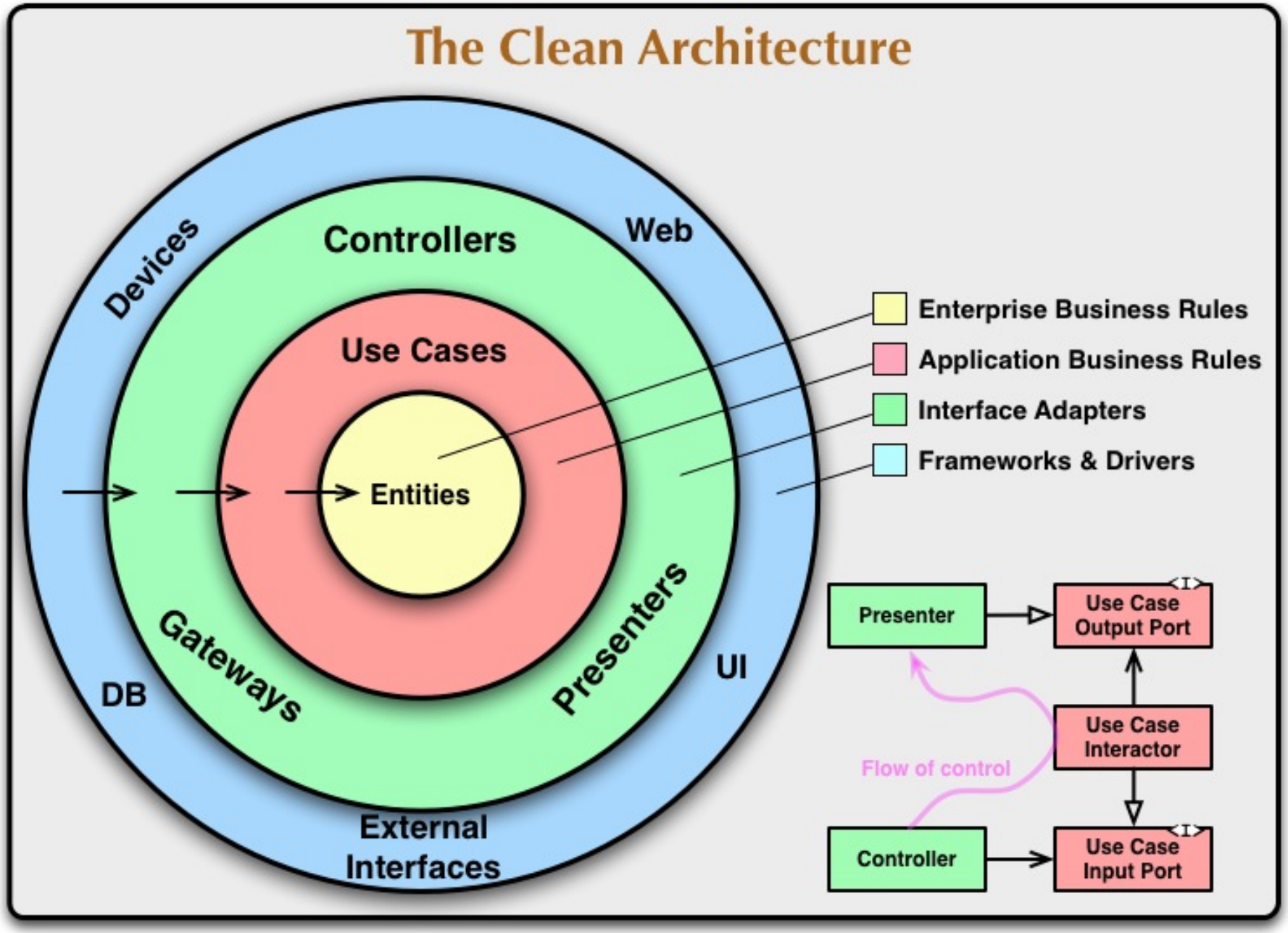
Clean Architecture is a pattern for making change easier to manage

