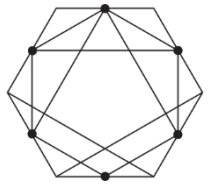
CLEAN CODE AND MORE

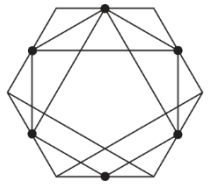
SOLID / GRASP / HexA



SOLID

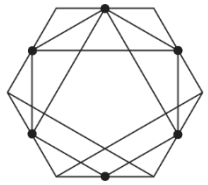
A collection of five guidelines collected by Robert C. Martin (Uncle Bob)

- Single Responsibility Principle (SRP)
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle



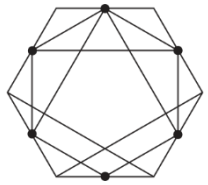
SINGLE RESPONSIBILITY PRINCIPLE

- For each class there should only be a single reason to change
- The name is not quite correct: SRP does not state that each class may only have a single responsibility)
- Closely connected to high cohesion / low coupling
- (Complementary to Information Expert from GRASP)



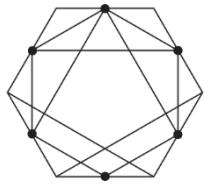
OPEN/CLOSED PRINCIPLE

- Software artifacts (classes, modules, functions, etc.) should be *open for extension but closed for change*
- More simply: it should be possible to modify the behavior of objects or introduce new kinds of objects without touching the existing code
- More concretely:
  - Implementation against interfaces, not concrete classes
  - Use of method overriding (polymorphy) instead of if or switch statements
  - (No use of checked exceptions)
- More general: Protected Variation, Information Hiding



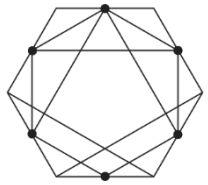
LISKOV SUBSTITUTION PRINCIPLE

- Instances of subclasses can take the role of superclasses without changing program behavior
- More relaxed version: without causing problems
- This is a fundamental feature of class hierarchies



INTERFACE SEGREGATION PRINCIPLE

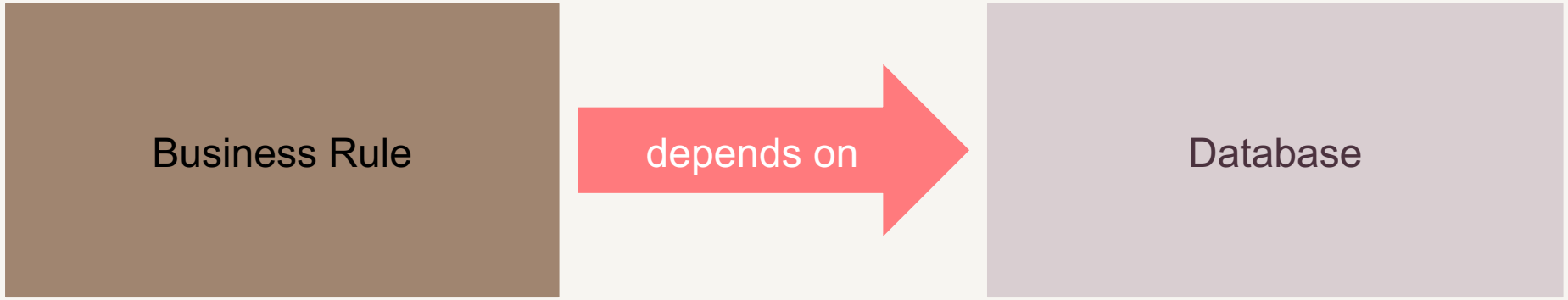
- No client of a class C should depend on methods it doesn’t use
- If that is not the case
  - Divide the interface of C into multiple independent interfaces
  - Replace C in each client with the interfaces actually used by the client



