

Domain Driven Design Demonstrated

Alan Christensen @christensena

What is a Domain?

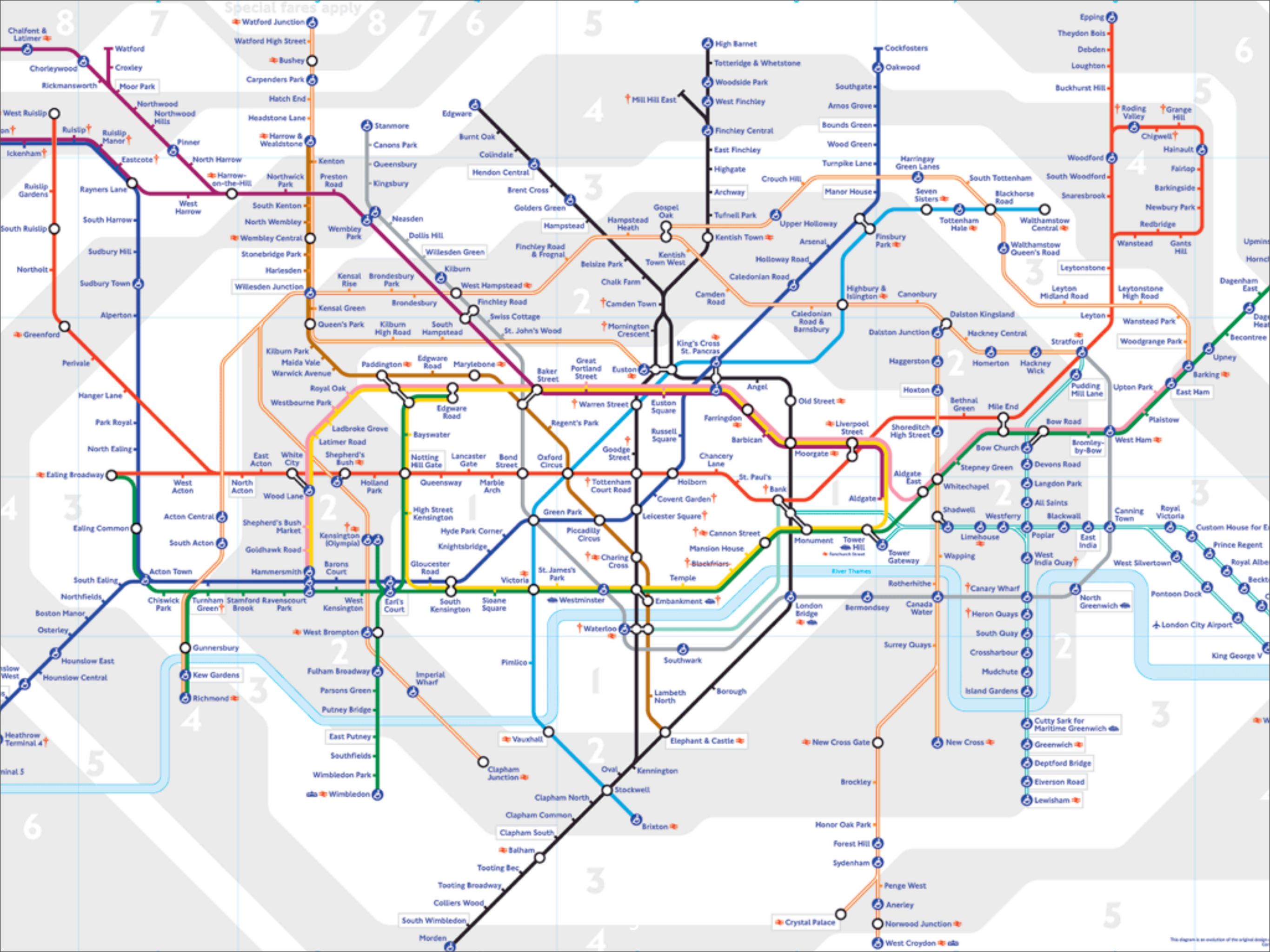
- “A sphere of knowledge, influence or activity”--Eric Evans

What is DDD?