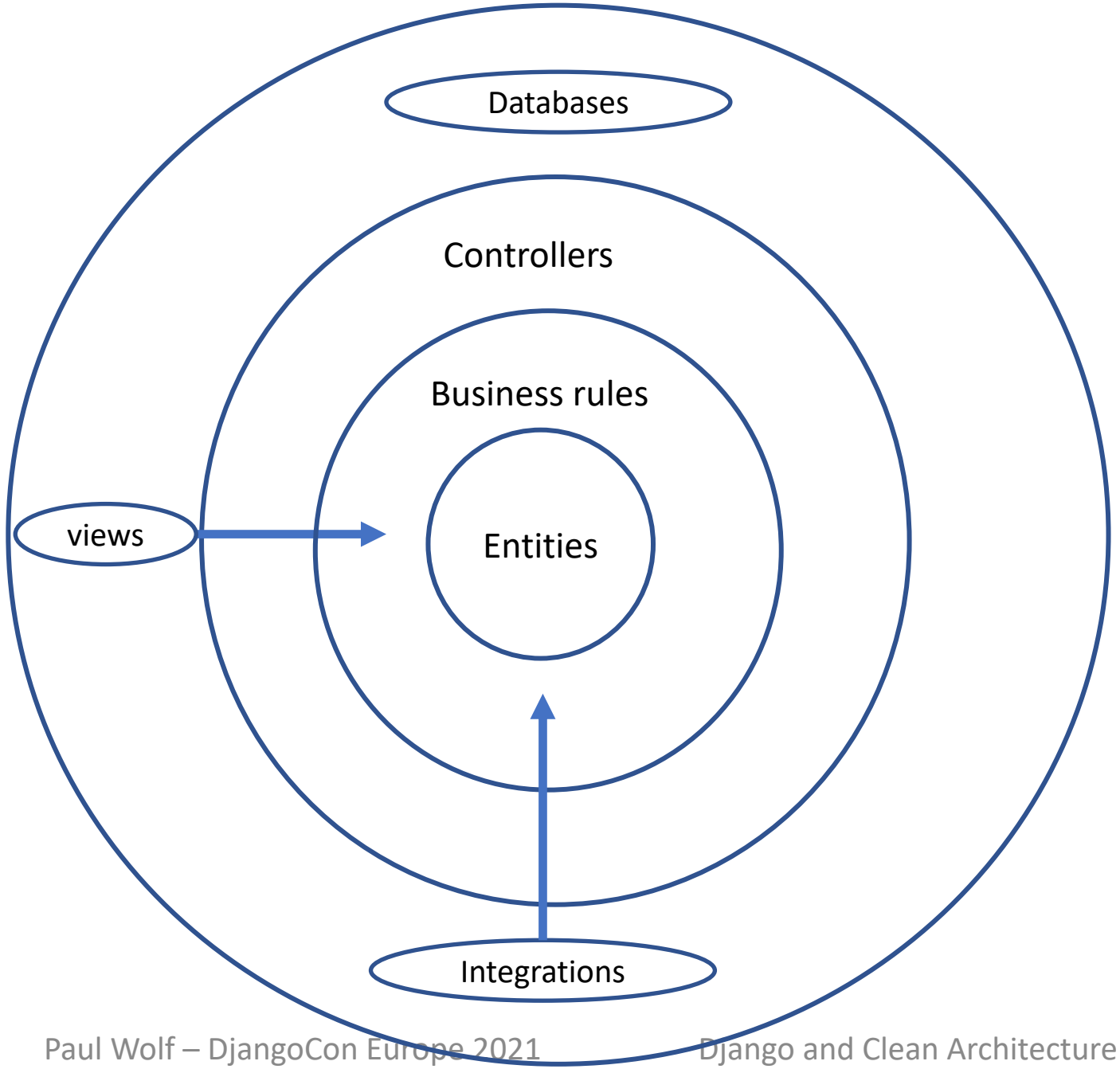
Clean Architecture



Volatile code should depend on less volatile code


The Dependency Rule
Generality
More important
Less important
Separation of Concerns
Loose Coupling
High Cohesion
Dependency Inversion Principle
Common Closure Principle
Common Reuse Principle
Reuse/Release Equivalence Principle
Model View Controller
Model–view–viewmodel
Model View Presenter
Model View Template


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port



Details

- Frameworks
- UI mechanics
- Databases
- Integrations
- IO mechanisms

Clean Architecture Biases

- Architecture rather than procedure oriented
- Cost is front-loaded. Higher upfront investment in design is assumed
- It favours languages that have strong abstraction features: interfaces

CA “likes” C++ and Java

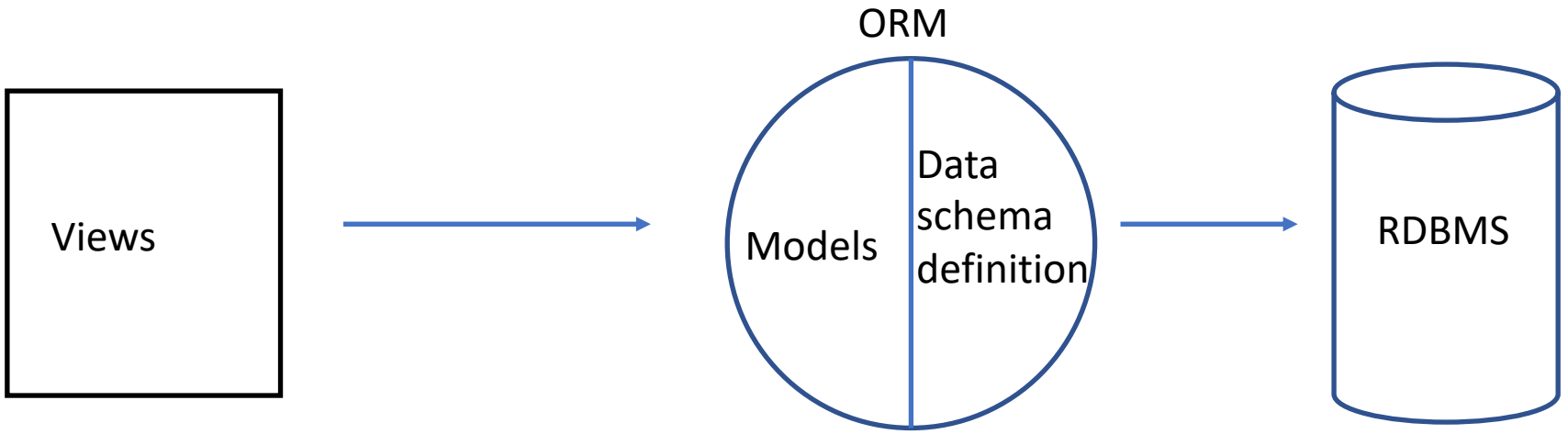
Frameworks

“Architectures are not (or should not) be about frameworks. Architectures should not be supplied by frameworks. Frameworks are tools to be used, not architectures to be conformed to. If your architecture is based on frameworks, then it cannot be based on your use cases.”

https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

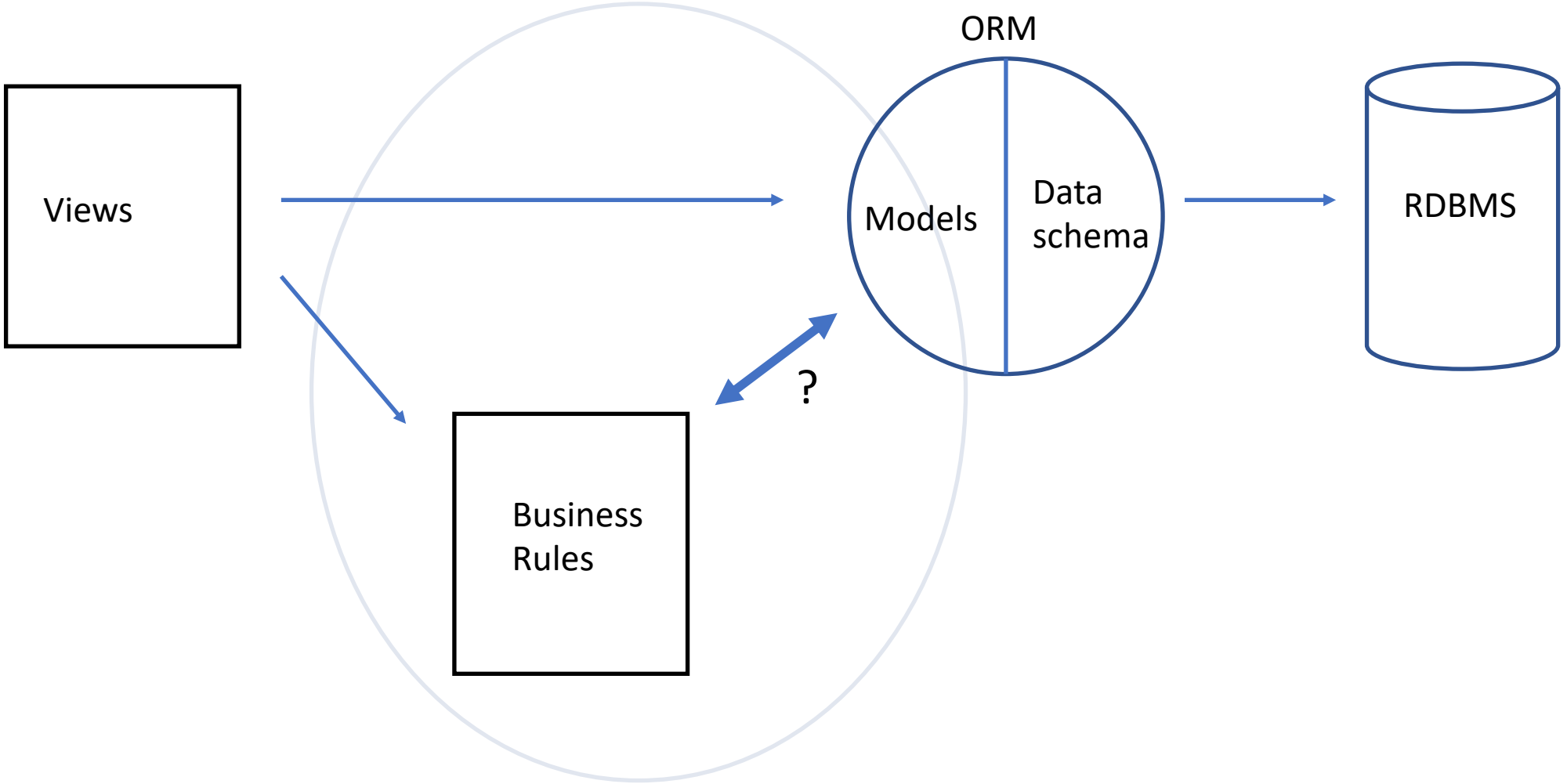
- Interfaces are abstractions
- Most dependencies should be on Interfaces
- In Python
  - Abstract classes
  - Duck typing
  - Factories
  - Dependency Injection Libraries

