The Selfish Object

Kevlin Henney

kevlin@curbralan.com

Agenda

- Intent
  - Present a design style that addresses dependencies in a cohesive, open and manageable fashion
- Content
  - Key concept
  - Dependencies and pluggability
  - Control and flow
  - Partitioning
  - Summary

Key Concept

- Intent
  - Describe the essence and implications of the selfish object style of design
- Content
  - Selfish objects
  - Architectural consequences
  - Common versus selfish approaches

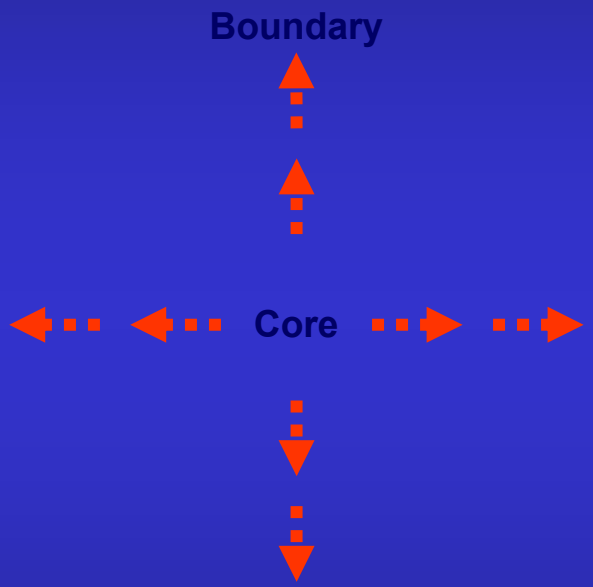
Selfish Objects

- Instead of focusing on what an object can use or even be given, focus on what it wants
  - In essence, express external dependencies by defining specific, narrow, plug-in-style interfaces
- This style is in contrast to common approach of abstracting interfaces from implementations
  - Although better than not abstracting interfaces at all, such an approach often ends up presenting a broad and unfocused façade rather than a specific and focused usage interface

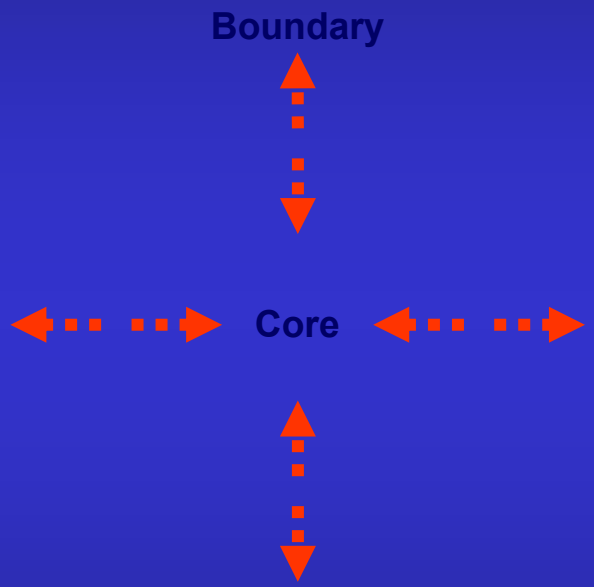
Architectural Consequences

- In the large, object self-centredness leads to a highly localised, open and testable architecture
  - Consistent parameterization from above, across packages and layers as well as objects, results in a more inverted layering, keeping the core domain model separated from the plumbing
- Locality and loose coupling are important considerations in architecture
  - Respecting them can simplify comprehensibility, extensibility, changeability, testability, etc.

Common versus Selfish Approaches



The common approach to layering can result in the core concepts of an application depending, ultimately, on the I/O (streams, UI, database, etc.).



The selfish approach makes the core concepts dictate what they need from other parts of the system.

