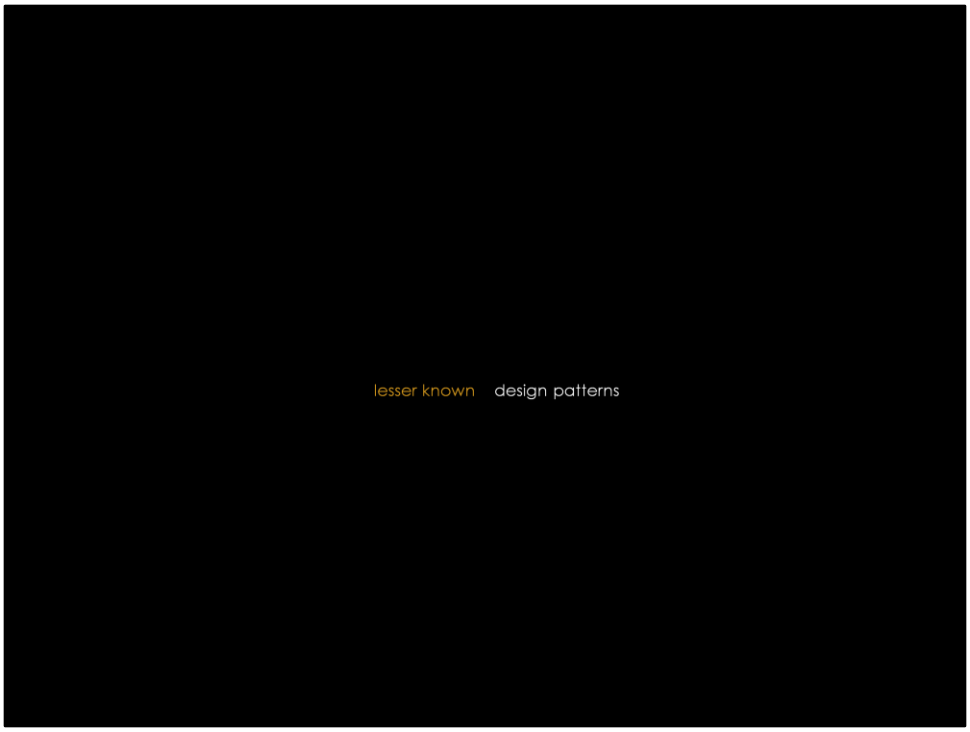
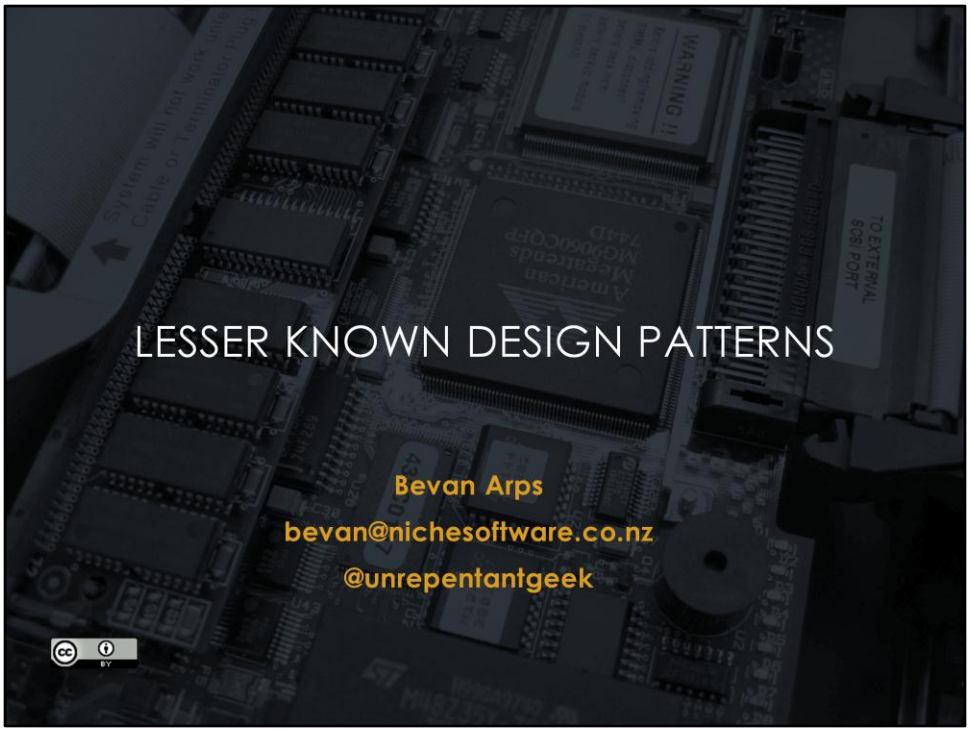

