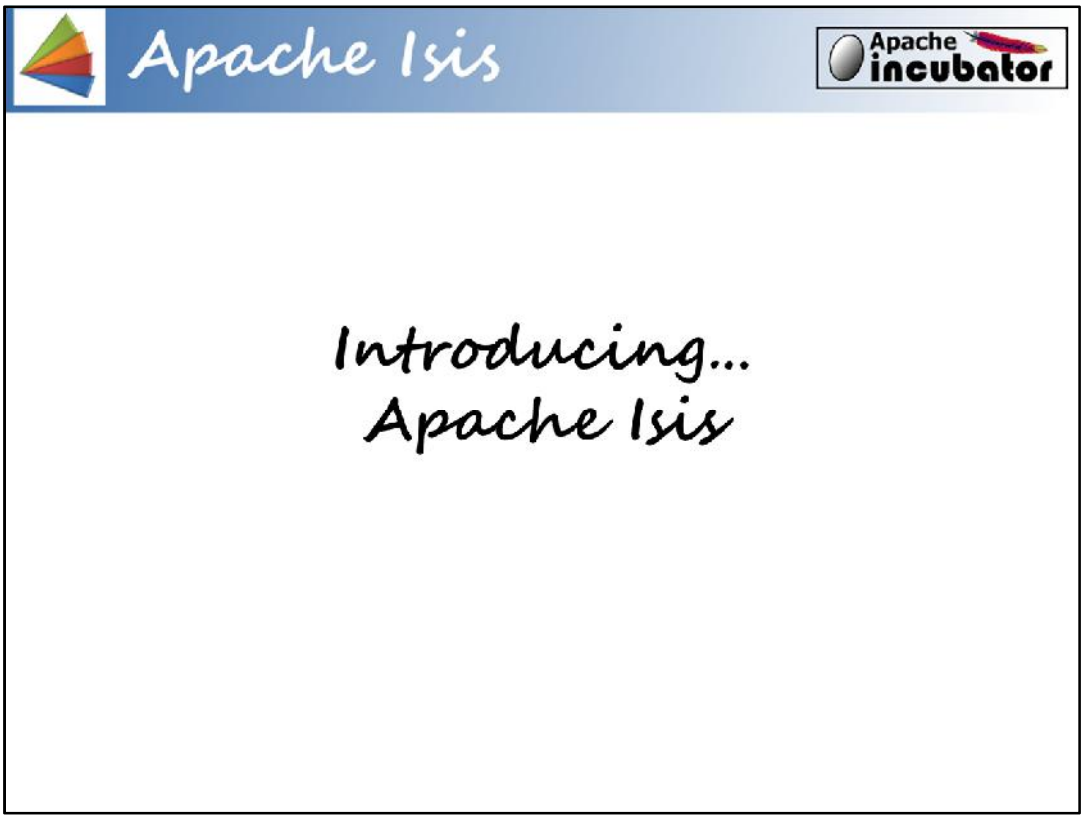
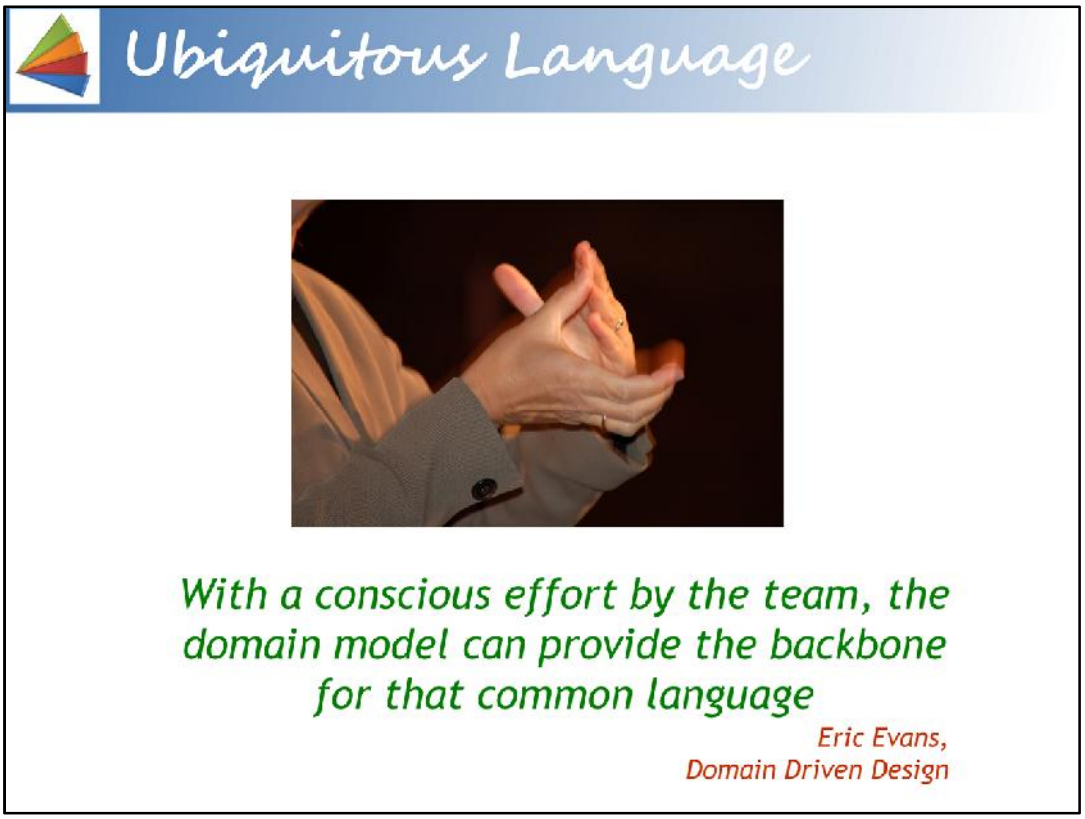


This presentation has been put together for anyone wanting to do an introductory talk of Apache Isis to their work colleagues or fellow geeks at work, a user group or at a conference.