Dependencies and Pluggability

- Intent
  - Introduce dependency management techniques that promote loose coupling and pluggability
- Content
  - Dependency management
  - Singletons and other globals
  - Parameterize from Above
  - Dependency Inversion
  - Inversion Layer

Dependency Management Problems

- You have a dependency management problem if you find that...
  - You cannot unit test your application's core without connecting to a database or touching external config
  - A single class hierarchy dominates your code and becomes the focus (and pit) for all changes
  - You have Singletons all over your code, ostensibly introduced for expediency and to simplify code
  - You have cycles between packages or headers, or you are unsure what a cyclic dependency is

Dependency Management

- The *dependency horizon* should be kept close
  - A component's total dependency set is formed by following the dependencies from the component until they either run out or hit the 'system' libraries
  - This limit or boundary is the dependency horizon
- Interfaces, formal or otherwise, often play a key role in loosening a system's coupling
  - Interfaces may be expressed using a variety of mechanisms, depending on the technology

Singletons and Other Globals

- Singleton is a common source of dependency-related problems
  - It is normally used by coincidence, it introduces a centralised point of coupling, it complicates testing, and it comes with various lifecycle-related problems
- Consider avoiding modifiable *static* data — and even reducing use of immutable *static* data
  - This moratorium on *static* includes the Monostate pattern, which is also known as the Borg pattern... which tells you everything you need to know

Hardwired versus Pluggable

- *Pluggability* describes a design property that is the opposite of *hardwired*
  - Hardwiring attempts to nail an assumption in place, which is a problem if the assumption represents a variable or critical dependency
- Pluggable designs are more testable and adaptable than hardwired designs
  - They also emphasise locality in a design by more explicitly dividing concerns between the pluggable and the kernel elements of a design

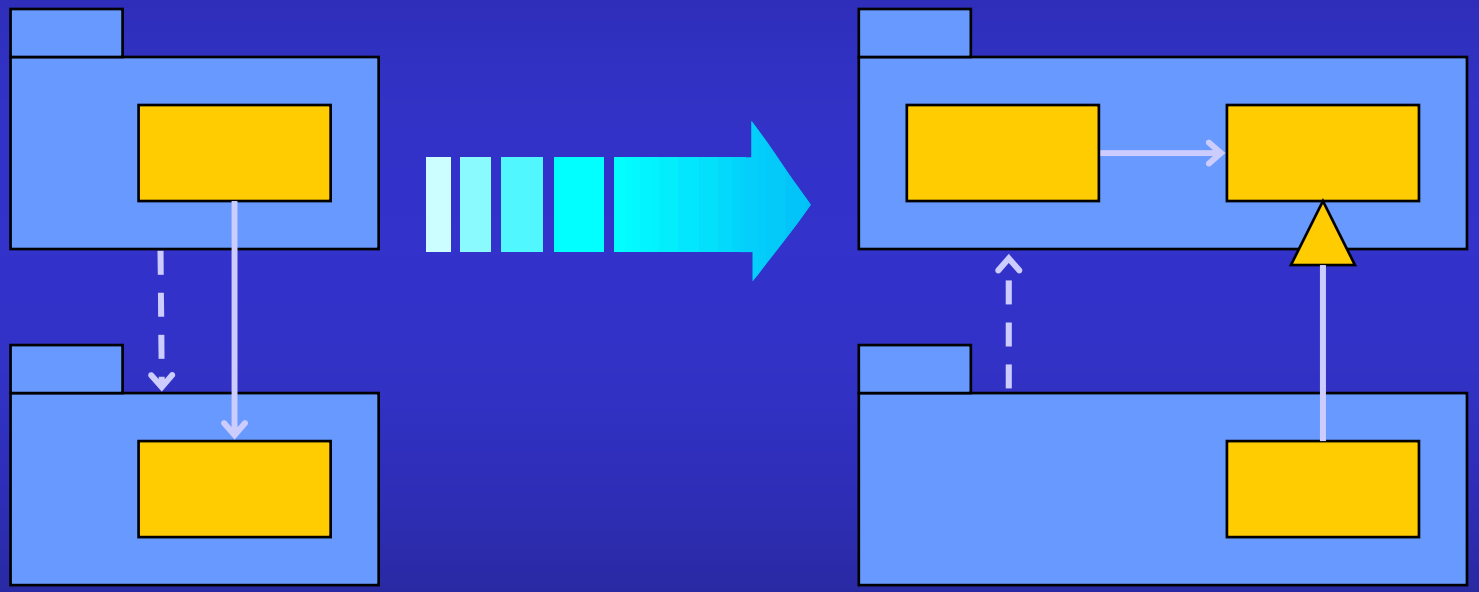
Parameterize from Above

- Pass in config parameters rather than having them global (e.g., Singleton) or pulled in
  - Communicate through constructor arguments, method arguments or generic parameters, as appropriate
  - Decentralise configuration constants
- Callout interfaces define the configurable dependencies of each part
  - E.g., the Context Object, Plug-In and Strategy design patterns or the Test Double testing pattern

