Clean architecture
aka Hexagonal architecture
aka Ports and adapters
aka Onion architecture

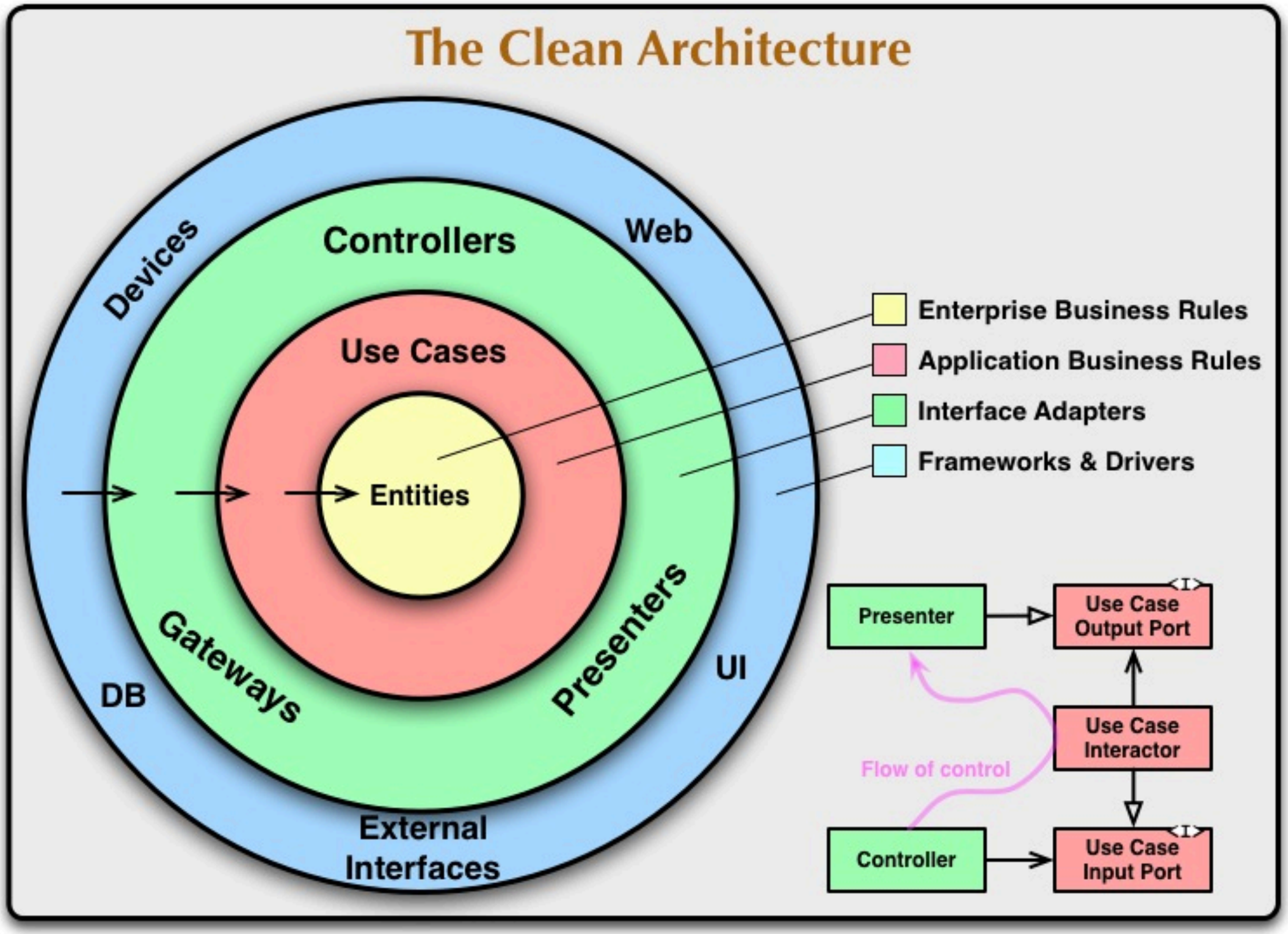
in Django

What's the problem?