Best use-case for Django



Django works best when the models have a representation that is very close to the representation that views present to users

Here's where it gets tricky



Dependency Inversion (DI)

DI is a method of turning the arrow of dependency in a different direction to what it would otherwise be to ensure that the arrow points towards entities and business rules, not to details

One way to achieve Dependency ***Inversion*** is to use the Dependency ***Injection*** Pattern

- https://martinfowler.com/articles/injection.html
- https://github.com/ets-labs/python-dependency-injector

Dependency Injector is a dependency injection framework for Python.

“Separating configuration from use”

Interface Adapters

Interface Adapters are for going from one level of abstraction to another without breaking the dependency rule

“Keep dependencies pointing in the direction of less volatile, less detail oriented code”

(R. Martin)

Django Object Relational Mapping

- CRUD is natural and ready-made
- Conceptual simplicity
- Very fast setup
- Migrations are a snap (with conditions attached)

The Django ORM does heavy lifting

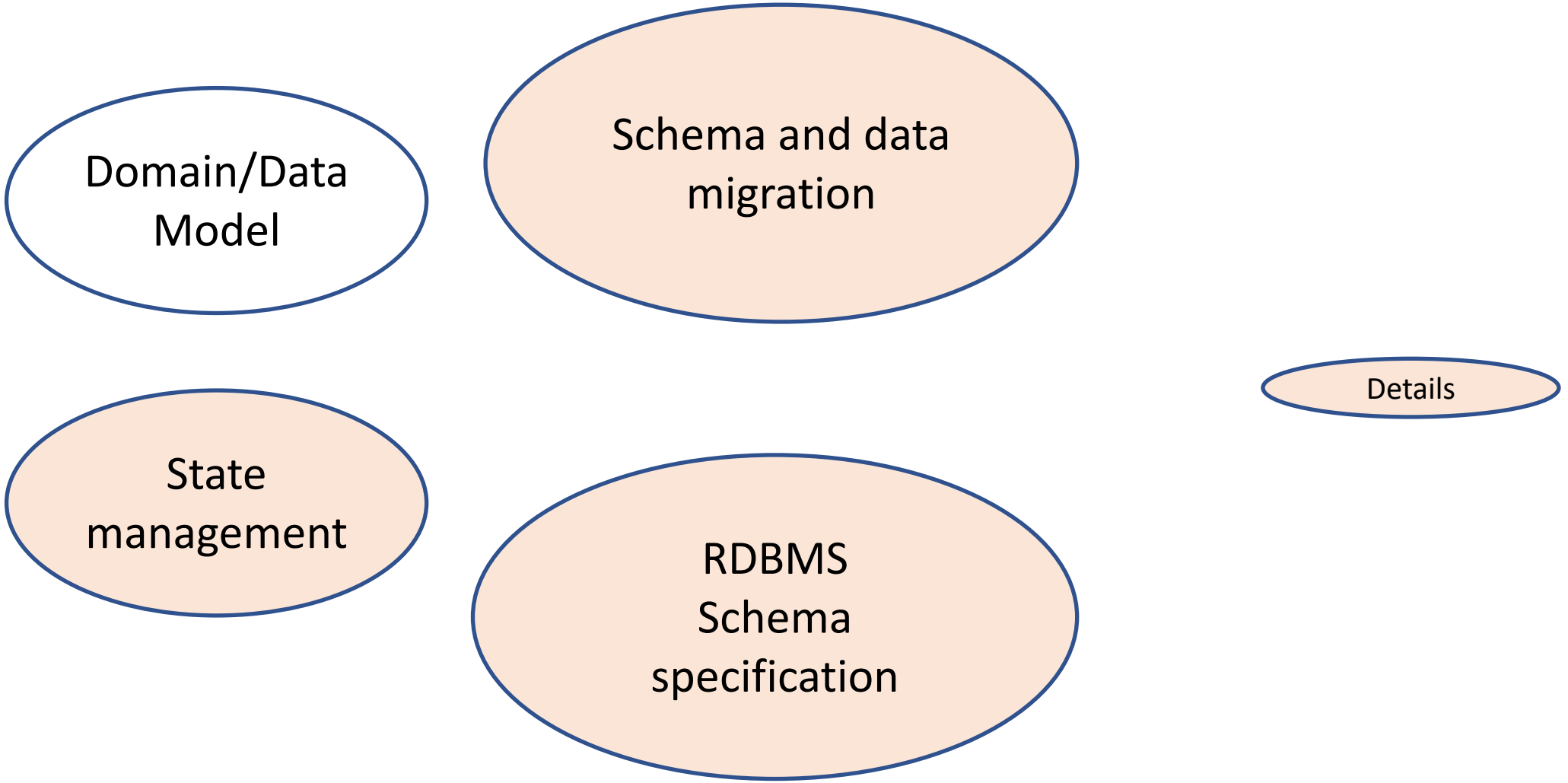
- Domain modelling
- Database connection management
- Object state management
- Caching

Details are now intermingled with abstractions about the business domain

“There is no such thing as an ORM”

R.Martin

ORM Responsibilities



Data schema and business domain definition

```
class Asset(models.Model):
    asset_type = models.IntegerField(choices=ASSET_TYPES)
    owner = models.ForeignKey(Owner)
    ...

    class Meta:
        abstract = True

class AssetPrint(Asset):
    ...

class AssetDigital(Asset):
    ...
```