lesser known design patterns

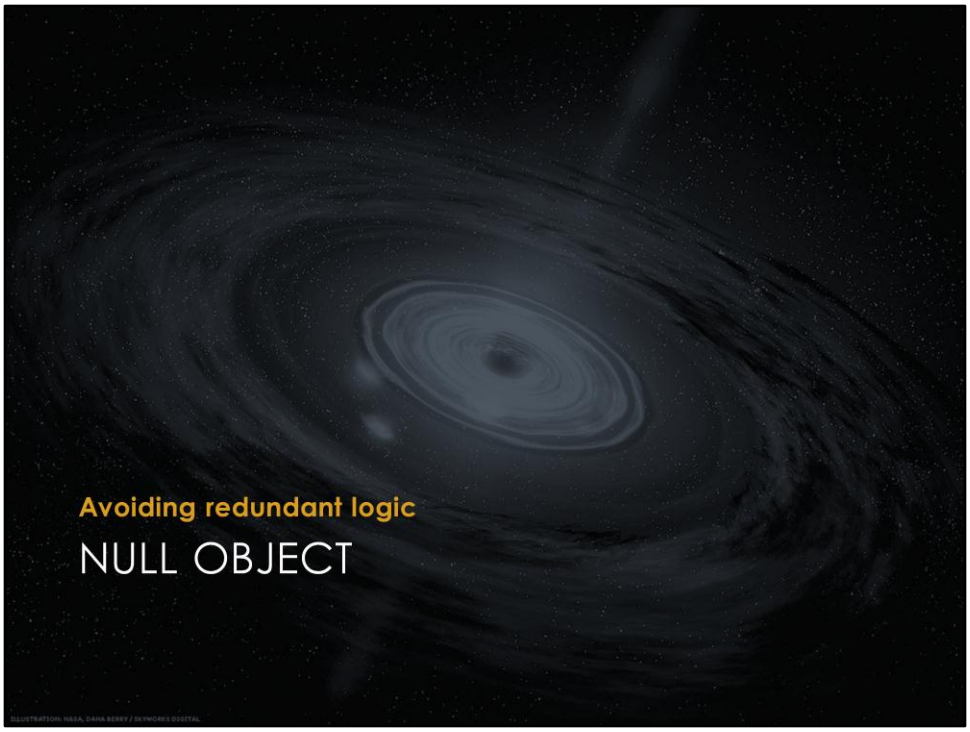

LESSER KNOWN DESIGN PATTERNS
Bevan Arps
bevan@nichesoftware.co.nz
@unrepentantgeek

This is the book that started it all
Published 17 years ago in October 1994

Many of the patterns they catalogued have become commonly known
Including Factory, Adapter

The book does have a C++ bias, focus on patterns that solve problems not encountered by other platforms

There are other patterns – lesser known patterns
Some covered by the GoF book, some documented elsewhere