- Where do we put business logic?
    - **Views**: hard to read, hard to reuse
    - **Models**: creates complicated model dependencies
    - **Forms**: couples logic to UI
    - **Any of the above**: hard to unit test, hard to share with other services

What's the solution?

- Isolate business logic into its own *inner* layer
- Organize this layer around use cases
- Write adapters for any external dependencies (eg models)
- Have caller (eg view) inject adapters into use case layer



Source: “The Clean Architecture” blog post by Bob Martin
https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

What is clean architecture?

1. **Independent of Frameworks.** The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
2. **Testable.** The business rules can be tested without the UI, Database, Web Server, or any other external element.
3. **Independent of UI.** The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
4. **Independent of Database.** You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
5. **Independent of any external agency.** In fact your business rules simply don’t know anything at all about the outside world.

Source: “The Clean Architecture” blog post by Bob Martin
https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

A simple use case

notes/use_cases.py:

```python
from .entities import Note

class UseCases():
    def __init__(self, storage):
        self.storage = storage

    def create_note(self, title, body):
        note = Note(title, body)
        return self.storage.save_note(note)
```

Use cases class exists for the sole purpose of injecting adapters

Use cases use entities, and don't know about Django models

notes/entities.py:

```python
class Note():
    def __init__(self, title, body, id=None):
        self.id = id
        self.title = title
        self.body = body
```

Entities are just simple classes

adapters/django_storage.py:

```python
from notes.models import Note

class DjangoStorage():
    def save_note(self, note):
        django_note = Note.from_entity(note)
        django_note.save()
        return django_note.to_entity()
```

The storage layer takes entities & returns entities, hides storage details

…and the view

```
from django.http import JsonResponse

from adapters.django_storage import DjangoStorage
from .use_cases import UseCases

storage = DjangoStorage()
use_cases = UseCases(storage)

def create_board(request):
    req_data = json.loads(request.body)
    board = use_cases.create_board(
        title=req_data['title'],
        body=req_data['body']
    )

    return JsonResponse(
        {'board': board.to_dict()},
        status=200
    )
```

Use case abstractions

```
class NoteActions():
    def __init__(self, storage, logging):
        self.use_cases = NoteUseCases(storage)
        self.logging = logging

    @log('board.create')
    def create_board(self, *args, **kwargs):
        return self.use_cases.create_board(*args, **kwargs)

    @log('board.delete')
    @permission('delete')
    def delete_board(self, *args, **kwargs):
        return self.use_cases.delete_board(*args, **kwargs)
```

Use case abstractions

```
from rentomatic.shared import response_object as res

class UseCase(object):

    def execute(self, request_object):
        if not request_object:
            return res.ResponseFailure.build_from_invalid_request_object(request_object)
        try:
            return self.process_request(request_object)
        except Exception as exc:
            return res.ResponseFailure.build_system_error(
                "{}: {}".format(exc.__class__.__name__, "{}".format(exc)))

    def process_request(self, request_object):
        raise NotImplementedError(
            "process_request() not implemented by UseCase class")


class StorageRoomListUseCase(uc.UseCase):

    def __init__(self, repo):
        self.repo = repo

    def process_request(self, request_object):
        domain_storageroom = self.repo.list(filters=request_object.filters)
        return res.ResponseSuccess(domain_storageroom)
```

One more thing: attrs library

```
>>> import attr
>>> @attr.s
... class Coordinates():
...     x = attr.ib()
...     y = attr.ib()
>>> c1 = Coordinates(1, 2)
>>> c1
Coordinates(x=1, y=2)
>>> c2 = Coordinates(x=2, y=1)
>>> c2
Coordinates(x=2, y=1)
>>> c1 == c2
False
```

Further reading

- https://github.com/bjudson/topsy

- https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

- http://blog.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example/

- https://www.youtube.com/watch?v=DJtef410XaM