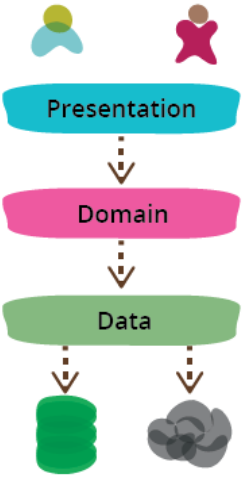
Application Layers



PresentationDomainDataLayering — Fowler
https://www.martinfowler.com/bliki/PresentationDomainDataLayering.html

Why Separate Layers?

- Understandability, each layer has a coherent set of responsibilities, concerns are separated
- Substitutability, e.g. easy to substitute different front ends or data stores
- Testability
- Supports parts of the system changing at different rates
- Team specialisation

Which Layer's Code Is The Most Valuable?

- The domain layer contains the business IP
- Presentation and data access layers are typically built from, and use, standard components and frameworks

The domain layer is special $\Rightarrow$ isolate and protect this layer

Domain Layer Isolation and Protection

The domain layer should

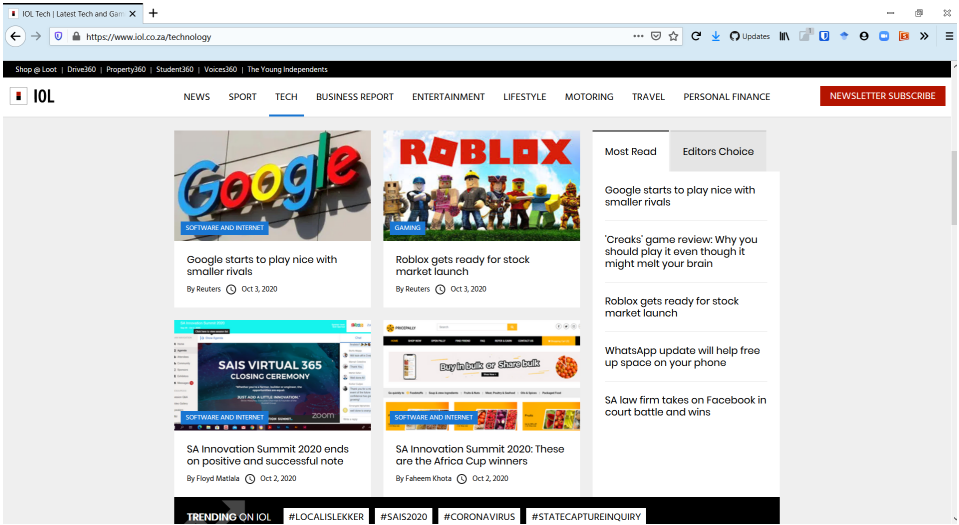
- make use of a *ubiquitous language* (this part of the system should reflect the problem domain in a very literal way, so the mapping is obvious)
- contain a clean, expressive domain model unpolluted by infrastructure concerns
- have limited exposure to external components beyond the team's control — these represent risk if the components change (eg. Web API's)

Domain Layer Isolation in Practice

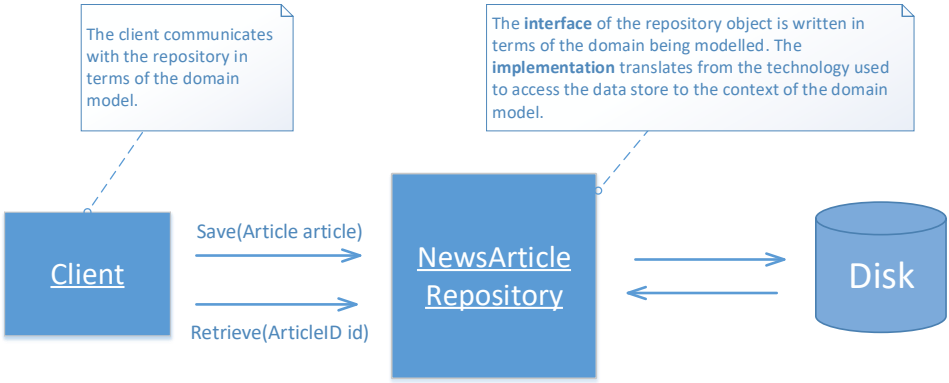
> *"You don't want the domain model to [directly] depend on anything that talks to any kind of external system"*
>
> — Mathias Verraes

- Isolation from the presentation layer happens by default if the dependencies are right
- Isolation from the data store, and other services, requires you to write your own classes or interfaces which shield your domain from these external components (eg. the data access layer)

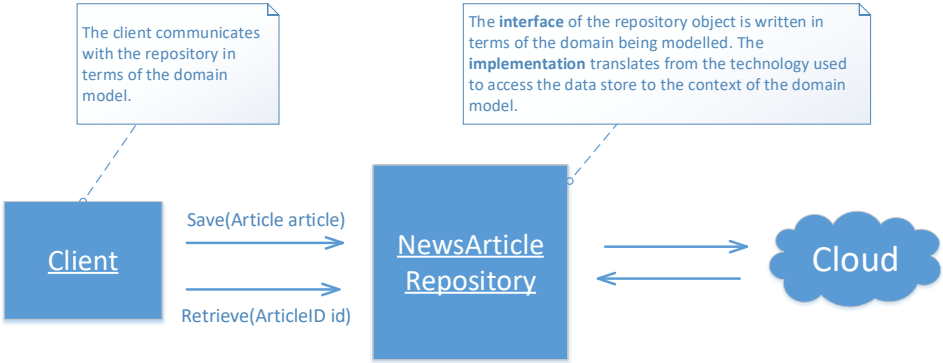
News Site Domain



Isolation From The Data Store



Isolation From the Data Store cont.



Using Third-Party Libraries Within The Domain Layer

- Avoid re-inventing the wheel, use third-party libraries and classes which are well-written, stable, tested, and maintained
- Consequences
  - Tightly coupling the domain logic to code you do not own/control
  - Be wary of frameworks and libraries that force you to compromise your design
- Use library classes *as is* if they represent genuinely useful abstractions
  - `boost::scoped_ptr` was the forerunner of `unique_ptr`
- If necessary, *wrap* library classes (using composition) to make them meaningful for the domain, and to expose only the methods that the domain requires
  - In a "shipping and delivery" domain create a `DeliveryDate` class which internally uses `boost::date_time` in order to exclude delivery dates on weekends