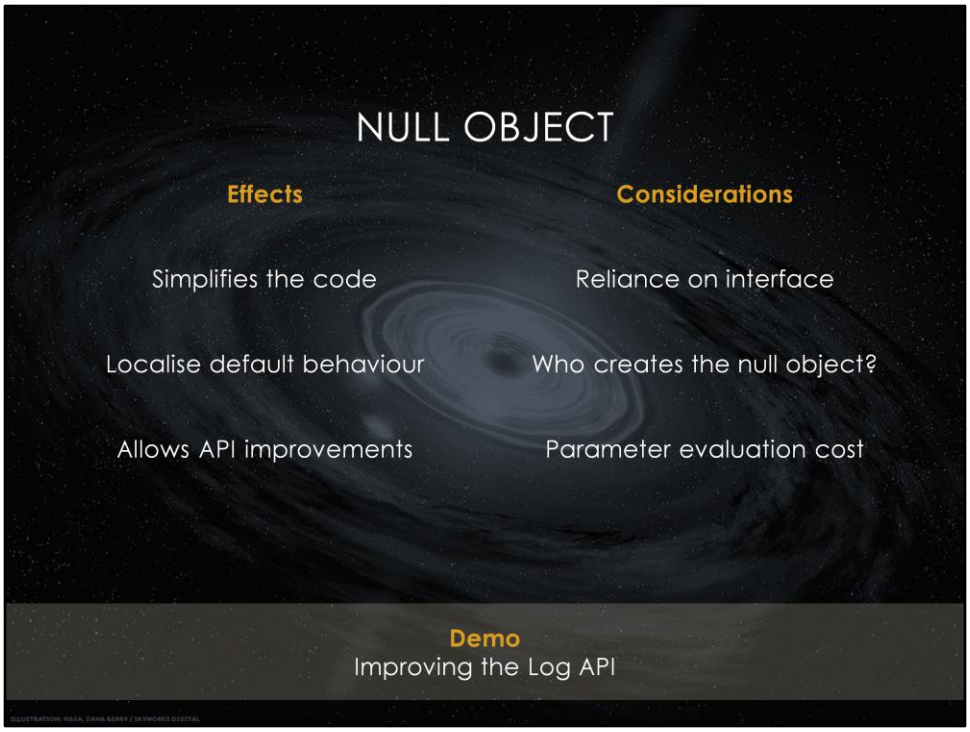
Problem

Repetitive logic around deciding what to do when a resource is unavailable

Solution

Create a stand-in that gets used when nothing else is supplied

Since the resource is always available,
you can dispense with code that checks for it being there



Effects

Removing all the null-checks makes the code simpler, easier to read and maintain

Any default behaviour now has a place to live, further reducing code clutter

With null-checking code out of the way,
improvements to the API of the resource are possible

Demo

Changing ILog to IDisposable Action()

Considerations

Reliance on a common interface to allow multiple implementations

Need to decide who creates the default:
handle this within the object constructor, a property setter
or through configuration of the IoC container

Parameters will always be evaluated, even if never used.
theoretical performance penalty Somewhat balanced by the fewer null checks.
Not a serious problem in reality

Problem

Complex operations are being implemented across multiple objects in your domain
Operation is difficult to understand because the code is distributed
Adding new functions requires modifying multiple classes

Image Credit: http://www.flickr.com/photos/istm/354032848/in/photostream/

Solution
Bring all of the code for each function
together into a single command object
separate from the hierarchy, works by interaction