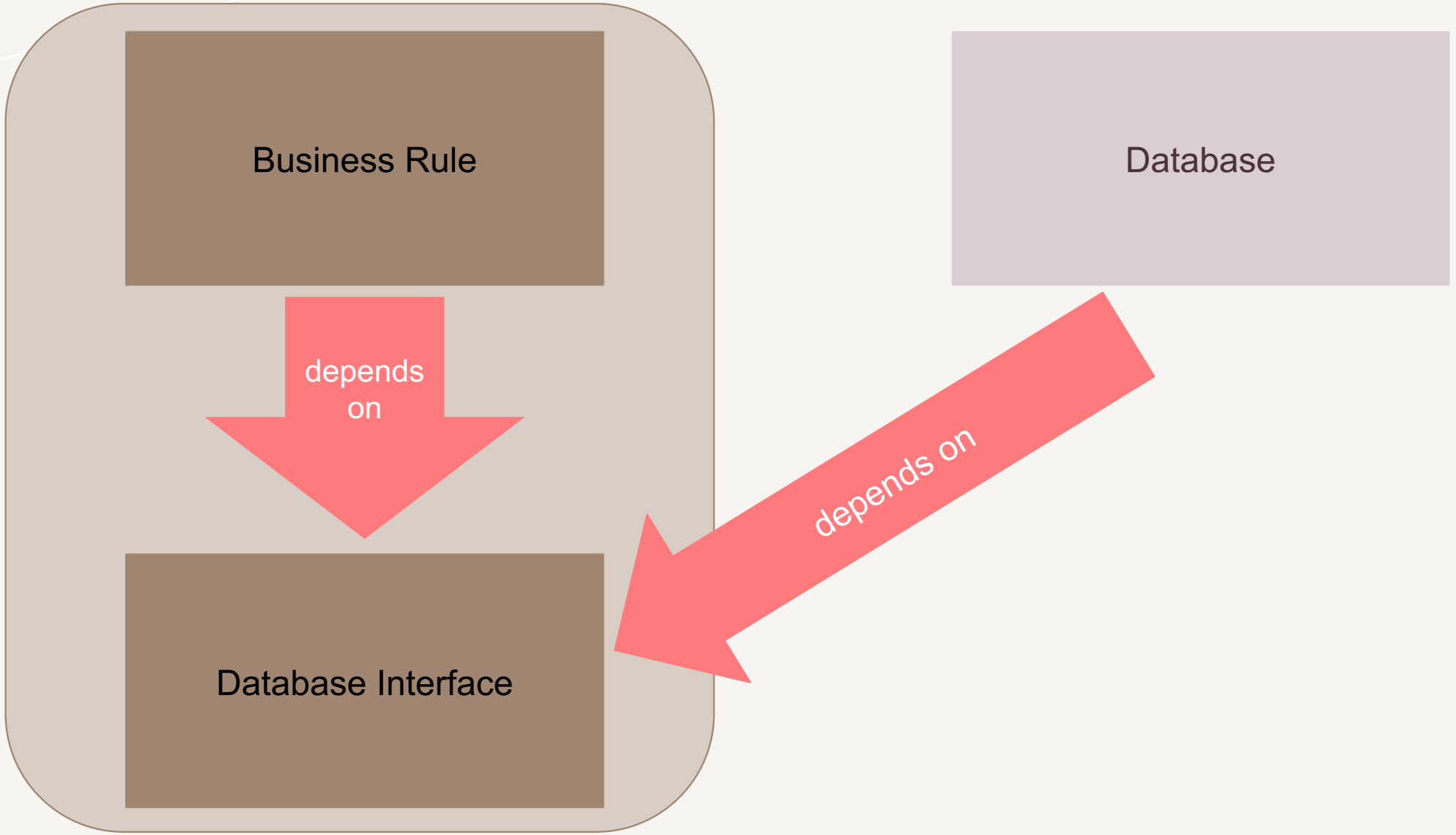
DEPENDENCY INVERSION PRINCIPLE

- The core functionality of a system does not depend on its environment
  - **Concrete artifacts depend on abstractions** (not *vice versa*)
  - **Unstable artifacts depend on stable artifacts** (not *vice versa*)
  - **Outer layers** of the architecture **depend on inner layers** (not *vice versa*)
  - Classes/Modules depend on abstractions (e.g., interfaces) not on other classes/modules
- Dependency inversion achieves this by introducing interfaces that “reverse the dependencies”