We are defining details, a data schema; you are deciding to use an RDBMS

```
class Asset(models.Model):
    asset_type = models.IntegerField(choices=ASSET_TYPES)
    owner = models.ForeignKey(Owner)
    ...


class AssetPrint(models.Model):
    asset = models.ForeignKey(Asset)
    ...

class AssetDigital(models.Model):
    asset = models.ForeignKey(Asset)
    ...
```

We are also defining the business domain

Object Relational Mapping Limitations

- A set of models is a data representation valid for specific use cases; other valid representations may be possible
- The query interface starts to get awkward
- Vectorised operations
- Geospatial operations
- CQRS

Context Freedom

A variable is context-free if it has minimal dependencies

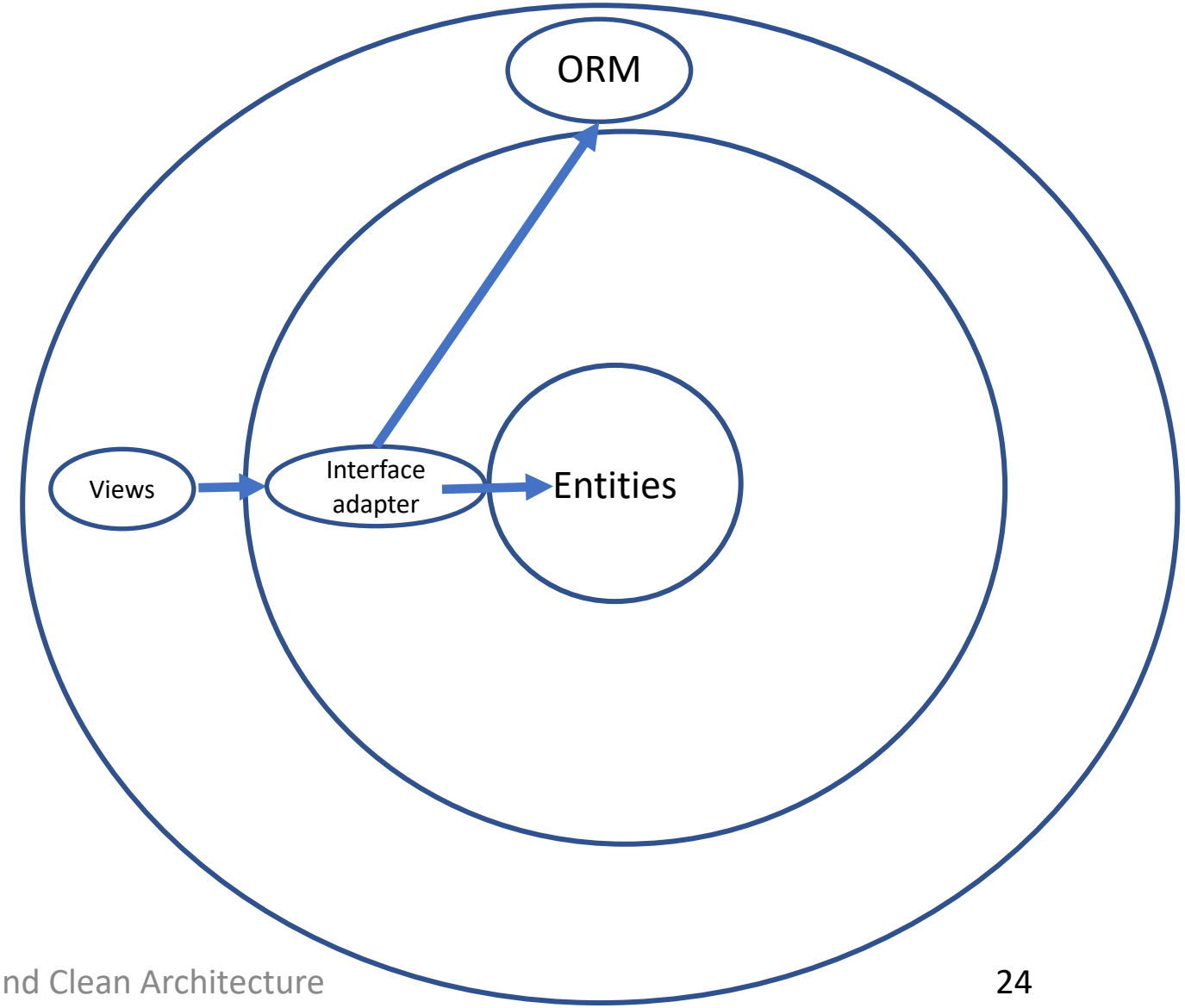
- No integration state
- No framework dependency
- Easy to create

We could do this

Have an interface adapter that returns only Python dataclasses or dicts, tuples, etc.

Now views only ever depend on the business domain

Undermines the Django framework



The Django ORM Use-Case Principle

The data representation in the view layer is similar to the data representation in the model layer.

The Django ORM and framework works extremely well within the intended use case

- Excellent performance
- Easy to use
- Great experience!

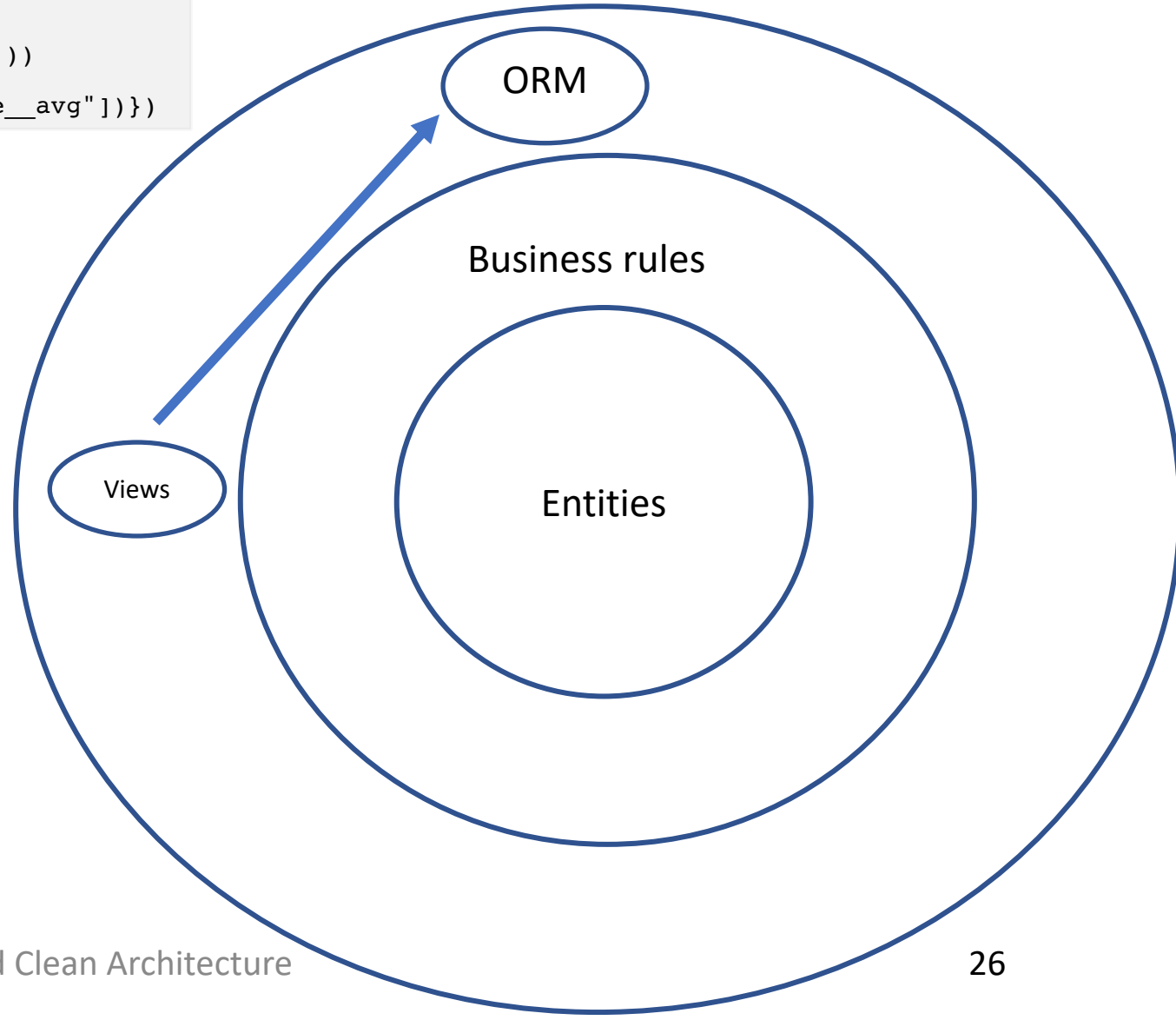
The main Django Use-Case

```
def books_average_price(request):
    price_data = Book.objects.all().aggregate(Avg('price'))
    return JsonResponse({"price": float(price_data["price__avg"])})
```