It consists of 9 slides outlining some of the main themes of Apache Isis, domain-driven design and the naked objects pattern. There are plenty of notes for each slide, so you shouldn't run out of things to say. In addition, one of the slides is a placeholder for a demo, so you have the opportunity to show Isis "in the flesh".

After the final slide ("Resources"), there are 3 additional slides relating to the Irish DSP project. This was the "flagship" project that was the first to put naked objects to real-life use, and is a substantive and highly successful "existence proof" of the validity of the naked objects pattern. You're free to interweave these additional slides into the main material if you wish, or just read them for a little additional context.

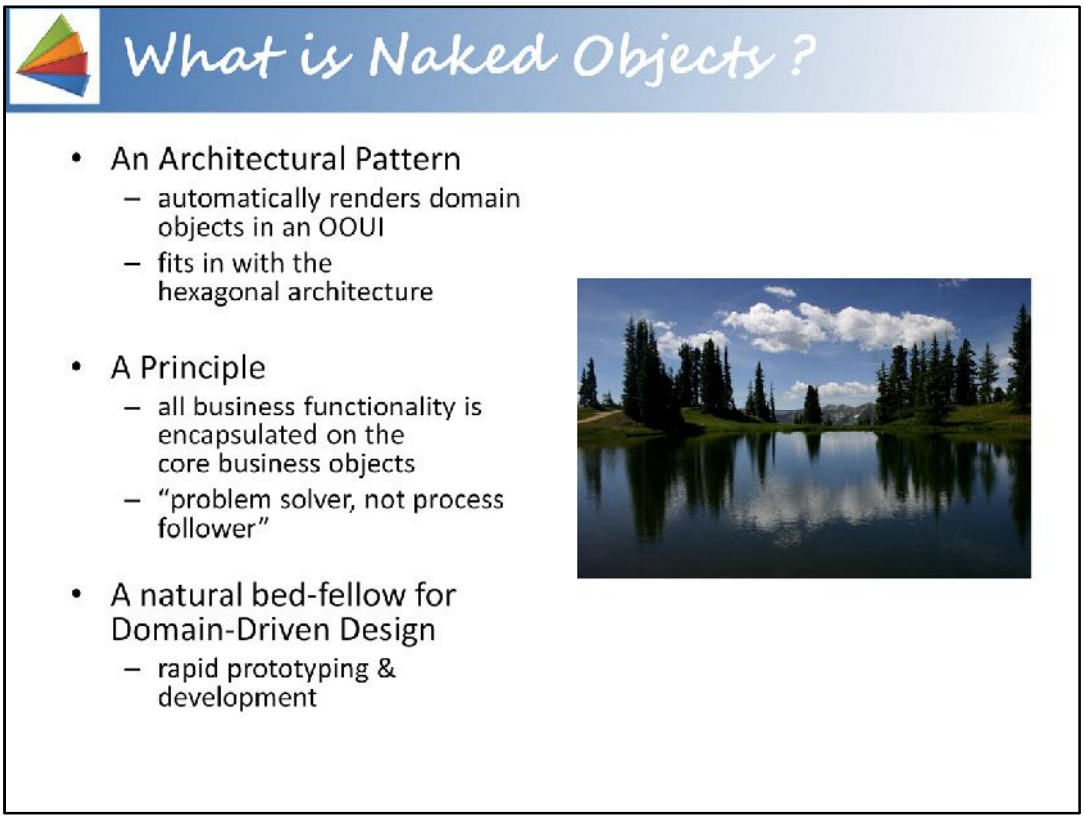


Apache Isis is a framework to support domain-driven design. Eric Evans, the author of the DDD book, introduces the idea of "ubiquitous language" as one of the two core patterns of domain-driven design. The idea of a ubiquitous language is for the team (and by this we mean both the business-oriented domain experts as well as the techies) to communicate ideas only in terms of the core domain objects. They should be talking about customers, orders, products; they shouldn't be talking about screens, persistence, remoting or security. If the team struggling to communicate, then it is probably because some domain concept has not been identified or defined.

An example: on a big government project (DSP) building a new social welfare system, one of the analysis workshops was discussing the process of sending out pension books when they expire. Part of the discussion kept mentioning the cycle of renewals for pension books. Since this phrase kept coming up, it gave rise to the BookRenewalCycle class, which ended up with its own set of responsibilities.

Using a ubiquitous language helps the developers come up to speed with the domain concepts, but it also helps ensure that the domain experts fully understand and define their own domain terms. Non-automated business processes (ie as executed by humans) are able to cope with a degree of fuzziness, but that isn't the case for business processes that are being automated by computers. And of course, the ubiquitous language becomes the vocabulary by which users stories are articulated.

The other core pattern of DDD is "model-driven design"; if anyone asks we pick up on this topic later on in the presentation.