- In order of importance:
 - Design methodology
 - Architectural style
 - Set of software patterns
- “Tackling Complexity in the Heart of Software”--DDD book tagline

Domain Modeling

- Understand your domain
- Model it in the code to suit the purpose and context for which it was intended
- Leave out details and concepts that don't add value
- Keep refining: “refactor to greater insight”



Ubiquitous Language

- Naming is important!
- As is a shared understanding and consistent use of terms
- The code should use the same terms used in documents and discussion
- Both domain experts and developers contribute to the shared language

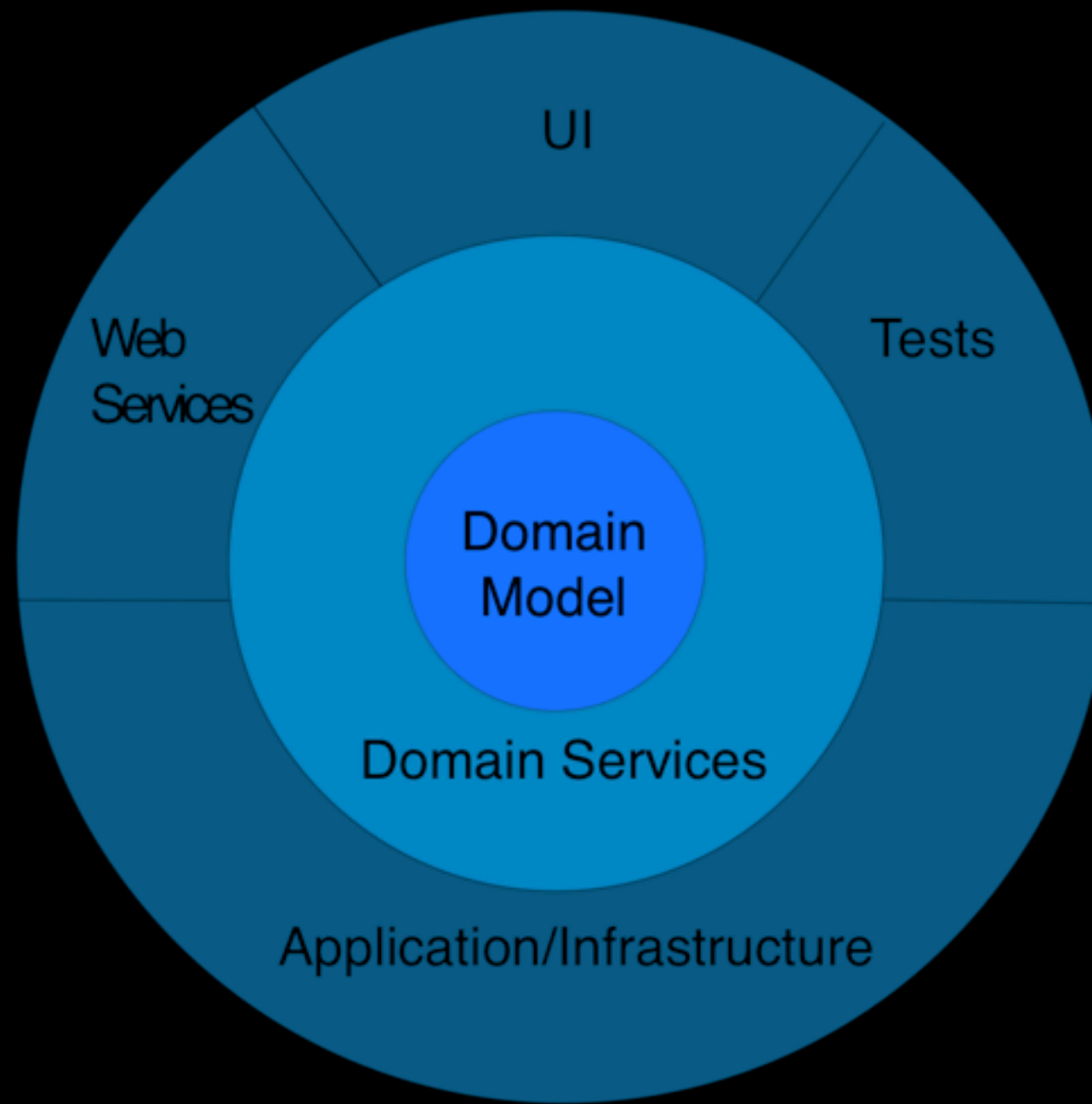




No entry for heavy
goods vehicles.
Residential site only
←
Nid wyf yn y swyddfa
ar hyn o bryd. Anfonwch
unrhyw waith i'w gyfieithu.

the Welsh reads "I am not in the office at the moment. Send any work to be translated."