Inversion of Dependencies

- Dependency Inversion is a technique for rearranging (reversing) dependencies in code
  - Normally based on introducing an interface of some kind that plays the role of a plug-in point
  - Inverting dependencies can be used to break cyclic dependencies between packages by containing the cycle within a package
  - Inversion of dependencies often leads to inversion of control, i.e., plug-ins lead to callbacks and the dependency horizon becomes an event horizon

Dependency Inversion in Practice



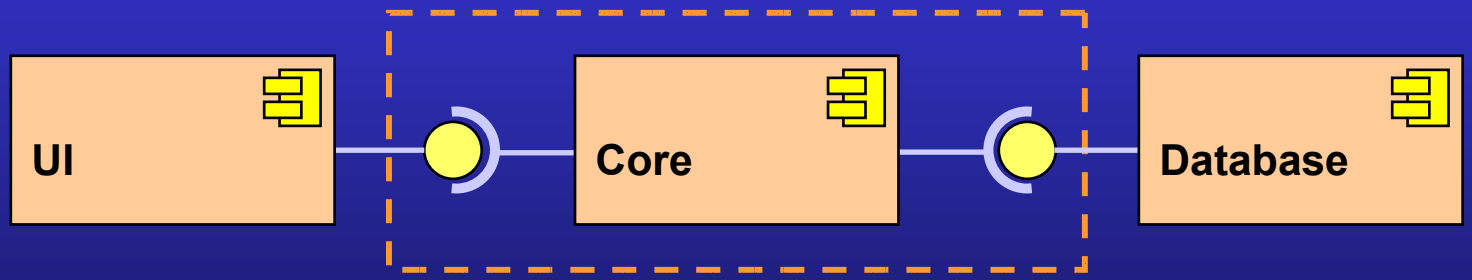
Dependency Inversion allows a design's dependencies to be reversed, loosened and manipulated at will.

Transitive Dependencies

- Some partitions appear encapsulated, but hidden dependencies still create coupling
  - Traditional layering partitions and groups immediate concerns well enough, but it does not fully separate them — the transitive dependencies can make for a distant dependency horizon

Inverted Layering

- It is possible to invert dependencies in classic layered architectures
  - An Inversion Layer introduces a separation between concepts and mechanisms
  - Simplifies testing and parallel development



Inversion Layers

- In practice, an Inversion Layer is often also an adaptation or bridging layer
  - The core owns the interfaces ('outerfaces') it uses
  - The boundary code wraps and encapsulates external dependencies
  - The code in between, which bridges boundary and core, is normally based on adaptors
- Amount of executable code is often conserved
  - It is the partitioning of the code that has changed

Control and Flow

- Intent
  - Focus on control flow model and location of active control in a design
- Content
  - Inversion of control
  - Dependency Injection
  - Callback mechanisms
  - Micro-Kernel
  - Interceptor

Inversion of Control

- A description of the control flow relationship between one component and another
  - A lower-level component calls out to a higher-level component, rather than the higher-level component calling the lower-level one
  - Often a result of dependency inversion
- Inversion of control is based on the Hollywood principle: "Don't call us, we'll call you"
  - Common in framework designs that use a push rather than a pull approach to event handling

Applications of Inversion of Control

- Inversion of control makes for a more event-driven programming style
  - Aligns control flow with event flow
  - Aligns event horizon with dependency horizon
- It is found in many common patterns
  - Observer propagates event notification
  - Enumeration Method is used for iteration
  - Lifecycle Callback maps lifecycle events to callbacks
  - Visitor complements class hierarchy behaviour