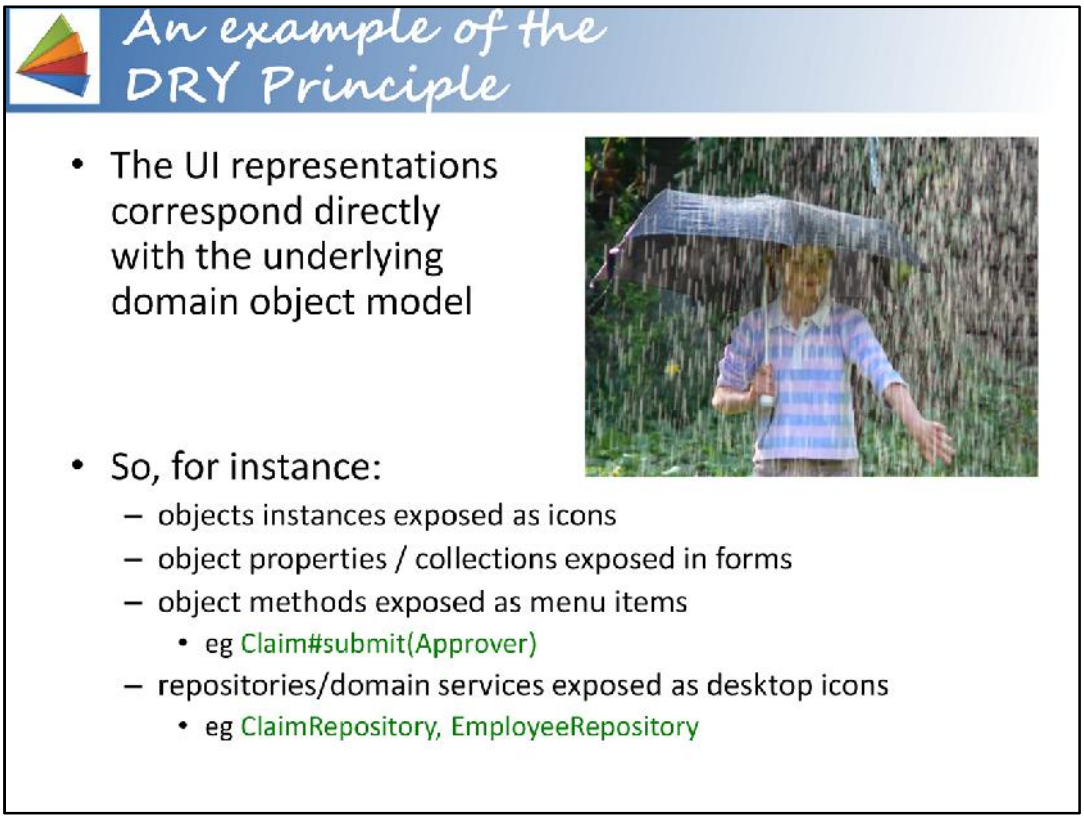
Apache Isis supports domain-driven design by implementing the naked objects pattern.

"Naked Objects" as a term has several definitions. First, it is an architectural pattern for a system whereby the presentation layer is an automatically generated object-oriented UI, constructed at runtime from the domain model. This isn't compile-time code generation scaffolding, it's building a metamodel on-the-fly at runtime. As an pattern, it also fits in well with the hexagonal architecture (discussed shortly).

Perhaps more fundamentally, the principle of naked objects is object-orientation as "your mother taught you", applied to enterprise business applications. So we're not talking about objects as anaemic data holders that are manipulated by service layers; these are proper objects with "know-whats" (encapsulated state) and "know-how-tos" (encapsulated behaviour).

Another theme to naked objects is that the UI should allow the user to perform their job in the way that they want, without forcing them to follow narrowly defined paths. This is especially useful for expert users who have a deep knowledge of the domain (something often true for end-users within an enterprise). The domain objects take responsibility for ensuring they are never placed into an invalid state, and provide behaviours for the user to manipulate them, but say nothing about the order in which they are manipulated. We usually describe this as naked objects systems being written for the "problem solver, not process follower". (Note that the above doesn't preclude the UI being customized to provide additional support for non-expert users).

As we'll learn about in the following slides, it's the rapid application development of naked objects applications that makes it a natural bedfellow for DDD: the team can quickly build up their ubiquitous language with the minimum of distractions.