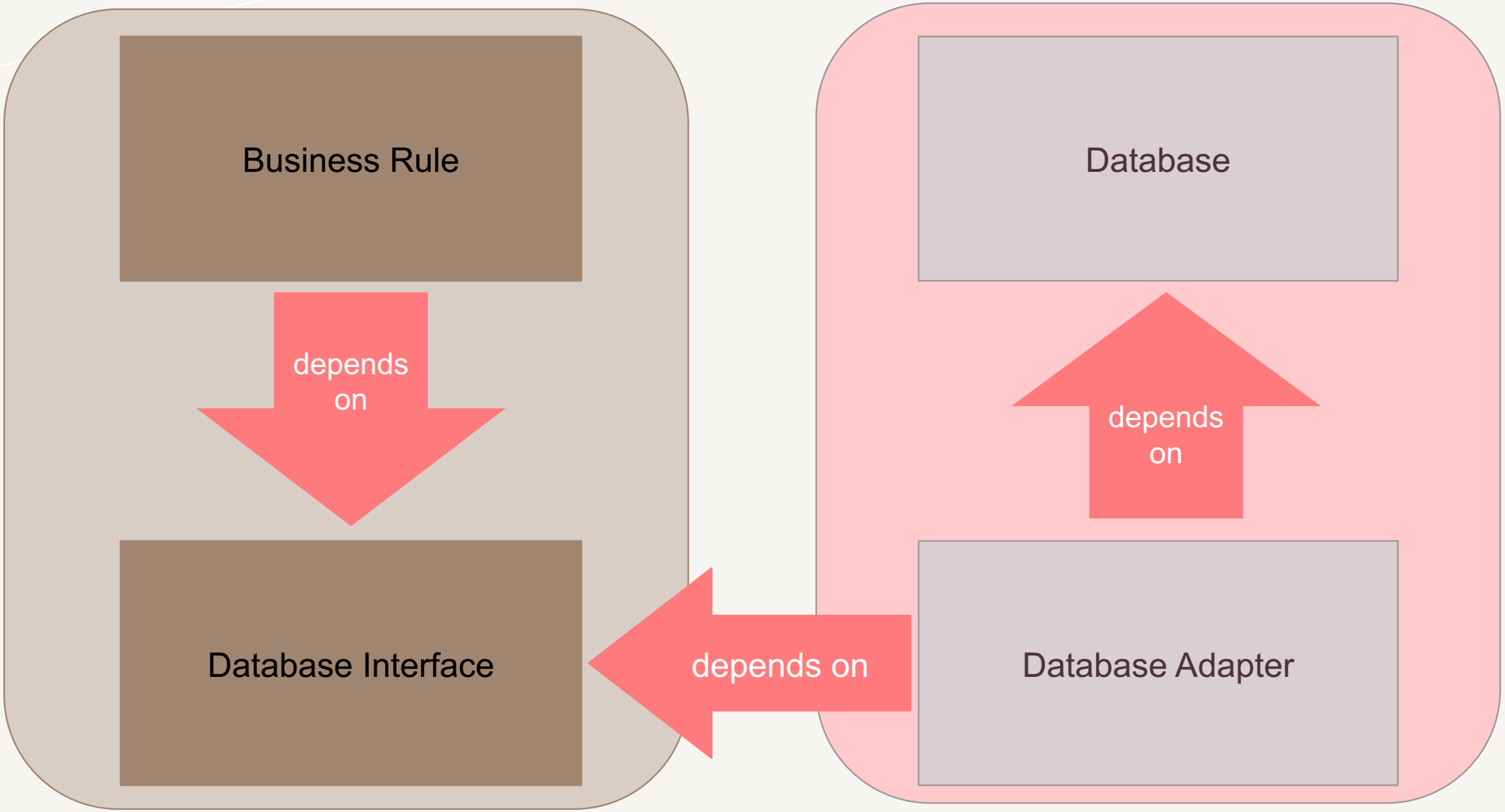
Bad Dependency

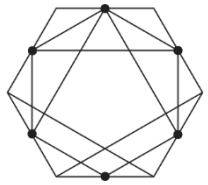


Dependency Inversion (Step 1)



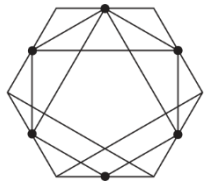
Dependency Inversion (Step 2)