Onion Architecture



Entities

- Have an identity
- Identity may be determined by a natural or assigned key (e.g. Id)
- Equals implementation to distinguish identity normally uses key
- Mutable - can be changed

Value objects

- No identity. Can be mixed and matched
- Equals implemented as “all fields/properties match”
- Immutable - replace instead of change

Worked example

- What domain should we use?
 - Inventory?
 - Payroll?
 - Stock broking?



Code demo #1

- Model Driven Design
 - Domain Methodology
 - Entities and Value Objects
 - Invariants

Invariants

- Invariants ensure consistency in the domain model
- They allow us to code with confidence that invalid/unnatural states are not possible
- They enforce domain rules and prevent logical fallacies

Invariants

- Examples
 - Private setters
 - Required constructor/factory method parameters
 - Exceptions for invalid operations or invalid arguments to methods

Validation?

- Validation is not really a domain concept. Invariants are the richer idea.
- Validation should be done outside the domain to prevent invariants from ever occurring (exceptions are for exceptions)

Persistence Ignorance

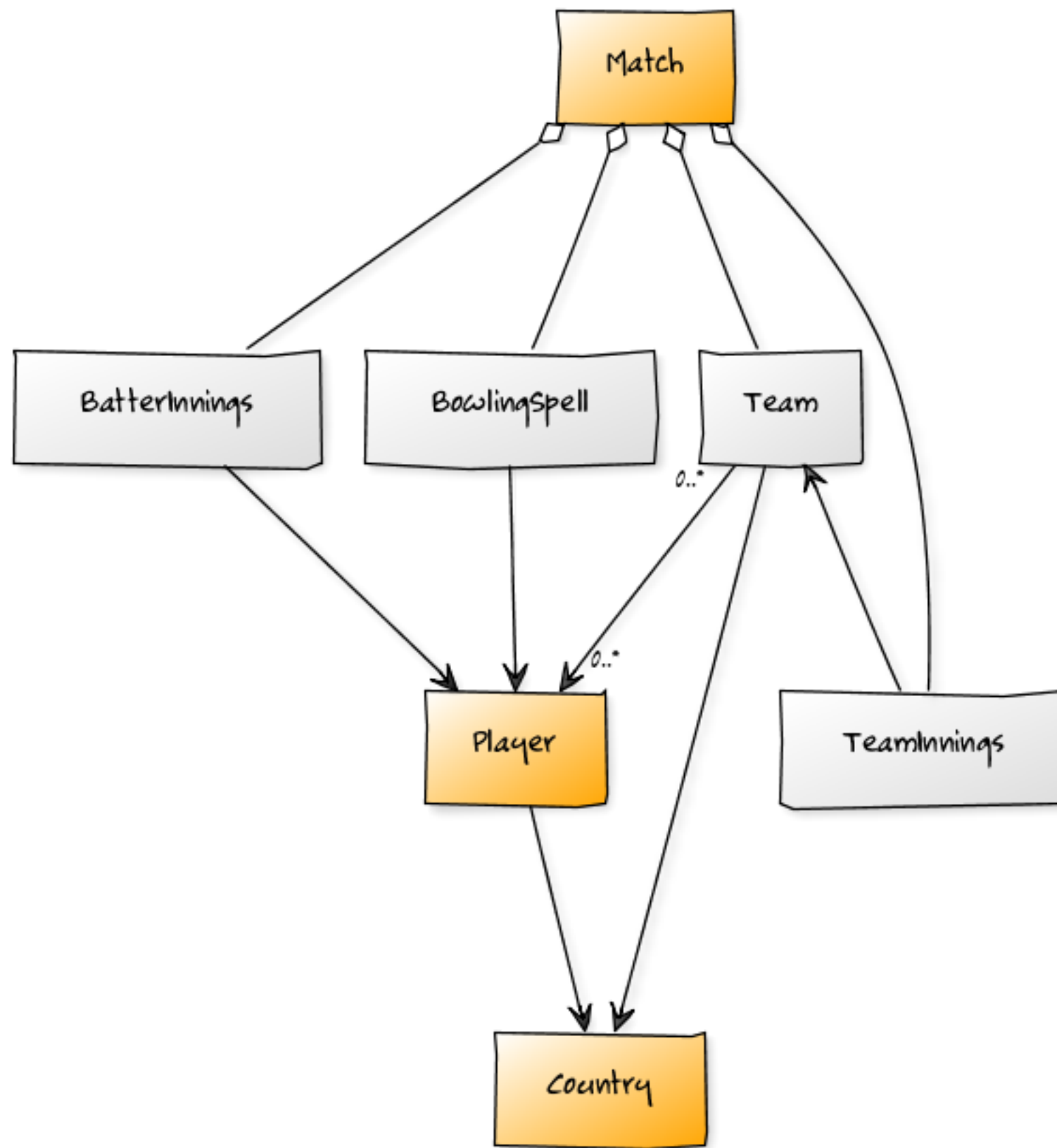
- Persistence is a technical concern. Not part of the domain.
- Fortunately good ORM's support Persistence Ignorance out of the box
- Transactions can be packaged in a "Unit of Work" concept