Call another detail layer code module, the ORM

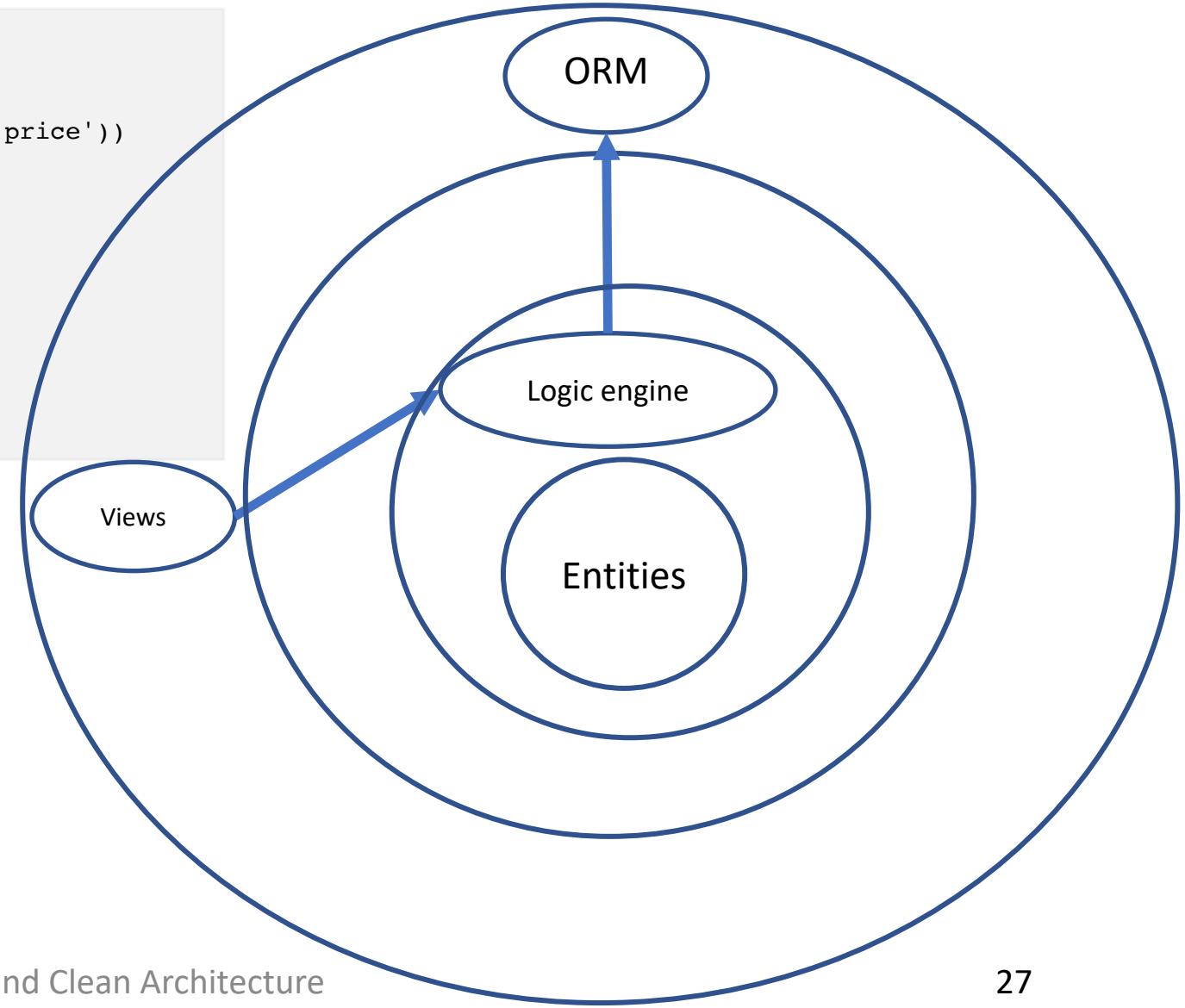
Not very Clean Architecture

But it works really well, given some assumptions



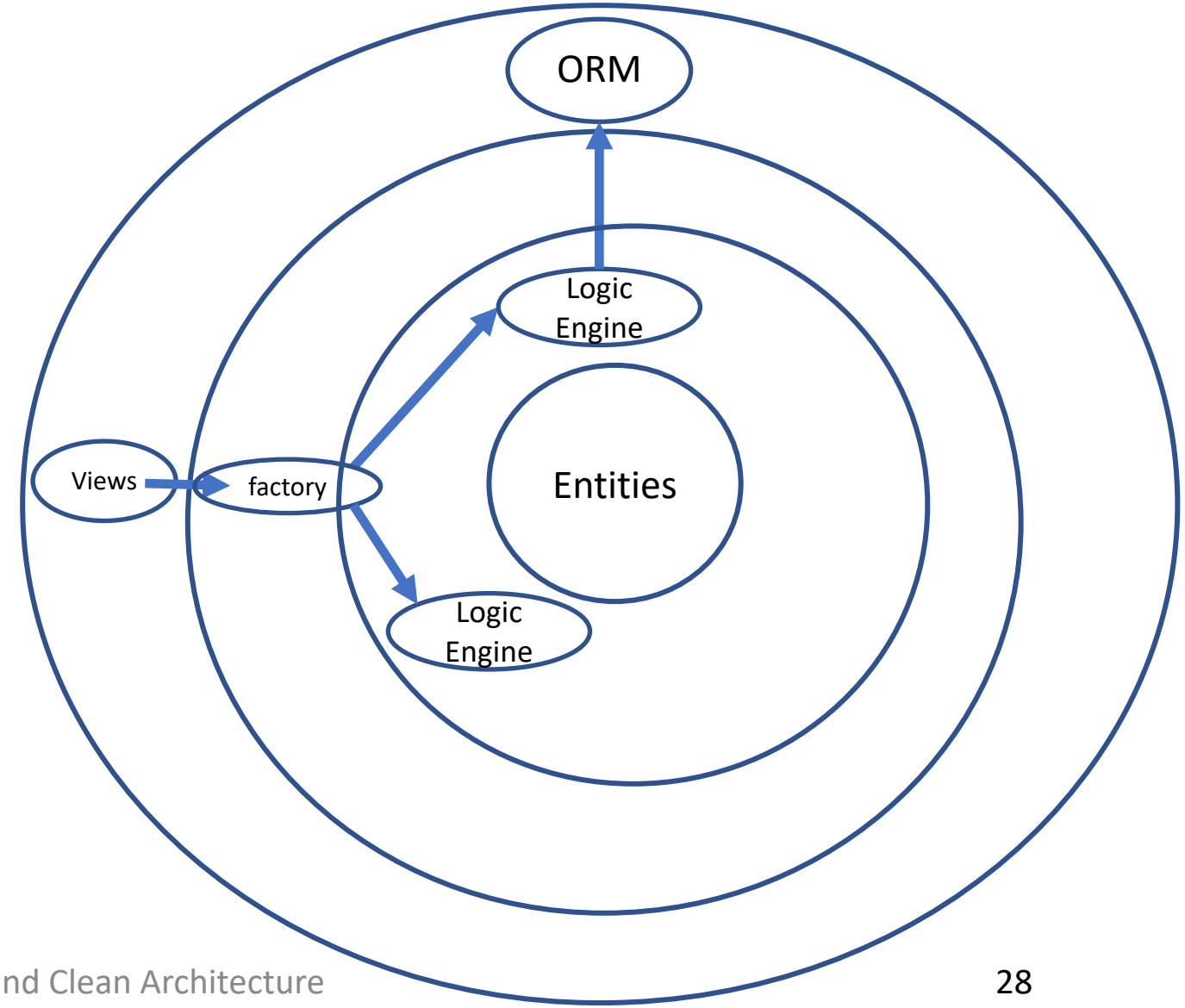
```
class PriceEngine:
    def get_average_price(self):
        price_data = Book.objects.all().aggregate(Avg('price'))
        return float(price_data["price__avg"])

def books_average_price(request):
    price_engine = PriceEngine()
    price = price_engine.get_average_price()
    return JsonResponse({"price": price})
```



Call a business rule module

```
def is_summer():
    return datetime.datetime.now().month in (6, 7, 8)
class PriceEngine(ABC):
    @abstractmethod
    def get_average_price(self):
        pass
class PriceEngineGeneral(PriceEngine):
    def get_average_price(self):
