Introduction

• Why we chose DDD
• Inputs to our DDD process
• What we did
  – Design patterns we applied
• Where we deviated from DDD
  – For better and for worse
What is DDD?

• The construction of software around a domain model
  – The model and the heart of the design shape each other
  – The model is the backbone of a language used by all team members
  – The model is distilled knowledge
Why DDD?

• Business has a core domain that is represented in multiple technologies. Technologies change, business domain is relatively fixed.

• Instead of learning technologies, developers interface via “simple” domain objects when interacting with business systems
Why DDD?

- Domain model forms a key part of a bigger architecture. This is a nice abstraction to work against, and is understood by developers and domain experts.

- “Anti-corrupt” legacy systems with differing domain models.
Inputs

- *Domain Driven Design* – Eric Evans

- CSLA framework and *Expert C# Business Objects* – Rockford Lhotka
Inputs

• *Patterns of Enterprise Application Architecture* – Martin Fowler
• TMF’s NGOSS specification
• Knowledge of domain and current systems
Design Patterns Applied

• Value Types
  – Value classes are immutable, simple C# objects (*String* is an example of an intrinsic Value Type in .Net)
  – Apart from serialisation, they have no “behaviour” and remain simple.
  – Have helped to simplify problems related to keys used by foreign entities.
Design Patterns Applied

• Data mapper
  – Keeps domain class clean – it does not need to inherit from some special base class. Can be a “normal C# class”.
  – Mappers can inherit implementation characteristics from specific technology base mappers.
  – One domain class may have many mappers: one via custom web service, one via SQL server, one via Oracle and so on.
Design Patterns Applied

• Repositories
  – “One place” to get domain object instances. Loading and the construction of object happen through these classes.
  – Repository decides which mapper to use to load objects from some persistent technology.
  – Some questions remains on how many repositories to create. We used less repositories than described in the book.
Design Patterns Applied

• Identity map
  – Loaded object will not reload.
  – Makes client side simpler (the developer does not have to maintain references to increase performance)
  – Enables a “pre-load” possibility when objects in an object structure can be identified beforehand (avoids the load-by-walk bottleneck of OO models)
  – Map can be cleared when reload is required. Not really a big issue for web applications
Design Patterns Applied

• Unit of Work used to commit changes
  – Provides a mechanism that goes beyond data layer; could call web services, insert MSMQ messages etc.
  – Commit only starts after all changes have been validated
  – Added an item generation facility to known pattern: can be used for “side-effect” changes (after saving entity A, send an email, sending an email is next generation)
Design Patterns Applied

- CSLA Validation
  - The DDD book had no clear direction on an approach for validation
  - CSLA was regarded as stable by some developers and employed for this task.
  - Using CSLA validation is implemented in a consistent manner
  - Clients can access validation errors from the domain object. They are not coded on the client side (we need only English messages 😊)
Design Patterns Applied

• Anti-Corruption Layer
  – Keep the domain model separate from the models of legacy / external systems
  – External functionality is exposed as a service in terms of the domain model
  – The service uses adapters and a façade internally
Where we deviated

• Repositories vs. Factories
  – The DDD book proposes the separation of object creation into a separate factory class. This seems to be overkill when you have simple creational patterns

• Repository granularity
  – The DDD book proposes a separate Repository for each Aggregate. This leads to many repository classes when you have lots of small aggregates
Where we deviated

• The same set of services is required from multiple business systems
• Integration to these business systems was defined as a Service interface.
• Each system has an implementation which is chosen by the Repository
  • vs. a unified anti-corruption layer.
Where we have failed

• Refactoring towards deeper insight
  – Failed to identify domain experts
  – Failed to separate contexts
• Ubiquitous Language
  – Failed to achieve a common language with the domain experts we have identified
Questions and Answers

???