Dependency Injection

- Principle of separating configuration from use and injecting the configuration dependencies
  - Used in lightweight component container models
  - Although it uses inversion of control, Dependency Injection it is not a synonym — inversion of control is a broader concept, and the key to Dependency Injection is the inversion of dependencies
  - An assembler role is responsible for configuring objects, whether through constructor arguments or 'injecting' methods

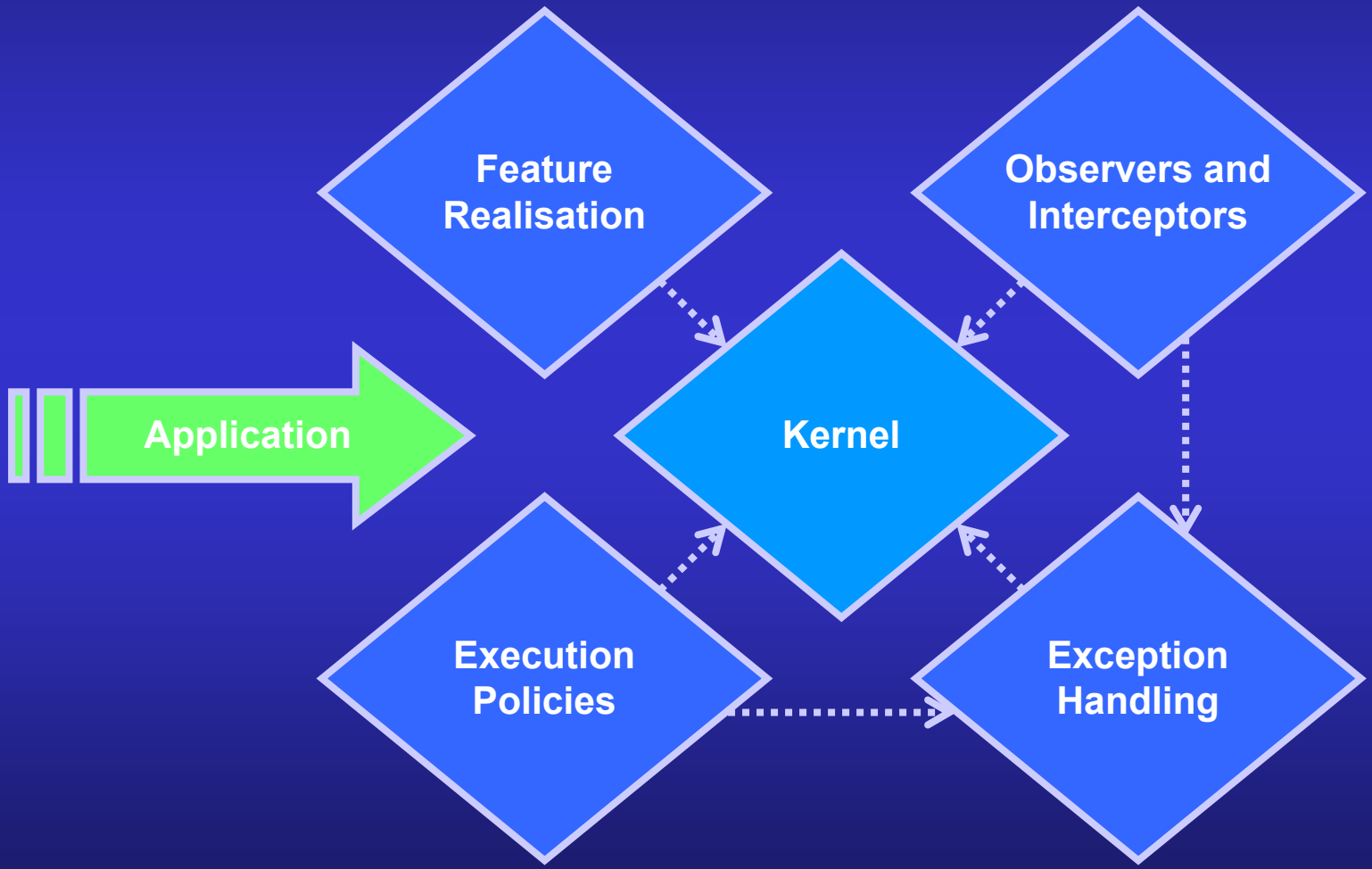
Callback Mechanisms

- Callback mechanisms depend on the language and the desired
  - A method selector, such as a delegate or function pointer, allows plugging in of a single method
  - Interfaces — as in the *interface* construct — supports a broader interface in statically typed languages
  - A dynamically typed protocol may be a more normal approach for a language, or it may be possible through reflection
  - Templates and other generic forms are also usable