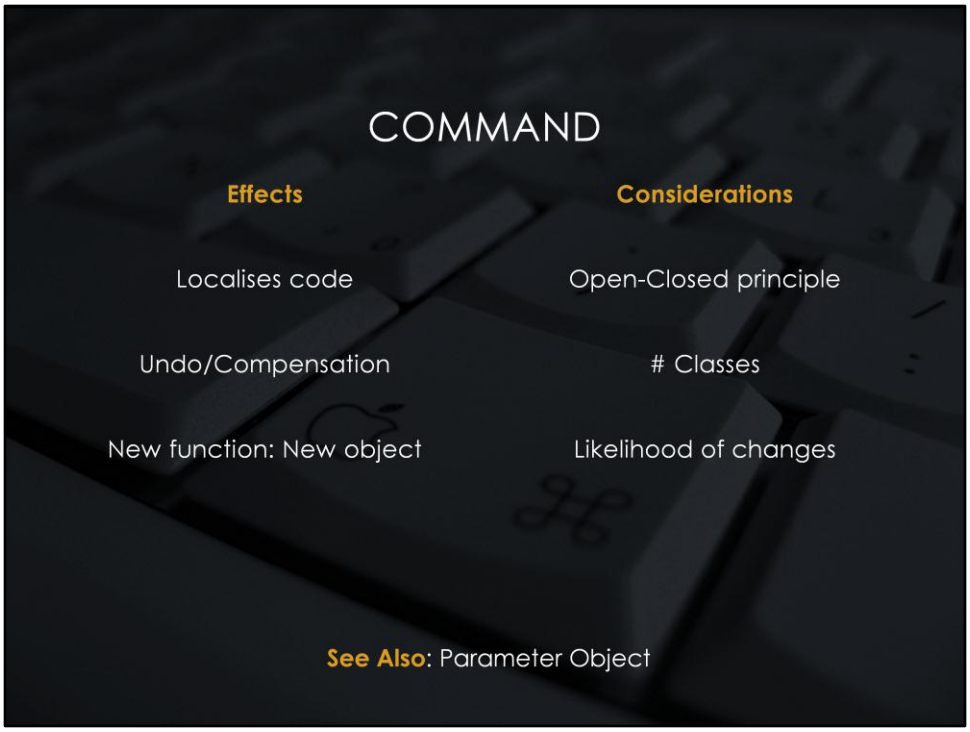
Demo
Changes to File System

We want to add a recursive delete function
Node.Delete(bool force, bool recursive) – but recursive doesn't make sense for files

How to do a wildcard delete? Why do we need to rewrite code when we already know how to delete?
How would we track which files were deleted, for reporting or undo?
How would we delete by wildcard within selected directories?

Use Delete command object, can be written without changing existing code
Adding other features is straightforward

Same idea for ChangePermissions



Effects

Instead of code being distributed across multiple locations,
it gets brought together into a single class.
Easier to understand and easier for common functions to share implementation.

As each change is triggered by a dedicated object,
you have a convenient location for storing reversal information,
allowing for undo or compensation of the command.

Creating a new command just involves creating a new object,
usually not a need to modify existing code
Less opportunity to introduce defects

Considerations

Open/Closed Principle
Objects should be open for extension closed for modification

You do end up with a large number of classes to maintain
But those classes are smaller and well focussed, with a very well defined purpose

What's most likely to change in your system –
the hierarchy of objects, or the functions themselves?

Problem

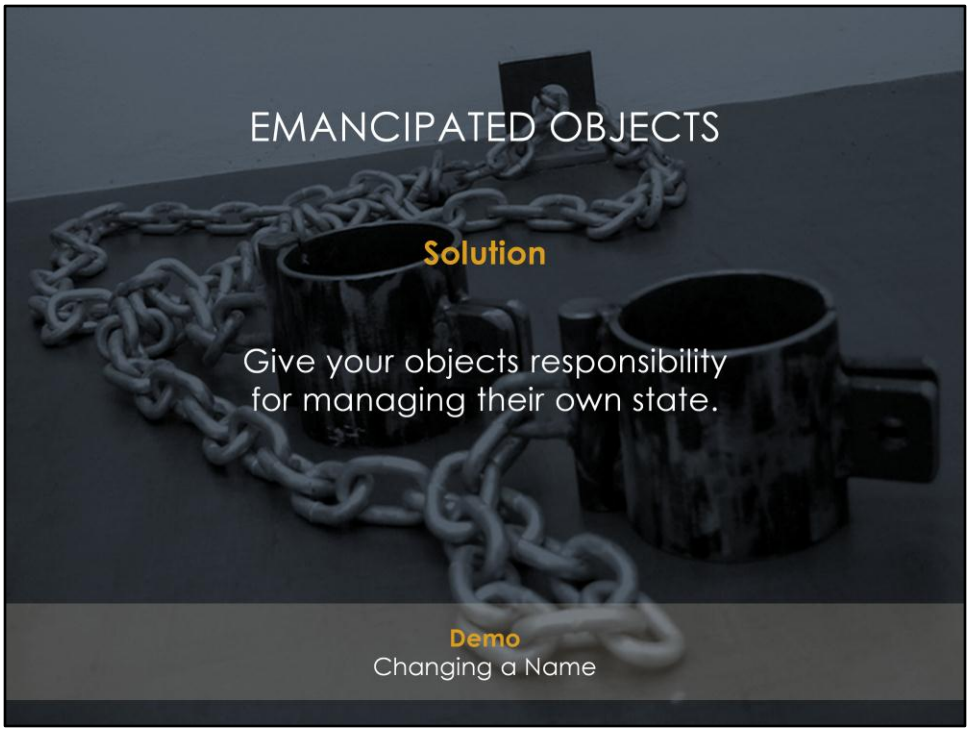
Code that directly modifies properties of objects
Lots of logic outside your domain objects making decisions about those objects