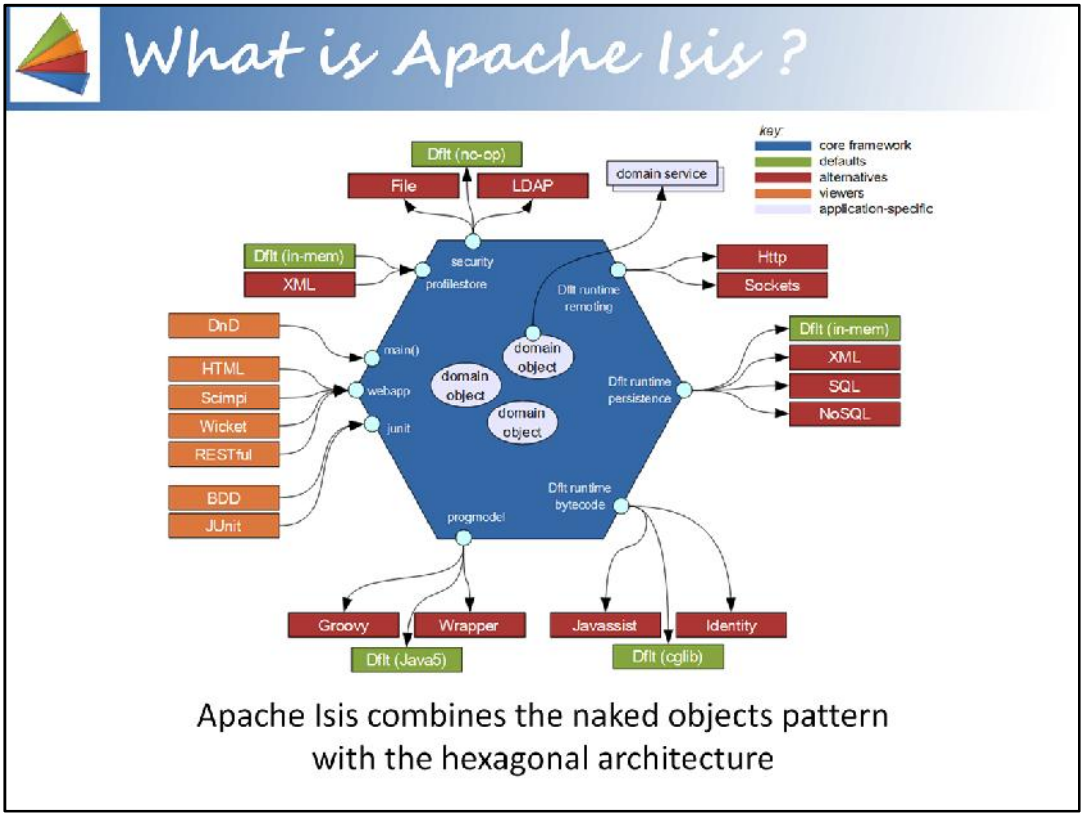


The "don't repeat yourself" (DRY) principle is the idea that every concept, business rule and validation should be expressed in one place and one place only. The term was originally coined by Dave Thomas and Andy Hunt in their book, the Pragmatic Programmer (2001), and has been widely quoted since then as sound advice.

Object/relational mappers such as Hibernate are a good example of the DRY principle; rather than writing lots of boilerplate JDBC to insert, update and delete objects into the database, we instead defining a mapping and let the ORM do the heavy lifting for you.

The naked objects pattern is another example of DRY, but this time applied to the presentation layer rather than the persistence layer. So, object instances are automatically exposed as icons, while the object can be opened up into forms showing the object's state (properties and collections). Furthermore all other public the object's methods (we call them actions) are rendered as menu items or links.

It's worth contrasting this with other tools that can generate CRUD applications. First, the UI is generated at runtime, not compile-time (there's no "generate scaffolding" command to run). Second, exposing object actions means this is more than just simple CRUD style applications. Third, with Apache Isis we always starts with the domain layer. Some other tools start with either a database schema (ie reverse engineering a data model), or start with the presentation layer (where the domain model can end up as a 2nd class citizen as the domain expert gets distracted by UI concerns).



Apache Isis is a full-stack framework, meaning that it provides an infrastructure for all the usual architectural layers (by which I mean: presentation, application service, domain model, persistence) of an enterprise application.

Isis is also modular, and has a number of APIs for different services. The slide above attempts to show these different modules, using the hexagonal architecture pattern (also called “ports and adapters”; see Alistair Cockburn’s blog for his original description). From a build perspective, Isis is built using Maven, so each of the boxes shown constitutes either one or several Maven modules.

The middle hexagon is the core framework which acts as a container for the domain objects. In programming terms, this is represented by the DomainObjectContainer interface, and is injected into every domain object. Using it, the domain object can do such things as raising warnings/errors, or creating/persisting/removing objects.

Around the outside of the hexagon are a number of “ports”. Of these, the only ones used directly by domain objects are the domain services, again automatically injected. The services you have will depend on your app, and are written by you.

All the other ports are APIs used by Isis . “Persistence” and “security” should be self explanatory; the “progmodel “ defines the mechanism by which Isis metamodel is built up, and “bytecode” is used for lazy loading. “Remoting” is used to enable client/server support, over a variety of transports.

On the left-hand side are the viewers. These (of course) implement the naked objects pattern: expose the domain objects automatically with respect to the metamodel.

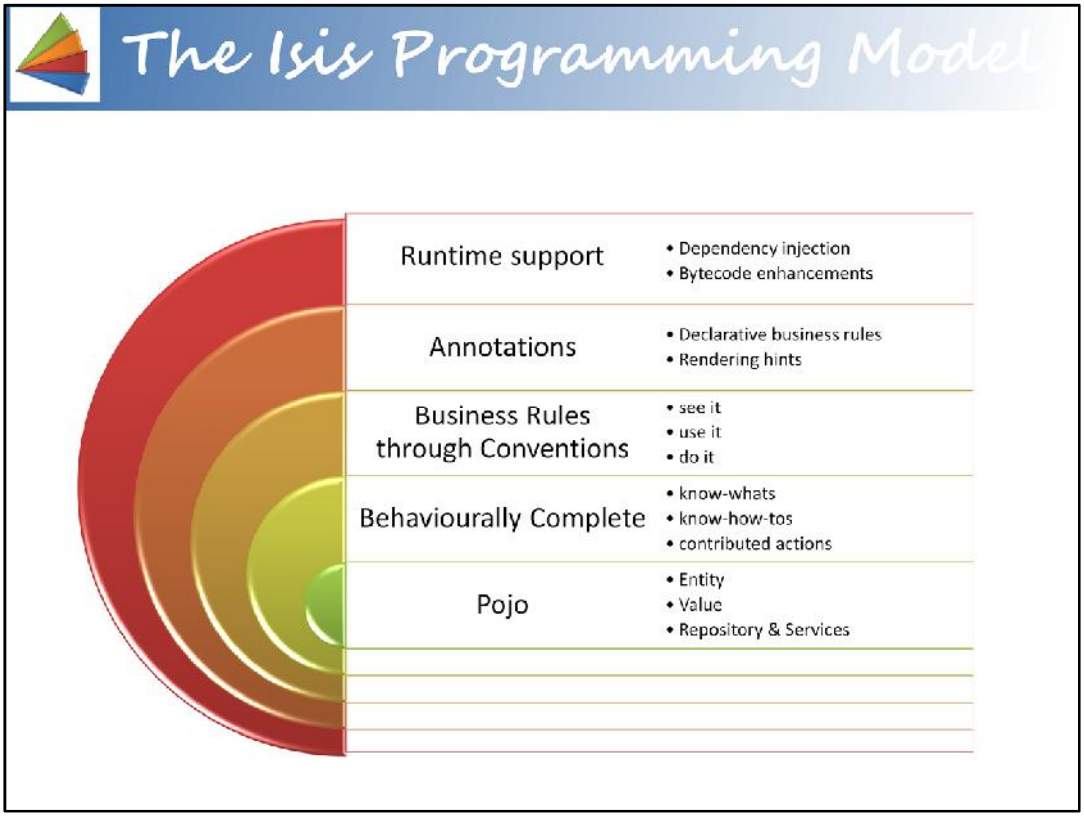
Earlier on we talked about the idea of the team building up a ubiquitous language to communicate, the vocabulary representing the domain of the system. And having a ubiquitous language is all well and good, but if there's no representation of it in the code, then that's a problem.