Repositories

- Semantically just “collections” with enhanced “find” functionality
- In reality they will be the “gateway” to the persistence store
- Repositories are the main mechanism for Persistence Ignorance

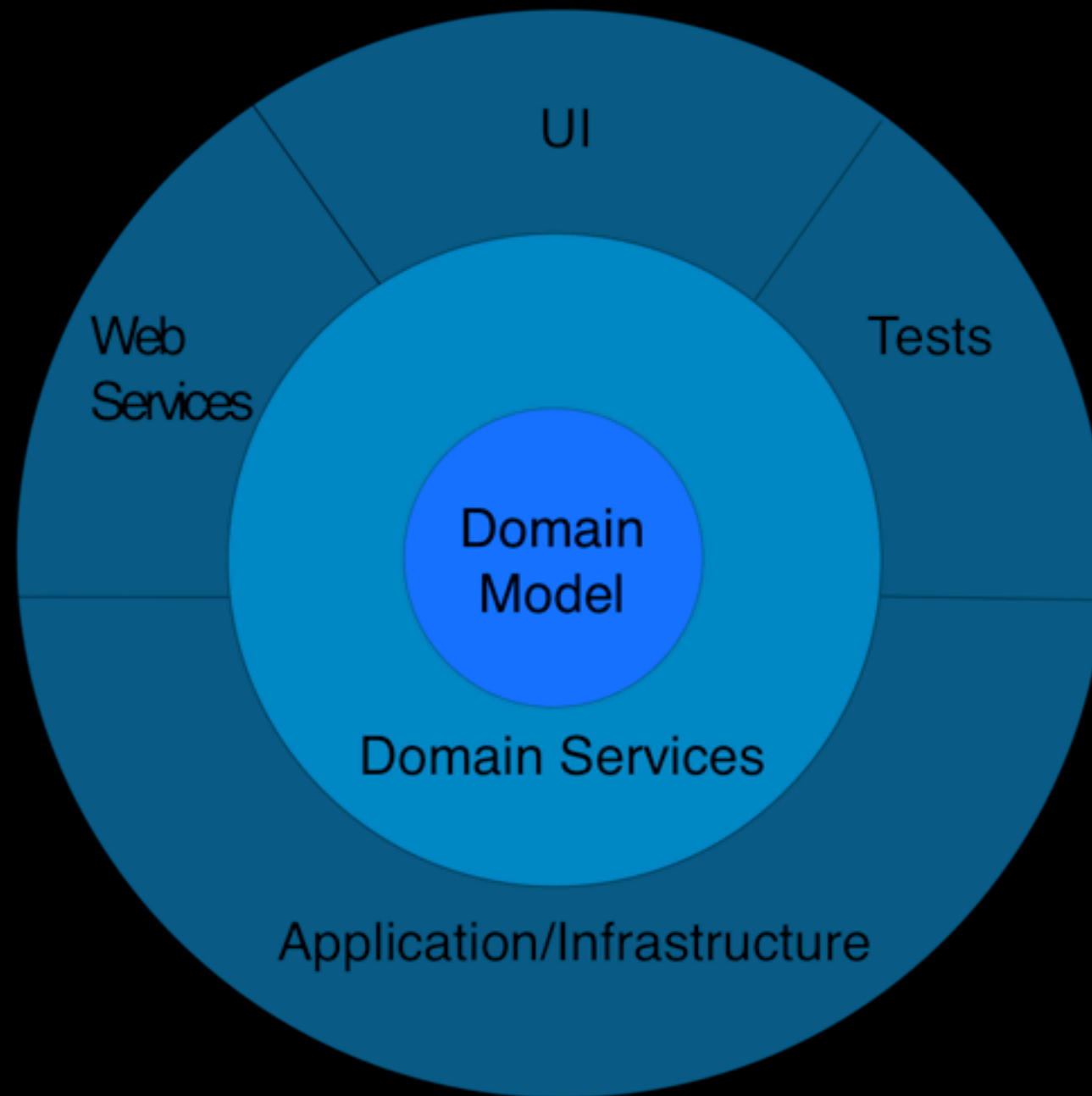
Aggregates

- Some entities only make sense in the context of a parent entity or hierarchy
- Nominate entities as Aggregate Roots
- Value objects and non-aggregate entities are only accessible by traversing from their aggregate roots
- Inter-aggregate relationships via queries/lookups on repositories



Code demo #2

- Repositories
- Unit of Work
- Fluent NHibernate AutoMapping
- Session per Request



Domain Services

- Not general-purpose
“services” (overloaded term)
- Repository/UoW aware (entities are not)
- Able to coordinate business processes
- Most logic should still be inside entity and value objects (Domain Model)

UI and Data Binding

- Don't try to data bind to anything in your domain model!
- Bind to “view models”, tailored to your view (MVC/MVP/MVVM)
- Read operations: Use tools like AutoMapper to map to view models.
- Write operations: Intention/behaviour oriented. Command processor pattern works well

Transports/Hydration

- ORMs such as NHibernate create proxies, bypass invariants for re-hydration
- Invariants mean .NET serialisation needs to be via DTO's.
- DTO's are best not as domain model clones! As with UI viewmodels, tailor for purpose.

NoSQL?