Definition

Emancipation: Freeing someone from the control of another;
especially a parent's relinquishing authority and control over a minor child
http://wordnetweb.princeton.edu/perl/webwn?s=emancipation

Image Credit

http://www.flickr.com/photos/thepen/428014152/



Solution

Move all functionality concerned with the object, especially code that changes the state of the object onto the object itself

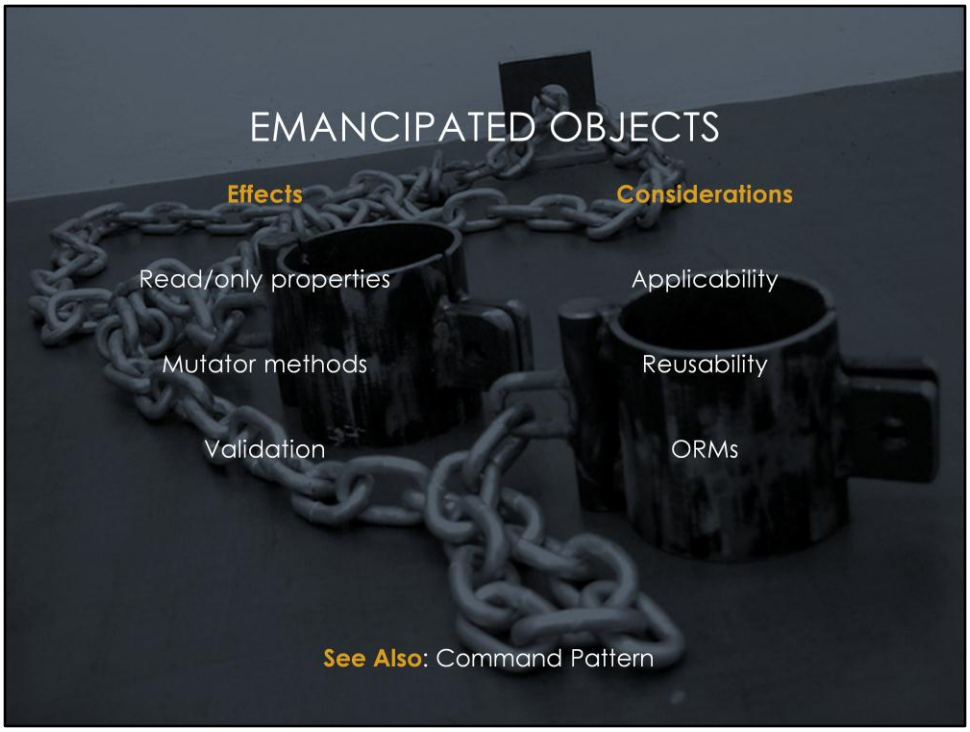
Demo

Person Object

Commentary

Don’t treat the object as a mindless slave
Instead, enlist the object as an active participant in solving the problem

Shared responsibilities leads to simpler, more robust code



Effects

Only the object may modify it's state: properties can be read/only (private setters)

Changing the object state requires calling methods for that purpose

Instead of trying to validate after the fact, don't allow domain objects to get into an invalid state in the first place. Mutator methods can fail for business reasons

Considerations

This pattern is mostly applicable to your domain layer, model of the business itself
Less appropriate for other areas, such as ViewModels using data-binding

Easier to find and reuse functionality that's present directly on the domain objects.