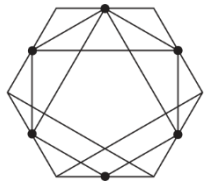
GRASP

- General Responsibility Assignment Software Patterns
- A collection of patterns, practices and principles published by Craig Larman



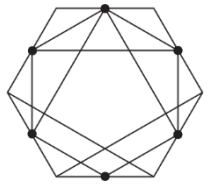
GRASP

- High Cohesion
- Low Coupling
- Information Expert
- Creator
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations



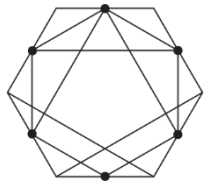
HIGH COHESION

- Measures how well different parts of an artifact fit together
- High cohesion simplifies evolution, reuse, testing, performance
- Low cohesion makes it difficult to understand the code or to figure out where to make changes
- The **negative effect** of low cohesion is **large**
- It is **difficult** to move a system with low cohesion to a state where it has more cohesion
- Code smells: shotgun surgery, divergent change
- Related: SRP, Command/Query Separation



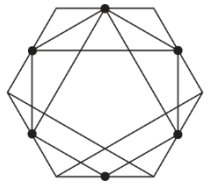
HIGH COHESION AND TESTS

- Low cohesion causes the system functionality to be “smeared” over the whole system
- This often leads to high coupling and large classes
- These classes are difficult to move into a desired state
- “Smeared functionality”
  - Makes unit testing difficult
  - Forces the use of many test doubles
  - Reduces the value of unit tests as documentation



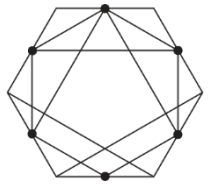
LOW COUPLING

- Coupling: measure of dependencies between different artifacts
- Exists for data, operations, names, etc.
- Low coupling is desirable from stable components to unstable components
- High coupling
  - Prevents us from understanding parts of the system in isolation
  - Causes every change to proliferate throughout the system



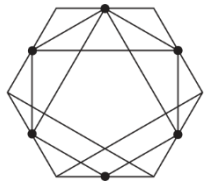
LOW COUPLING

- High coupling to stable components (e.g., the standard library) is impossible to avoid and not a problem
- High coupling on a local level is typically easy to remove and not a big problem
- High coupling on a large context or in the architecture is difficult to remove and thus a major concern when developing software
- Code smells: Shotgun Surgery, Divergent Change
- Related: Dependency Inversion Principle, Law of Demeter



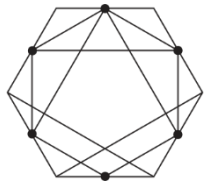
HIGH COUPLING AND TESTING

- High coupling makes testing difficult:
  - Dependencies have to be isolated using Dependency Injection to make the system testable
  - Many classes have to be instantiated for each test (see: Setup Sermon)
  - Many test doubles are necessary to break dependencies



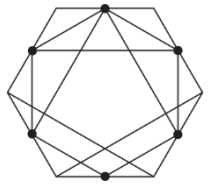
INTERMISSION: RESPONSIBILITY-DRIVEN DESIGN

- Low cohesion and high coupling are two major problems of many software systems
- One way to work against these effects is to consciously assign responsibilities to classes and modules (responsibility-driven design)



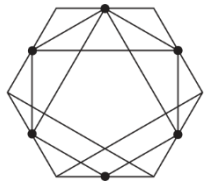
INFORMATION EXPERT

- Services / operations should be close to the data on which they operate
- Find the “information expert” that keeps this data and add the functionality there
- Related: encapsulation, abstract data types; tell, don’t ask; cohesion, coupling
- Testing: combination from data and operations simplifies unit testing



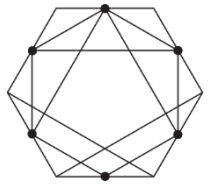
INTERMISSION: TELL, DON’T ASK

- Decisions and activities that depend only on the state of object X should be performed by X itself
- “Tell, don’t ask” doesn’t mean that queries should be replaced by commands!