- DDD style aggregates can be a natural fit for document oriented databases
- No need for mapping to relations!
- Use of event sourcing can feed same information into reporting database(s)
- If using relational as dual use e.g. reporting, consider the change resistance

Fashions & Developments

- CQRS
- Micro-ORMs
- Repository pattern?
- ORM hate

Summary

- Modeling Driven
- This means designing and writing code that
 - expresses the domain logic
 - follows/enforces the domain rules
 - uses the domain language
 - databases, UI, infrastructure, etc are outside the domain

Should I use DDD?

- DDD over engineered for simple CRUD or mostly data oriented applications
- Best suited to complex, behaviourally oriented applications
- However, many of the ideas and patterns are useful in all sorts of projects

Other DDD concepts

- Factories
- Bounded Contexts
- Anti-corruption Layers
- Domain Events

Umpire	Scorecard
Over Player Delivery No-Ball Wide Out Crease Boundary LBW Bat-Pad Stumping Light Weather	Over Player Run Team Innings Dismissal Bowling Spell Country Scorecard Match Fall-of-Wickets

References

- DDD Quickly book on InfoQ bit.ly/dddquickly
- Think DDD (Jak Charlton) bit.ly/thinkdddbook
- Onion Architecture bit.ly/onionarch
- Martin Fowler on Aggregates and NoSQL bit.ly/aggregates-nosql
- My example code (and these slides): github.com/christensena/DDDIntro
- New book “Implementing Domain Driven Design” by Vaughn Vernon out soon bit.ly/iddbook

Alan Christensen
@christensena

Microsoft®