Many object relational mappers (ORMs) rely on having property setters in order to properly hydrate instances being loaded from the database. Sufficient if the property setter exists, it doesn't necessarily have to be public.

See Also

For complex object changes,
the command pattern makes it easy to bundle together the information required
– better than having methods with 12 parameters.

Problem

Some of your domain objects are difficult to construct because they aren't created whole at a point in time but instead are the outcome of a process.

Demo

Shopping carts and orders

Image Credit

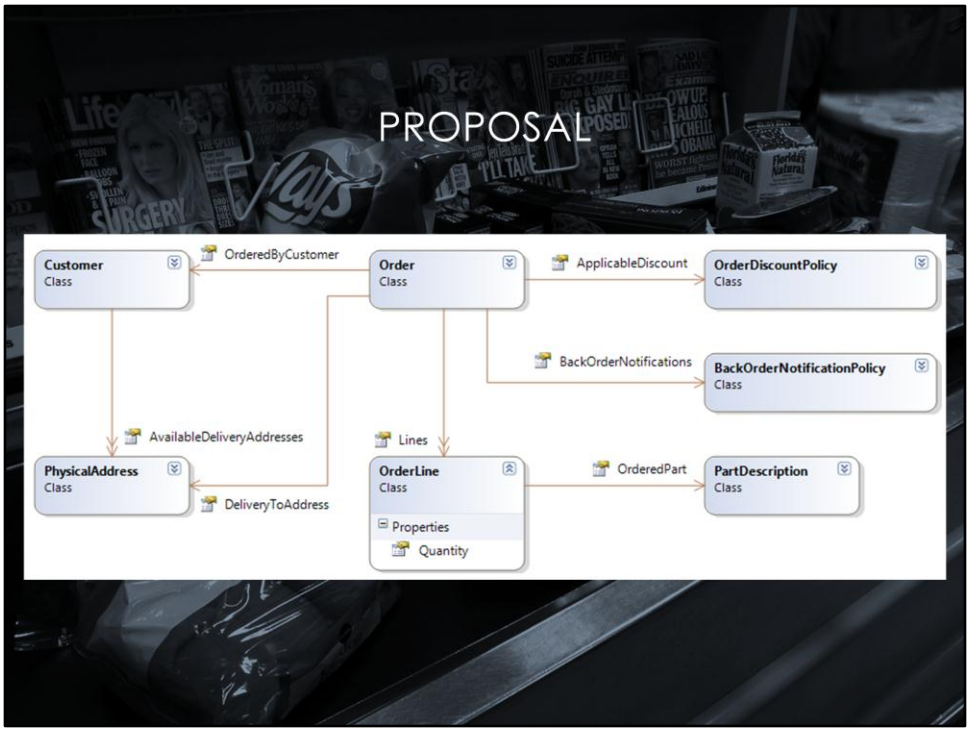
http://www.flickr.com/photos/23240330@N03/4399489577/in/photostream/

Solution

Instead of complex constructors, capture the entire process in your domain and use that as the factory

Demo

Shopping Cart



Order

Represents an Order in our system

We have the order itself
The customer who placed the order
The address of the customer to which the order should be delivered
A Discount policy applied to the order
A back order policy applied to the order

A list of parts that have been ordered, and the quantity of each

Good representation
But can only represent valid orders

Shopping cart

Isolated model – no external dependencies
Uses simple types – strings, decimals, ints
Can store invalid information – has to be valid to create the order, but is tollerant of real world issues

Effects

The proposal object is a natural location for complex validation to occur:
We can only create our object if these rules are satisfied

Creating our domain objects whole and valid reduces the number of states required, simplifying code around the constructed object

Explicitly modelling the process gives additional flexibility
E.g. a point of sale system that allows a partially processed sale to be parked on one register and activated on another

Proposals can handle information otherwise illegal
e.g. Unknown part numbers

Considerations

Not a good idea if your process is simple and quick, needs a process with some complexity

If the process takes a long time, your proposals can be stored for use as required