        price_data = Book.objects.all().aggregate(Avg('price'))
        return float(price_data["price__avg"])
class PriceEngineDiscount(PriceEngine):
    def __init__(self, discount):
        self.discount = discount or 0.8
    def get_average_price(self):
        price_data = Book.objects.all().aggregate(Avg('price'))
        return float(price_data["price__avg"]) * self.discount
def price_engine_factory(discount=None) -> PriceEngine:
    if is_summer():
        return PriceEngineDiscount(discount)
    return PriceEngineGeneral()
def books_average_price(request):
    price_engine = price_engine_factory()
    price = price_engine.get_average_price()
    return JsonResponse({"price": price})
```



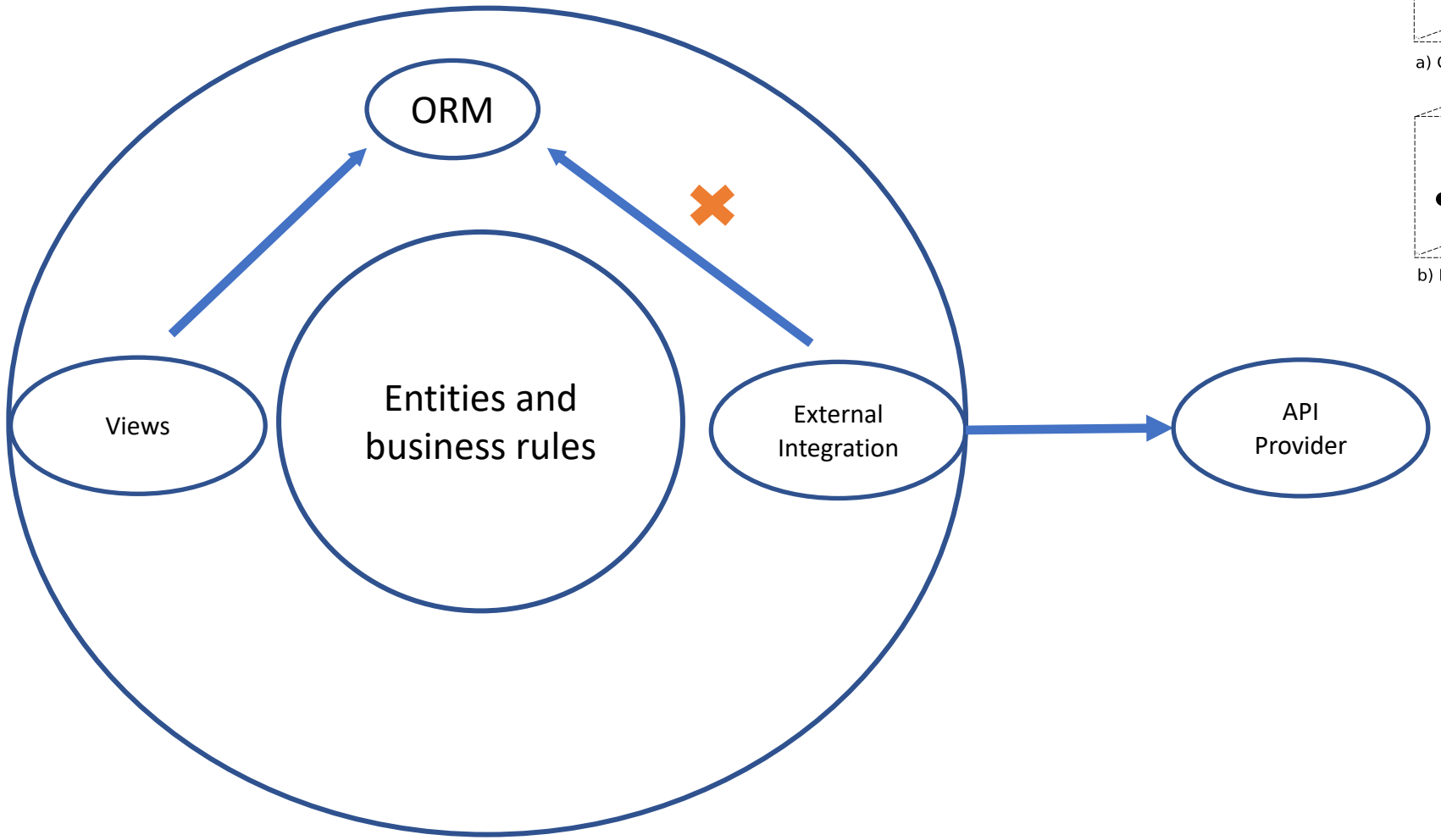
Cross detail dependencies: Example Reservation