With Apache Isis, the application fundamentally consists of pojos, one for each domain concept. Although you can customize the presentation layer, you don't need to, so in large part the developers can focus just on coding up the domain objects.

This ties in with the other core pattern for DDD, namely of being "model driven". This is the idea that the domain model described by the ubiquitous language should be reflected within the codebase. In other words, DDD isn't about just a bunch of UML diagrams disconnected with the software.

We tend to recommend that every time you identify a domain concept, add it in some form to the codebase. It may be an entity, an interface or a value, and may start out with very few or even no responsibilities. However, once it exists, it will become part of the team's language and over time will inevitably acquire its own set of responsibilities.

Adopting this approach also removes the artificial barrier that can arise between analysis and design, of maintaining an analysis model and a design model. Instead, analysis should be about being able to zoom in on (or filter out) details from the (one-and-only) domain model. Indeed, going back to the first slide, one thing that Evans says is "with a *conscious* effort the team can build a ubiquitous language". With Isis though, no particularly conscious effort needs to be made to bridge analysis and design, because the majority of coding effort put in by the team is on the classes that would represent that domain object.



Although we've said that Isis applications are "just pojos", there is of course a little more to it than that. Isis defines a set of programming conventions and a number of code annotations, and these together define the Isis programming model.

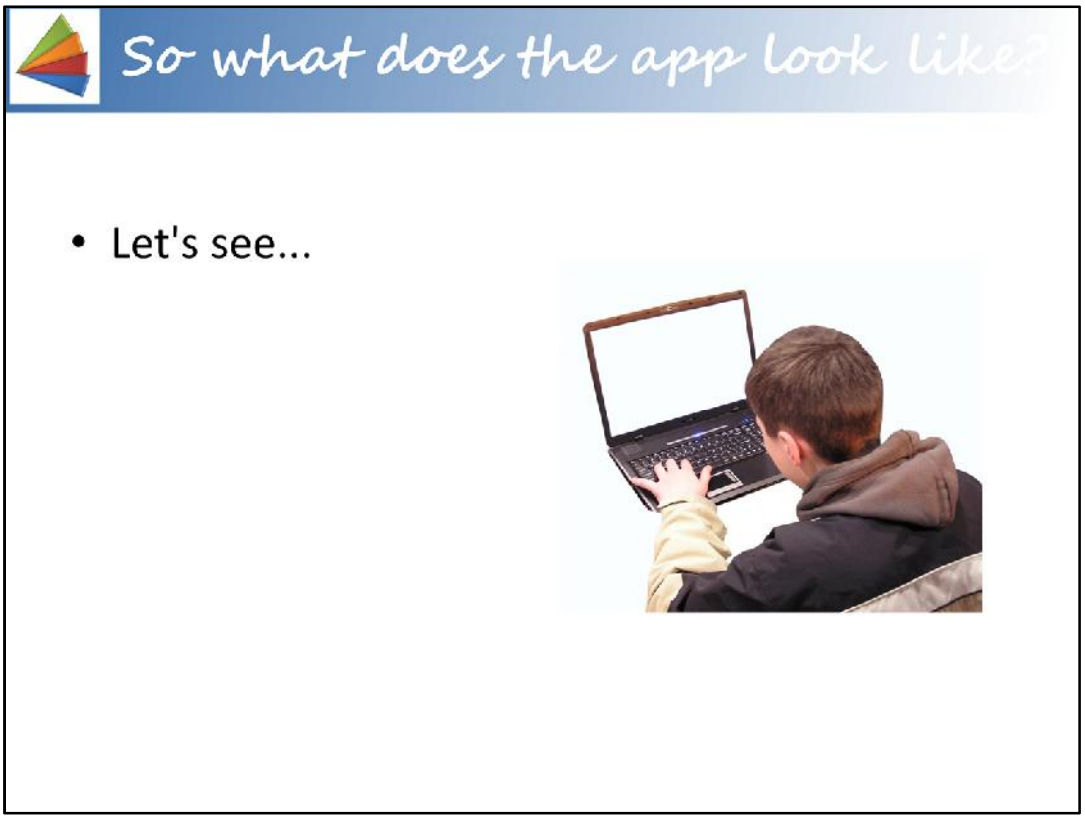
At the base level, you can build an Isis application out of pojos. Those with state and a lifecycle (Customer, Order, Product) will be entities, those that represent state (Money, Date, Duration, BigDecimal etc) are values, and those that are singletons are domain services (by which we include repositories).

