

General Responsibility Assignment Software Patterns

Dalibor Dvorski (ddvorski@conestogac.on.ca)

School of Engineering and Information Technology

Conestoga College Institute of Technology and Advanced Learning

POS NextGen Project

- the first two-day requirements workshop is finished.
- three of the twenty use cases – those that are the most architecturally significant and of high business value – have been analyzed in detail, including, of course, the Process Sale use case.
- programming experiments have resolved the show-stopper technical questions, such as whether a UI library will work on a touch screen.
- the chief architect and business agree to implement and test some scenarios of Process Sale in the first three-week timeboxed iteration.
- other artifacts have been started: Supplementary Specification, Glossary, and Domain Model.
- the chief architect has drawn some ideas for the large-scale logical architecture, using UML package diagrams.

Responsibility-Driven Design

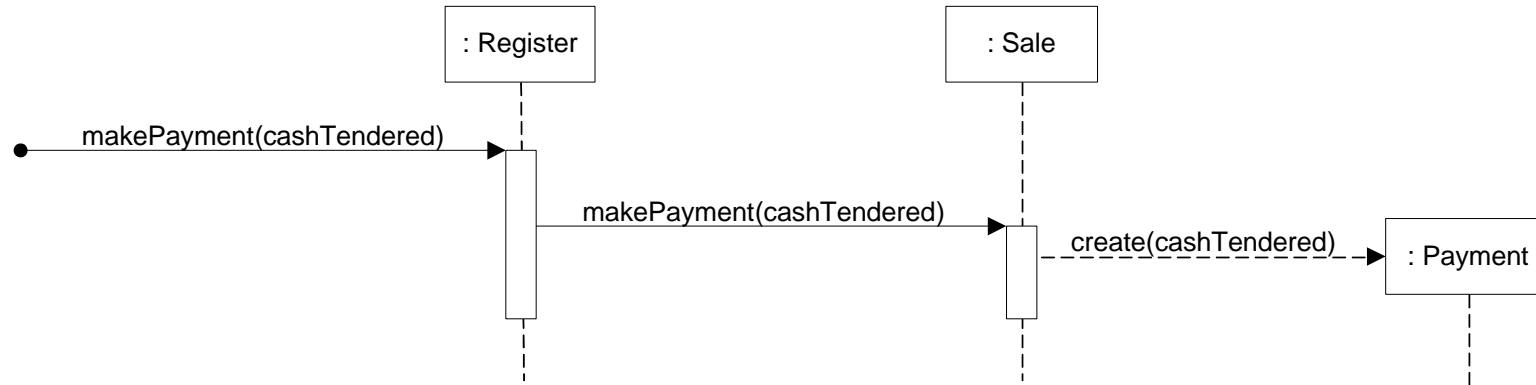
responsibility-driven design (RDD):

- think of software objects as having responsibilities – an abstraction of what they do.
 - *responsibility*:
 - a contract or obligation of a classifier. (Object Management Group)
 - two types: doing and knowing.
 - *doing responsibilities of an object include*: doing something itself (e.g. creating an object or calculation); initiating action in other objects; controlling and coordinating activities in other objects; e.g. 1. creating an object or calculation; e.g. 2. a Sale is responsible for creating SalesLineItems.
 - *knowing responsibilities of an object include*: knowing about private encapsulated data; knowing about related objects; knowing about things it can derive or calculate; e.g. a Sale is responsible for knowing its total.
- related to the obligations or behaviour of an object in terms of its role.

Responsibility-Driven Design

- includes the idea of collaboration.
 - responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects.
 - e.g. the Sale class might define one or more methods to know its total, such as getTotal.
 - to fulfill that responsibility, a Sale object may collaborate with other objects, such as sending a getSubtotal message to each SalesLineItem instance asking for its subtotal.
- a general metaphor for thinking about OO software design.
 - think of software objects as similar to people with responsibilities who collaborate with other people to get work done.
 - leads to viewing OO design as a community of collaborating responsible objects.