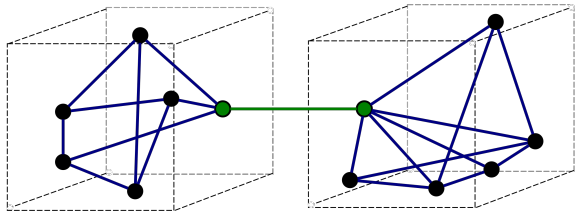
Call an external API that has start/end dates for a reservation using the Reservation Model instance

But the remote API only requires start/end dates

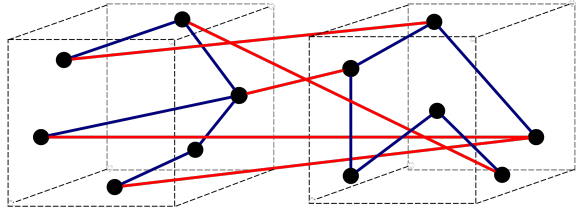
We can’t call the integration anymore without setting up a persistent object that might have nothing to do with our call to the API

Cross detail dependencies





a) Good (loose coupling, high cohesion)



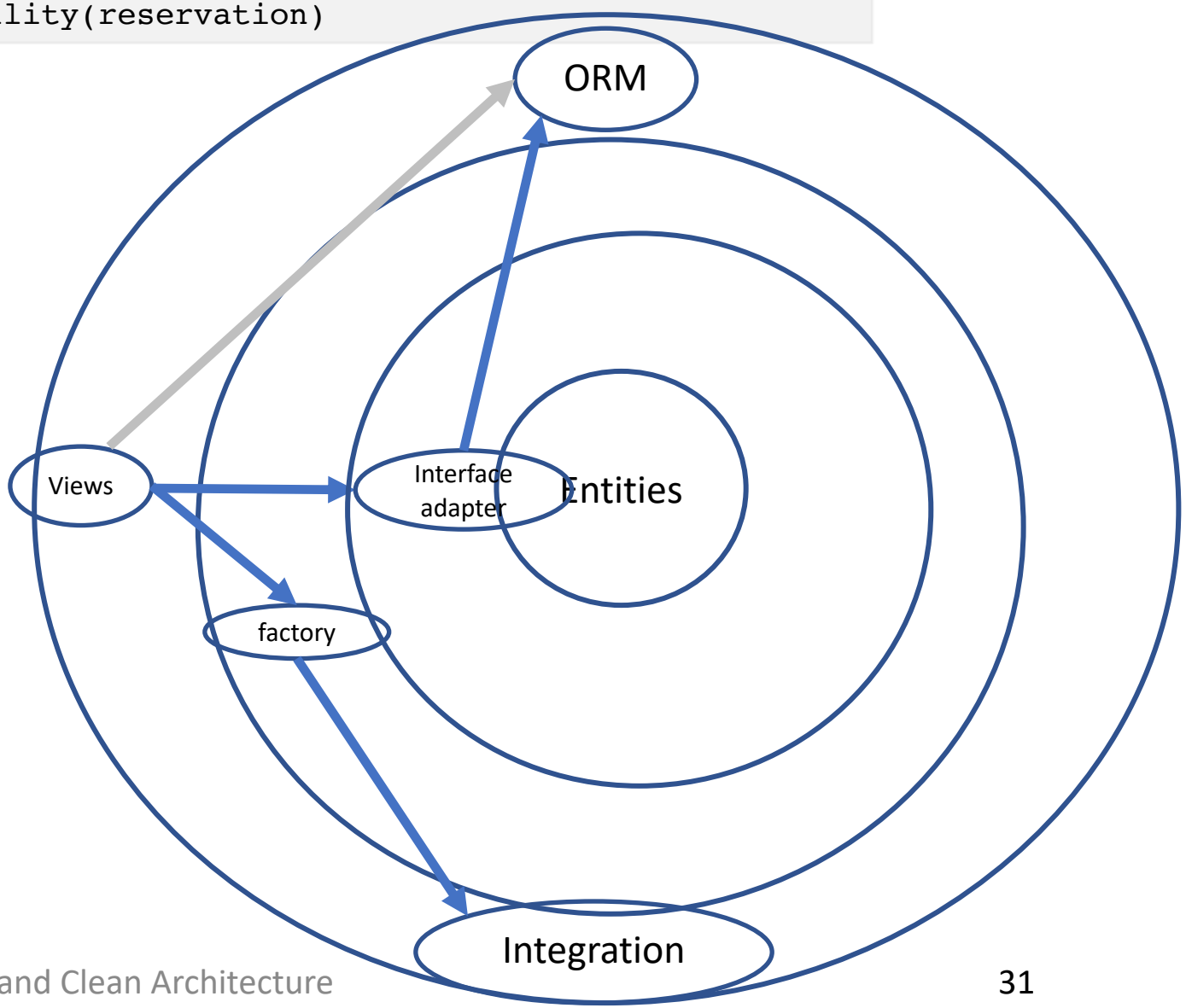
b) Bad (high coupling, low cohesion)

https://en.wikipedia.org/wiki/Loose_coupling

```
reservation_service = reservation_service_factory()
reservation = make_context_free_reservation(reservation_model_instance)
# reservation is a dataclass
response = reservation_service.get_availability(reservation)
```

Translating a model instance to a dataclass uses an adapter

There is overhead here but it keeps the integration service free of a detail dependency



To adapt the right abstractions and prevent dependency on the ORM, we'd need to define abstract interfaces that give us PSL objects

Creating interface adapters for the ORM is high overhead in terms of production effort, resulting complexity and performance