Problem

A complex domain object has behaviour that changes markedly (radically) with changes of state

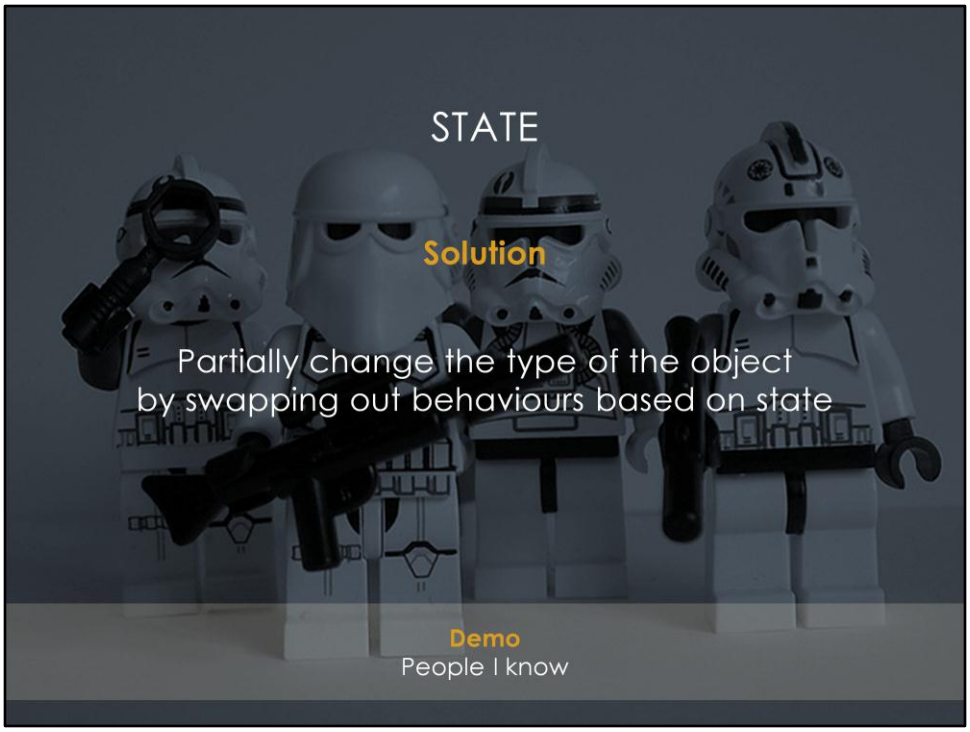
Implementation has lots of branching logic to choose the results of different requests