Over and above basic getters and setters ("know-whats" for properties/collections), any public methods represent actions ("know-how-tos"). This the behavioural completeness idea discussed earlier. Most business logic is implemented in such actions.

Isis also supports "pre-condition" business rules through conventions. A class member can be hidden; or if visible then it can be disabled (greyed out), or if enabled then the values/arguments can be validated. We summarize this as : "(can you) see it / (can you) use it / (can you) do it".

Such business rules can be specified declaratively through annotations (eg @MaxLength) or imperatively through supporting methods (eg validatePlaceOrder(…) for a placeOrder(…) action). The annotations are in the Isis applib; this is the only compile-time binding that an Isis application has to the framework.

Finally, Isis provides runtime support. It automatically injects all domain services / repositories as well as the Isis DomainObjectContainer, and if needed uses bytecode enhancement for lazy loading and object dirtying.

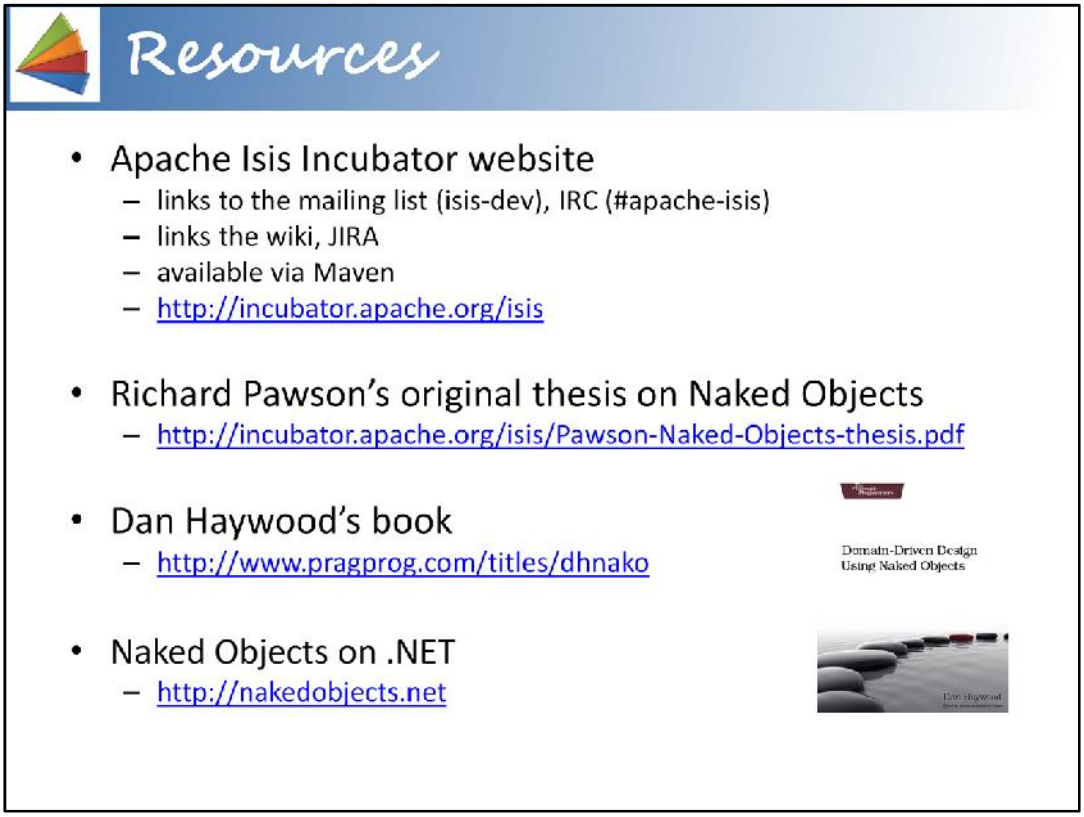


After all that preamble, it's probably worth showing what Isis looks like in action. Start with a pre-canned demo, and then extend it based on suggestions.

The sort of things you could demonstrate include:

- desktop icons with repositories
    - how are bootstrapped from isis.properties
- object properties, collections, properties, actions and their representation
    - eg Customer#getFirstName(), Customer#getOrders(), Customer#placeOrder(Product)
- add a new property or a new action
    - eg Customer#getMiddleInitial(); eg Customer#markAsBlackListed()
- add rendering hints
    - eg @MemberOrder, title(), icon
- add a disable rule (imperative)
    - eg Customer#disablePlaceOrder() - eg if blacklisted
- add a validation rule (declarative)
    - eg @MaxLength(1) for middle initial
- add derived property
    - eg getMostRecentlyOrderedProduct()
- add some utility methods
    - eg choices0PlaceOrder(): List<Product> - product(s) recently ordered
    - eg default0PlaceOrder – eg. the most recently ordered product

It's worth demoing both the DnD viewer and one of the webapp viewers. The HTML viewer works well; somewhat more sophisticated are the Scimpi and Wicket viewers. Other to demonstrate (dependent on audience) are the Restful and BDD viewer.s