And it breaks the transactional link

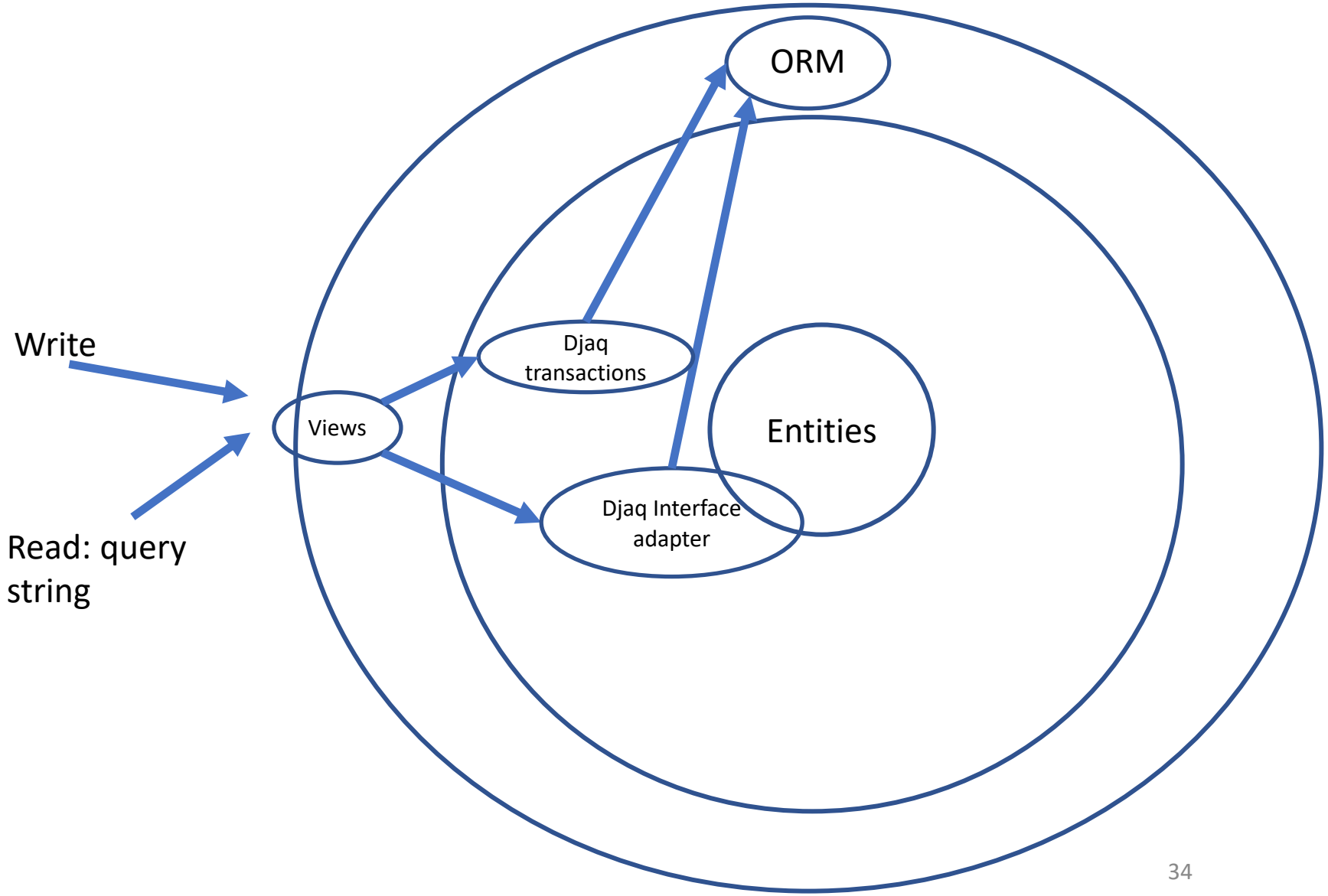
https://github.com/paul-wolf/djaq

The Djaq project (by this author) is an *example* of splitting the ORM along data schema and entity/query lines

It uses the schema definitions produced by the ORM and treats it purely as a data schema definition and data migration layer

It produces plain old Python objects, context-free

Djaq is an interface adapter that returns context free Python objects



REST Frameworks for Django

- Django REST Frameworks generally are heavily integrated with the QuerySet API
- All the different layers are mixed together, details, abstractions, domain entities
- REST frameworks tend to bake the ORM into the rest of the architecture

Recommendations

To profit from the application of Clean Architecture principles, we need to think in terms of how software in general should be structured, not just a Python project or, even more specifically, a Django project

CA requires abstraction overhead; don’t incur this unless the circumstances warrant it

Minimise Dependence on Details

- Any code not:

    - From your business logic code
    - From the Python Standard Library

“context-dependent” variables are objects that are bound to external details, like connection state, frameworks, io mechanics, request/response, databases, etc.

Recommendations (Do Not)

- Avoid passing context-dependent variables as far as possible
- Avoid cross detail dependencies except for ORM/Django View
- Don’t rely on integrations; these are details
- Don’t rely on non-Python Standard Library frameworks more than necessary. Try to write most stable code using only the PSL. Avoid dependencies on frameworks where possible and reasonable

Django Framework

Django is for doing Web view things, serving data models that look from the outside like the domain you've modeled

If you are in a part of your code that is not doing that one thing, don't use the Django framework

- Don't pass the `request` object as a function parameter any more than necessary

- Don't pass integration objects; model instances are integration objects

Recommendations (Do)

Use the Django framework in the way it was intended

Do call the QuerySet API in Django view functions; this is the intended use case for Django, unless you must call other detail level services

Remember this is the edge of the architecture where you are dealing with details. Models are details (and domain entities). Use them directly. Don’t try to hide them behind an abstraction unless that serves to prevent other dependencies

Use Dependency Injection liberally to reduce cross-detail dependencies

The REPL test

Can we load code into a REPL* easily? Loading into these tools should be easy:

- IPython
- Jupyter notebook
- Pdb

Is it easy to create a Django management command for any modules besides views?

* Read-eval-print loop

Unit tests

Unit testing is massively easier when you have an inner layer of business entities that are context-free

No amount of mocking will be as good.

Understand the Trajectory of Change

- Accretion
  - Additive complexity

- Functional Diversity
  - New kinds of complexity

Applications that become more complex because of Functional Diversity will require greater engineering investment, probably before you are aware of it

Finally

Django is for making the initial build of an application easier and less costly

Clean Architecture is for making change easier and less costly after the initial application is built

You can combine these two things. Applying the dependency rule can massively improve software design in a Django project.

References

- https://www.goodreads.com/book/show/18043011-clean-architecture
- https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://alistair.cockburn.us/hexagonal-architecture/
- https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- https://github.com/paul-wolf/djaq
- https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

Thank you for listening!

Many thanks to those who provided reviews of this presentation to help me improve it:

Daria Knyazeva, Jeff Whitehead, Nicola Pero, Andrey Pavelchuck, Andrew Hayton, Thibaud Colas