Micro- (and Nano-) Kernels

- A Micro-Kernel approach partitions control logic, not just concepts
  - Common logic and concepts are extracted into the kernel (or engine) and details are relocated within plug-ins
  - A Nano-Kernel is a more minimal and localised application of the same idea
- The kernel works in terms of out-bound callback interfaces on plug-ins
  - The domain model itself may well be a plug-in

Anatomy of a Plug-In Architecture

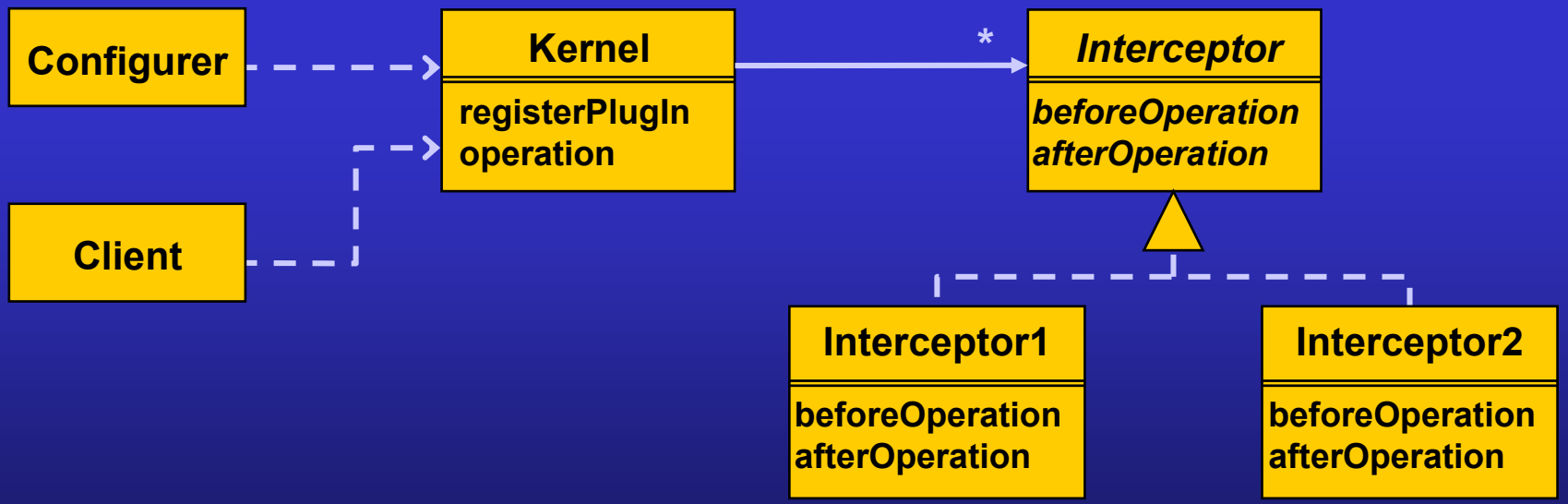


Interception

- How can a design be cleanly extended to accommodate extra-functional features?
  - Modifications of behaviour, such as filtering, or addition of features, such as logging
- Favour an Interceptor-based approach rather than an adaptation approach
  - An Interceptor is more configurable and less intrusive than many other approaches, such as Template Method (or NVI), that are hardwired

Interceptor

- An object, component or framework's basic behaviour can be extended
  - Interception plug-ins are called on certain actions