Responsibility-Driven Design



One thing that this figure indicates is that `Sale` objects have been given a responsibility to create payments using the `Payment` class, which is concretely invoked with a `makePayment` message and handled with a corresponding `makePayment` method. Furthermore, the fulfillment of this responsibility requires collaboration to create the `Payment` object and invoke its constructor.

General Responsibility Assignment Software Patterns

General Responsibility Assignment Software Patterns (GRASP):

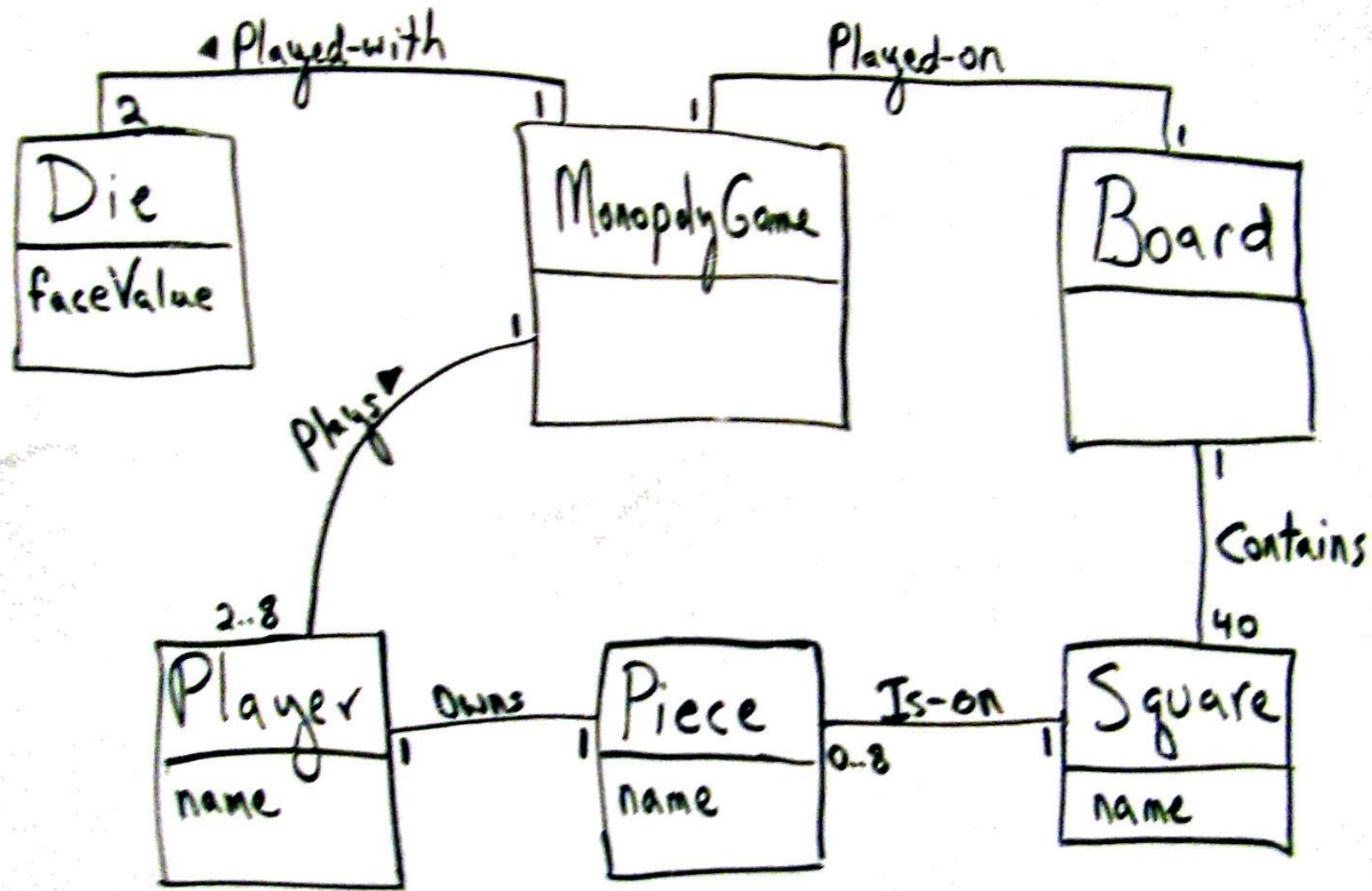
- design patterns that form a foundation for designing OO systems.
 - *GRASP design patterns*: Creator, Information Expert, Low Coupling, Controller, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations.
- name suggests importance of grasping certain principles to successfully design objects.
 - understanding and being able to apply the ideas behind GRASP – while coding or while drawing interaction and class diagrams – enables developers new to object technology needs to master these basic principles as quickly as possible.