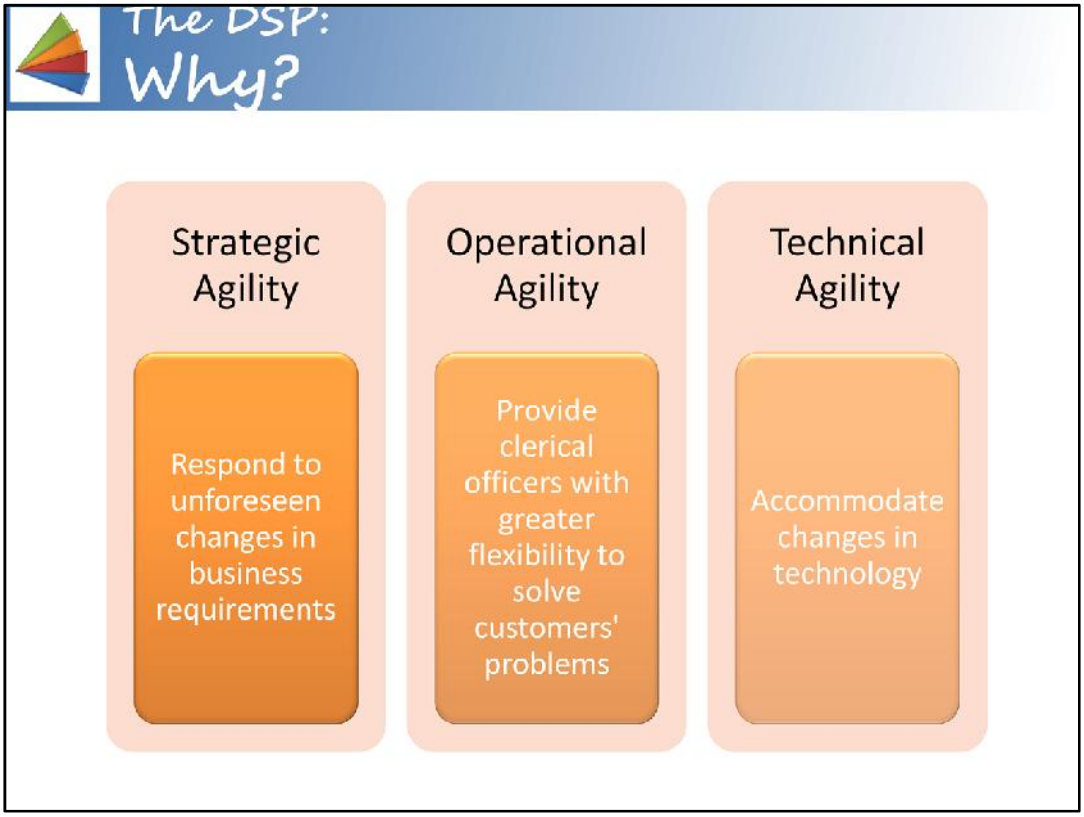


Closing slide, with some resources for those interested.

The main resource is the Apache Isis website, at the URL above. From their the audience can subscribe to the isis-dev mailing list, link to the wiki, go onto our IRC channel, and so forth.

Useful background reading is Richard Pawson's original thesis on Naked Objects. There's also Dan Haywood's book, Domain Driven Design using Naked Objects. This relates to the Naked Objects framework circa 2009, but applies more-or-less unchanged to Apache Isis (the main difference is that the applib package names have changed).

For those who work on the .NET platform, the Naked Objects for .NET platform may be of interest. Naked Objects runs either as a webapp (using the ASP.NET MVC 3 framework), or as a client/server WPF application.

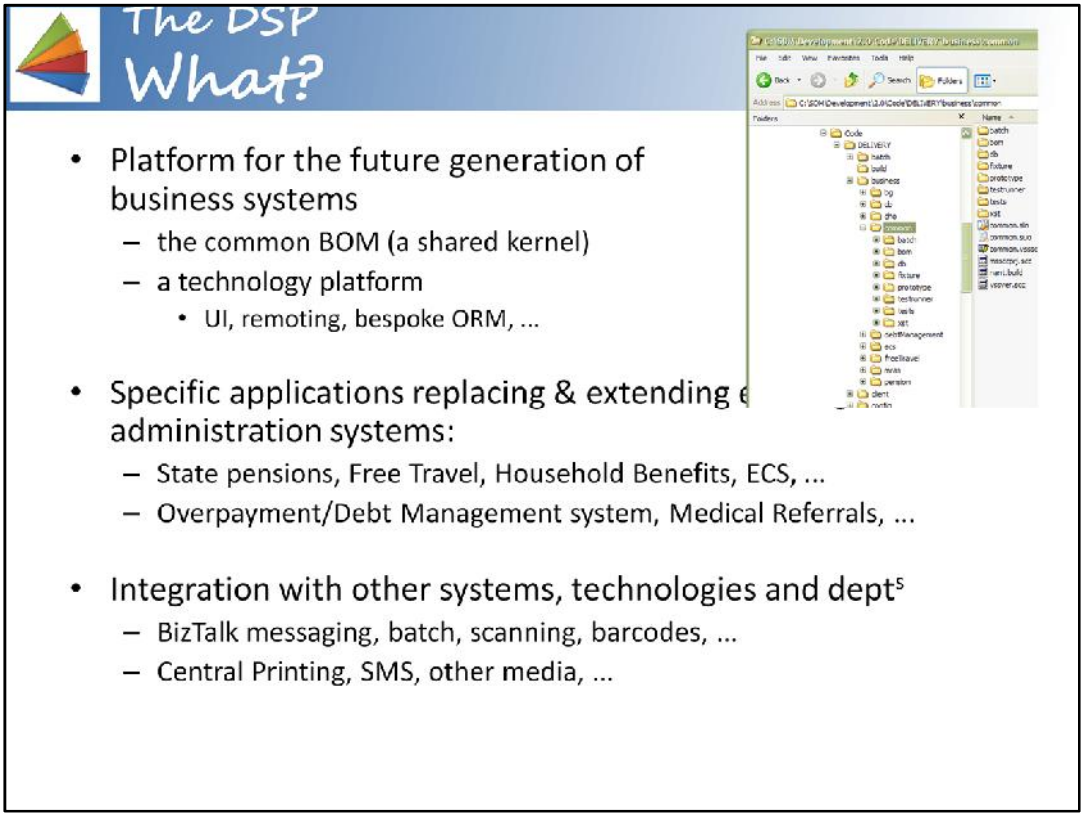


The remaining three slides provide some background on the DSP system. You're free to interweave these additional slides into the main material if you wish, or just read them for a little additional context.