Partitioning

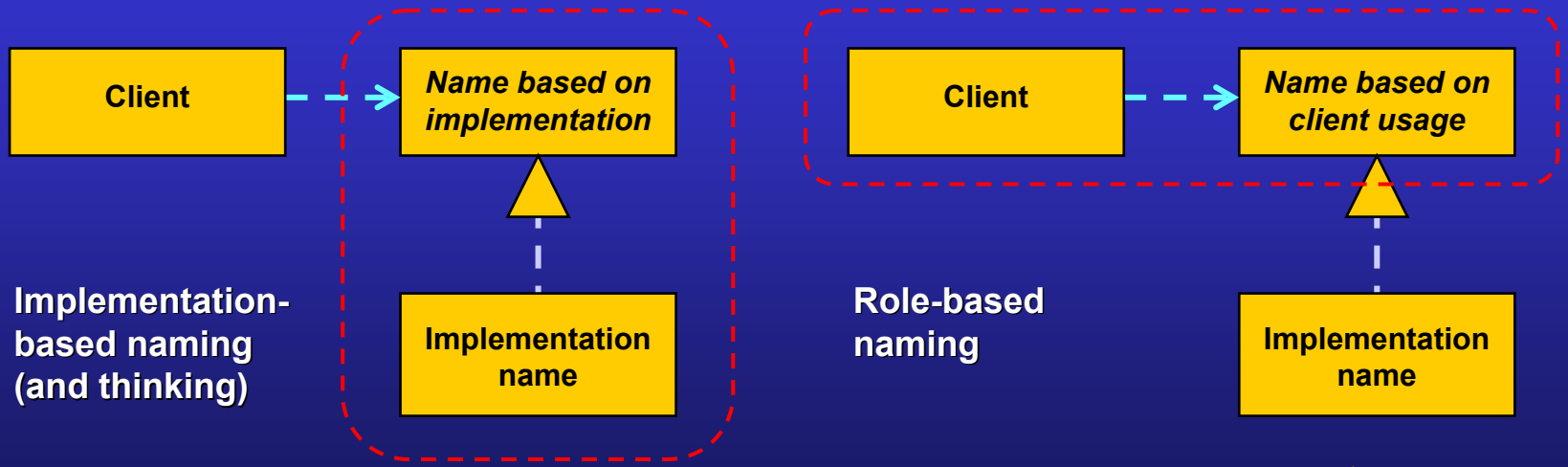
- Intent
  - Describe effective approaches for broader partitioning of a system's classes and components
- Content
  - Interface separation, role partitioning
  - Role-based naming
  - Partitioning by role
  - Partitioning for stability

Interface Separation, Role Partitioning

- One of the most common forms of partitioning is separating interface from implementation
  - "Program to an interface, not an implementation"
- Focus on object roles not object classes
  - A role defines a selfish perspective: how an object is to be used, not what it is or how it is made
  - Role-based design tends to give a cleaner separation of concerns and more focused interfaces
  - Class-centric design tends to give coarser-grained, implementation-focused classes

Role-Based Naming

- When extracting interfaces, focus on the usage and not on the implementation
  - Otherwise the interface is likely to be broader than necessary, and with an implementation- based name (a common problem with *I*-prefixing)



Partitioning by Role

- Role partitioning applies more broadly than just interface separation and segregation
  - Although this is perhaps one of the most visible applications of role partitioning
- Packages can be organised with respect to role
  - Packages should be cohesive with respect to usage and purpose
  - Packages should not be partitioned with respect to coincidental criteria, such all classes in a package being exceptions or value objects

Partitioning for Stability

- Different parts of a system are subject to different rates of development change
  - Layering should respect such change, so that less stable elements depend on more stable elements, and not vice versa
  - Stability can be tracked over a code base's lifetime, and the code can refactored accordingly
  - Dependency Inversion is a useful technique for rearranging dependencies along the lines of stability, such as introducing Inversion Layers

Summary

- A selfish object approach separates and localises concepts and dependencies
  - Simplifies modification, extension, and incremental development and testing
- In the large, the approach leads to inversion layers and an architecture with high locality
  - An onion-layered view centred on the domain model, rather than a stack-layered view, is often a more appropriate visualisation