Demo

Person - KnownPerson

Image Credit

http://www.flickr.com/photos/chaoticgood01/3368553952/

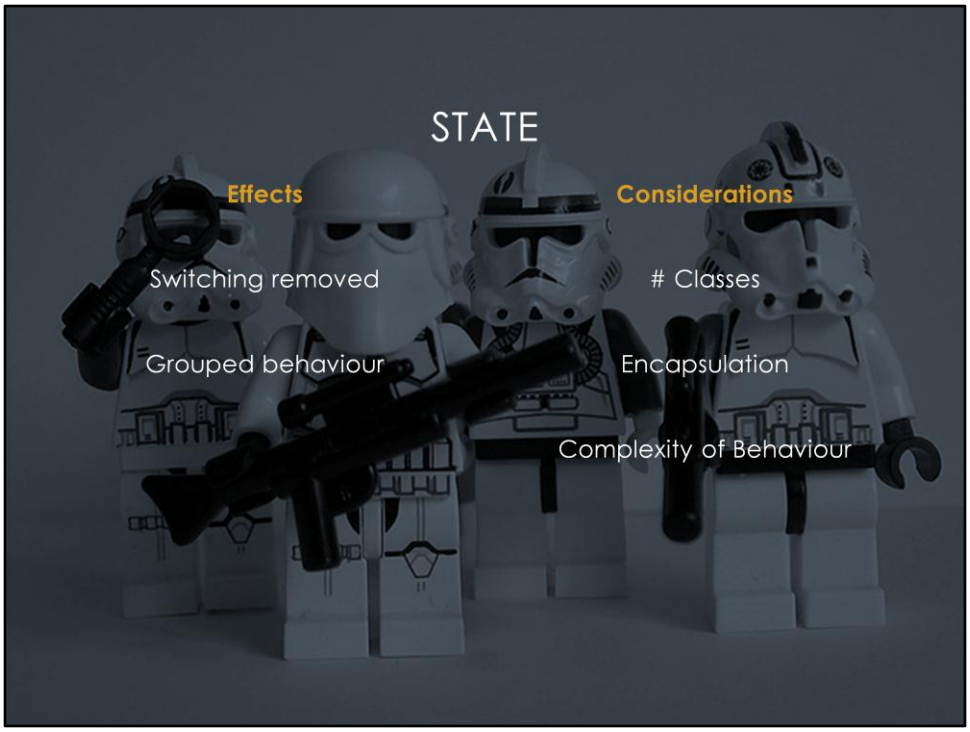


Solution

Delegate behaviour that varies to an internal object
Swap out that object with different implementations based on state

Demo

Person - KnownPerson



Effects

Switching logic is almost completely removed

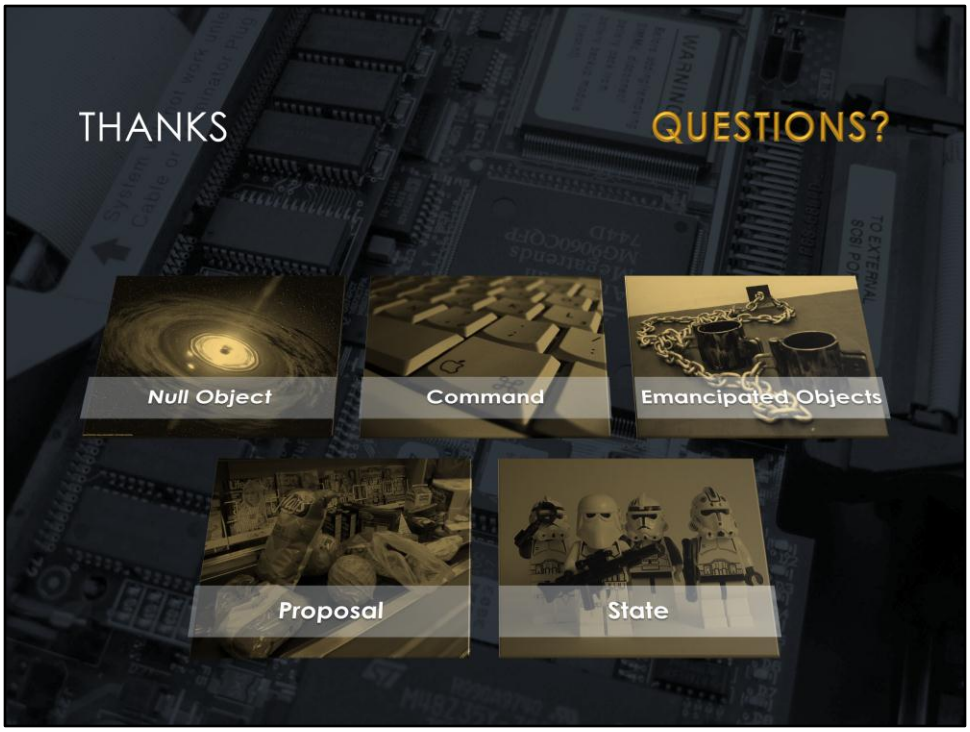
Behaviours that go-together (are all available at the same state) now live together

Increases number of classes to maintain, but each class is simpler and easier to understand

Considerations

Implementing the delegate objects as inner classes means they can access the private members of the domain class
Means you don't need to violate encapsulation

Needs behaviour that changes significantly
Not relevant for the simple case


THANKS
QUESTIONS?
Null Object
Command
Emancipated Objects
Proposal
State

twitter unrepentantgeek
email bevan@nichesoftware.co.nz
blog www.nichesoftware.co.nz