DSP is the Irish Government's Department of Social Protection, responsible for the payment of social welfare payments, such as pensions and child benefit. These are two of the largest benefits, but in all the department administers ~45 benefits. Of these, currently around 12 of these (including pensions and CB) are administered by a naked objects system (running on .NET). As such, the system is a substantive and highly successful "existence proof" of the validity of the naked objects pattern.

This slide talks about why the DSP chose a naked objects system in the first place, couching the answer in terms of letting the department become more agile. The system has indeed demonstrated agility on all three levels; for example:

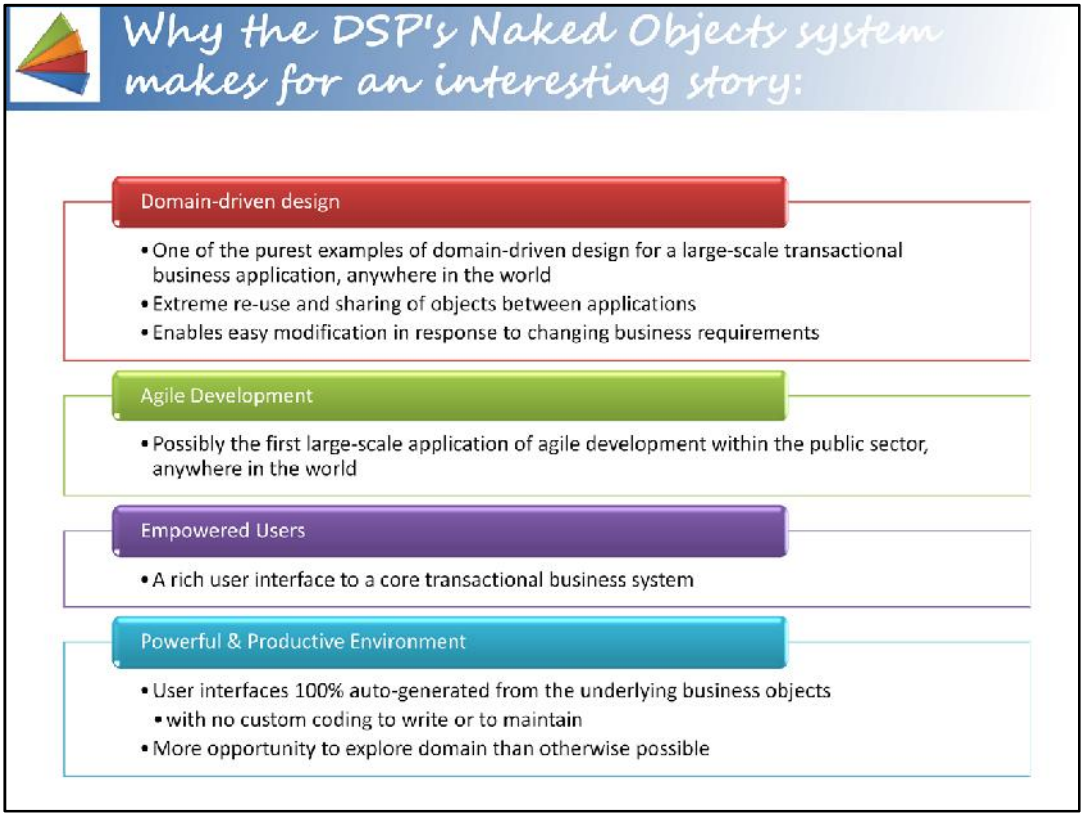
- strategic agility was demonstrated by a new "Early Childcare Supplement" (ECS) benefit that was announced in the Dec 2005 budget, and went live 9 months later, in Aug 2006. Similarly, new requirements in the Nov 2008 budget went live in Feb 2009 and in Apr 2009.
- operationality agility continues to be demonstrated by the OOUI; Richard Pawson's thesis includes details of this in terms of end-iuser interviews
- technical agility relates to the ability to incorporate new technologies, such as domain service implementations for SMS, Barcoding, printing etc; use of BizTalk to integrate with other departments, integration with varied legacy systems for unemployment benefit handling etc.



As mentioned on the previous slide, the DSP administers ~45 schemes, of which ~12 now run on naked objects system. The long-term plan is to support all systems via naked objects (in some cases wrapping legacy systems, in others porting the functionality over).

The system comprises:

- a common BOM (business object model), that acts as a shared kernel for the core concepts such as customer, scheme, payment, officer, auditing, workflow and such like.
- scheme specific “BOMs”, that provide the business rules for the various schemes administered (pensions, child benefit, unemployment benefit, bereavement grant, household benefits, free travel etc etc)
- a technical platform that extends naked objects framework for persistence, remoting, security etc
- domain services that integrate with other systems, technologies and departments



Domain driven design
- the common BOM consists of 120 classes, 60000 lines of code, written by 2 people
- pensions BOM is comparable size; written by 3 people
- extreme reuse:
    - child benefit domain model reimplemented on SDM platform
    - original application >50,000 lines of code
    - domain logic for replacement amounts to just 957 lines of code

Agile development
- monthly releases and planning games
- automated regression testing via FitNesse.NET and Nunit
- continuous integration

Some other facts and figures:
- >1000 end-users, working numerous local offices across the country
- €5bn paid out annually
- >100Gb database