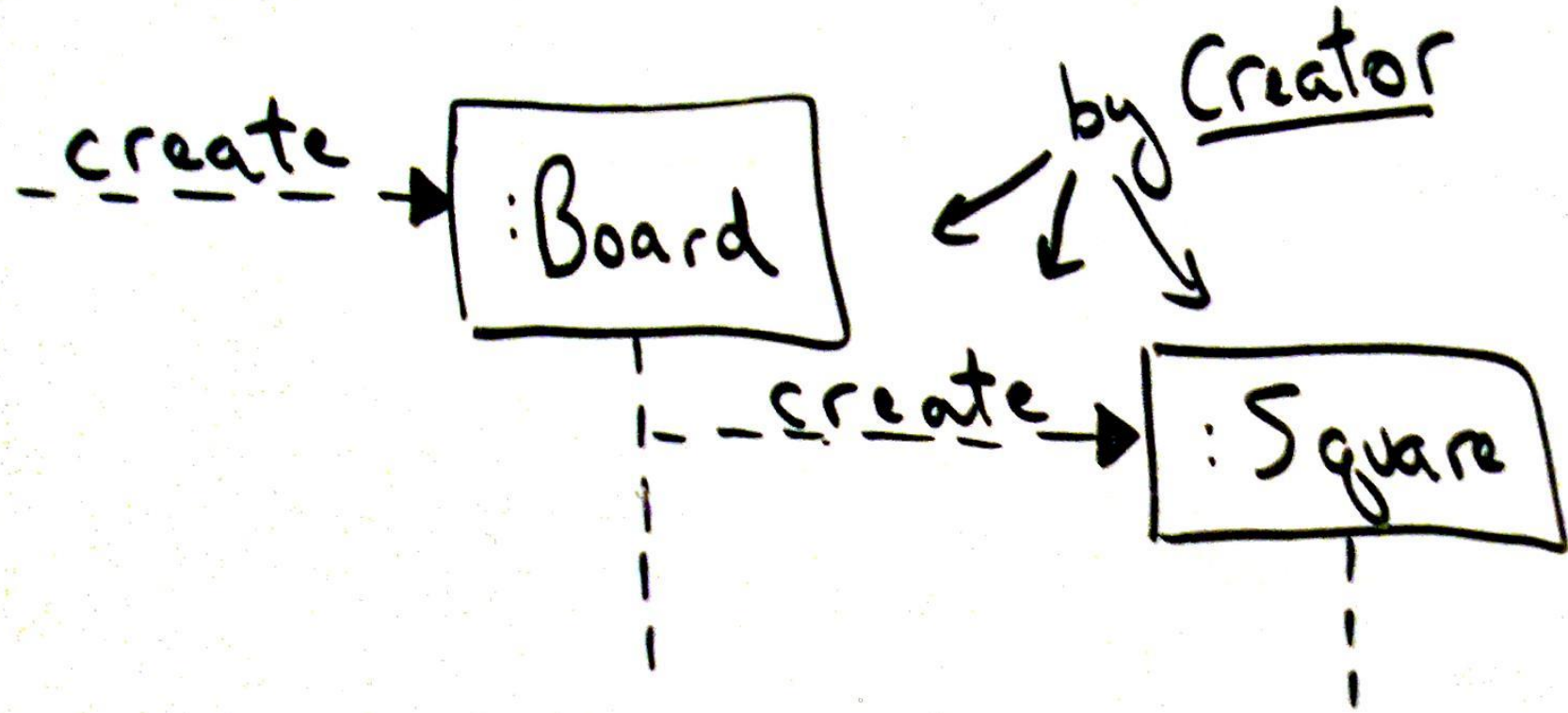
Creator Pattern

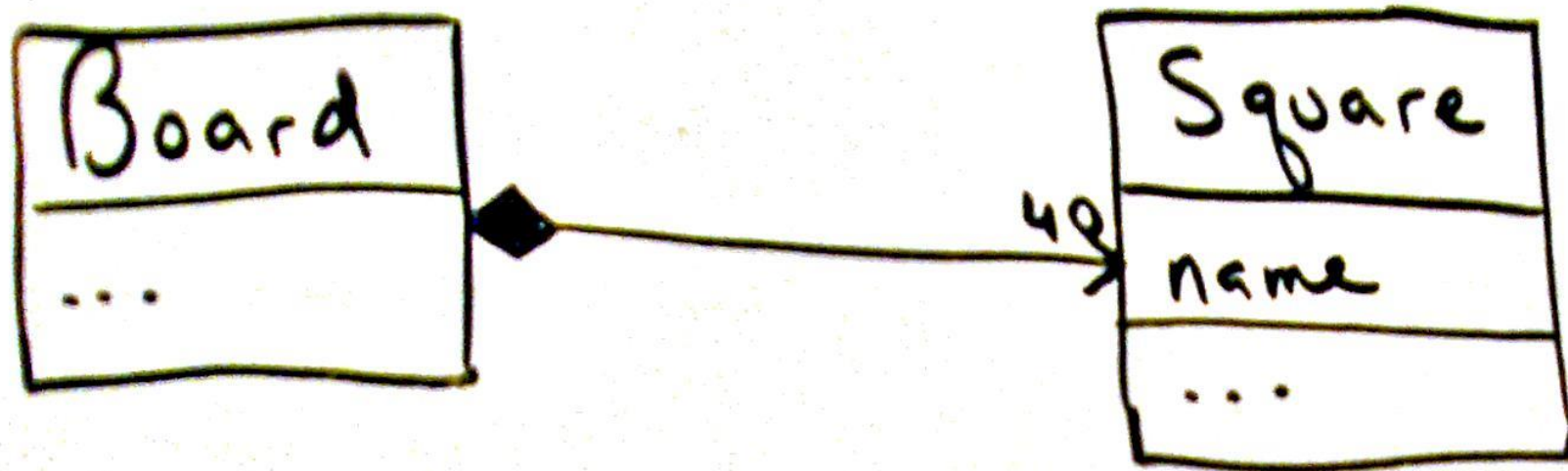
- one of the first problems you have to consider in OO design is: Who creates some object?
 - this is a 'doing' responsibility.
 - e.g. in the Monopoly case study, who creates a Square object?
 - any object can create a Square object, but what would OO developers choose and why?
 - ~~a Dog object creates a Square object.~~ a Board object creates a Square object.

Problem: Who creates an instance of class C_0 ?

Solution: Assign class C_1 the responsibility to create an instance of class C_0 if C_1 uses C_0 .