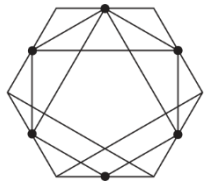
CREATOR

- An object C (the creator) creates an object X if
  - C contains X as aggregation
  - C is the only user of X
  - C holds the data to initialize X



CREATOR

- Related: dependency injection (as alternative)
- Relationship to testing: neutral/ambivalent



CONTROLLER

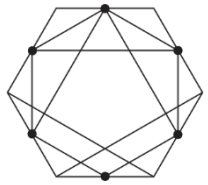
For each module/subsystem: external messages are handled by **Controller** objects that

- Are not part of the UI
- Cover one subsystem or use case each

The controller is the first object after the UI to handle events/messages. It coordinates the system.

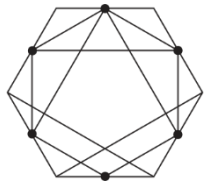
The controller is a façade, i.e., it delegates its work to other objects.

A use-case controller always processes a complete use case (but controllers can process more than one use case).



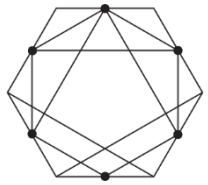
CONTROLLER

- Related: façade pattern (domain façade), domain controller
- See hexagonal architecture: Controllers are the ports in the hexagonal architecture.
- Tests: controller provide a central interface for single subsystems or use cases



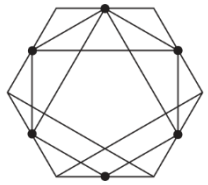
POLYMORPHISM

- Polymorphic operations describe similar (but not identical) behaviors that may change depending on the type of an object
- Typically no switch between behaviors during run-time
- Related: Type-hierarchy rules (isa-rule, only leaves are concrete, enforcement of invariants for subtypes by supertype)
- See also: LSP, Open/Closed principle



PURE FABRICATIONS

- A class that does not appear in the domain model
- Typically a counterweight to the Information Expert that wants to concentrate functionality in a single class
  - Example:
    Database-functionality in domain classes
    Consistent with Information Expert
    But: low cohesion, high coupling
    Introduction of Data Access Objects (a pure fabrication)



INDIRECTION

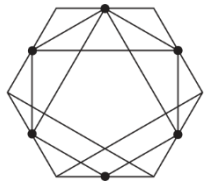
“Every problem in computer science can be solved by adding another layer of indirection”

– David Wheeler

“Except too many layers of indirection”

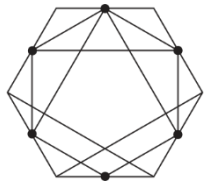
– N.N.

This is a very frequent pattern



INDIRECTION

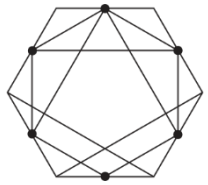
- Very frequent pattern on every layer
  - Operating system
  - Virtual machines
  - Polymorphic method calls
- Testing:
  - Indirections are seams that can be used for testing purposes



PROTECTED VARIATIONS

How do we design components so that variation or evolution points don’t have an undesirable impact on other components?

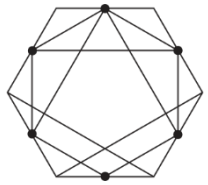
Another ubiquituous mechanism



PROTECTED VARIATIONS

Solution:

- Identify the points where variation or evolution may occur
- Introduce a stable interface protecting these points
- These interfaces are frequently pure fabrications
- Often these interfaces introduce an indirection

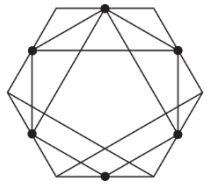


PROTECTED VARIATIONS

Very frequently applied:

- Private attributes with getters/setters
- Interfaces, polymorphism
- Virtual Machines
- Standards

- Protected Variations are often entry points for tests
- But often additional tests have to be written to ensure that the system still works when the variability is actually used



SOLID VS. GRASP

Single Responsibility	-	Information Expert
Open/Closed	-	Polymorphism(?)
LSP	-	Polymorphism
Interface Segregation	-	Controller(?)
Dependency Inversion	-	High Cohesion
Dependency Inversion	-	Low Coupling