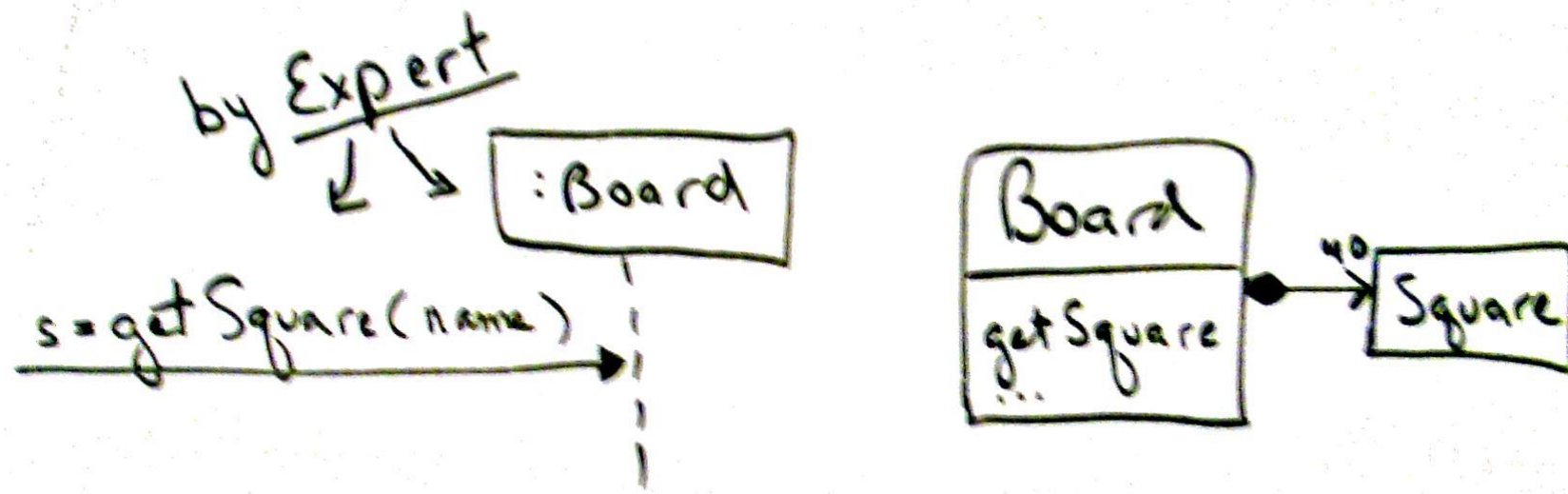


Information Expert Pattern

- suppose objects need to be able to reference a particular Square object, given its name.
 - Who should be responsible for knowing a Square object?
 - this is a 'knowing' responsibility (but also 'doing').
 - e.g. in the Monopoly case study, who knows a Square object?
 - any object can know a Square object, but what would OO developers choose and why?
 - a Board object knows a Square object.
 - to be able to retrieve and present any one Square – given its name – some object must know about all of the Square objects.

Problem: What is a basic principle by which to assign responsibilities to objects?

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

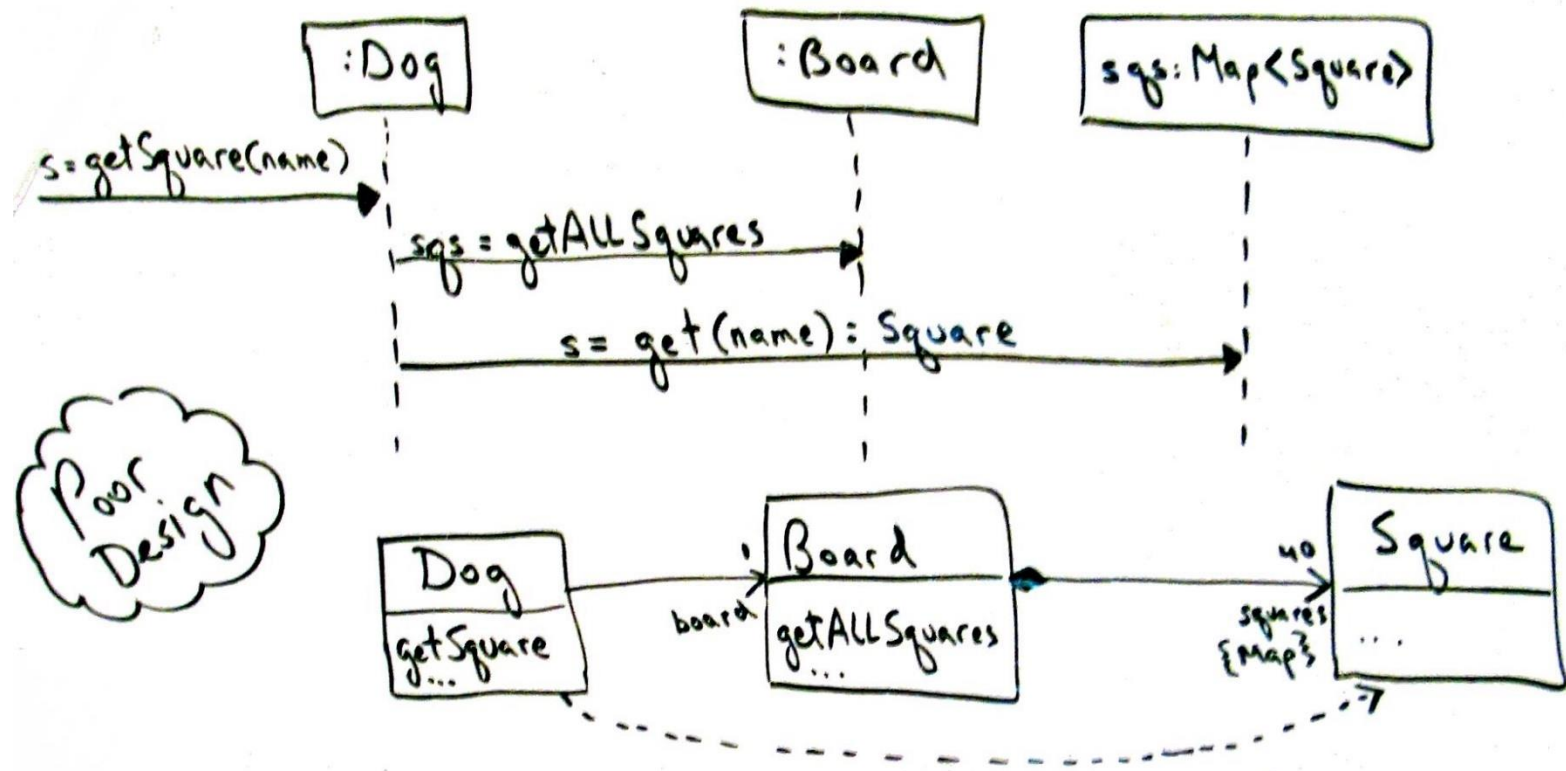


Low Coupling Pattern

- **coupling**: measure of how strongly one element is connected to, has knowledge of, or depends on other elements.
 - if there is coupling (i.e. dependency), then when the depended-upon element changes, the dependant may be affected.
 - e.g. 1. a subclass is strongly coupled to a superclass.
 - e.g. 2. an object O_0 that calls on the operations of object O_1 has coupling to O_1 's services.
- use low coupling to evaluate existing designs or to evaluate the choice between new alternatives.
 - prefer a design whose coupling is lower than the alternatives.
 - because it tends to reduce the time, effort, and defects in modifying software.

Problem: How to reduce the impact of change?

Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.



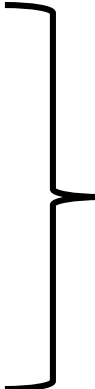
* Higher (more) coupling if Dog has getSquare!

Key Point: Information Expert Pattern Supports Low Coupling

The Information Expert pattern guides us to a choice that supports the Low Coupling pattern. Information Expert asks us to find the object that has most of the information required for the responsibility and assign responsibility there.

If we put the responsibility anywhere else, the overall coupling will be higher because more information or objects must be shared away from their original source.

Controller Pattern

- a simple layered architecture has a UI layer and a domain layer, among others.
 - actors (i.e. users) generate UI events (e.g. button click).
 - objects in the UI layer must react to UI events and cause something to occur.
 - recall that UI objects should not contain domain operations.
 - sometimes the name 'controller' is given to the application logic object that received and controlled (i.e. coordinated) handling the request.
- 
- it follows that once UI objects pick up UI events, they need to delegate the request to domain objects.

Problem: What first object beyond the UI layer receives and coordinates a system operation?

Solution: Assign the responsibility to an object representing the overall “system”, a “root object”, a device that the software is running within, a major subsystem, or a use case scenario within which the system operation occurs.

High Cohesion Pattern

- **cohesion**: measure of how functionally related operations of a software element are and how much work a software element is doing.
 - e.g. an object O_0 with 100 methods and 2,000 lines of code is doing a lot more than an object O_1 with 10 methods and 200 lines of code; if O_0 's 100 methods are covering many different areas of responsibility (e.g. database access, random integer generation, etc.) then it is less focused and less functionally cohesive than O_1 .

Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support the Low Coupling pattern?

Solution: Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

Assigned Readings

- Chapter 17: GRASP: Designing Objects with Responsibilities from Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
- Chapter 18: Object Design Examples with GRASP from Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.