GRASP

Table of Contents

GRASP: Overview and Interrelationship

- General Responsibility Assignment Software Patterns
- **pattern**: named and well-known problem and solution that can be applied to new contexts, providing guidance for assessing trade-offs

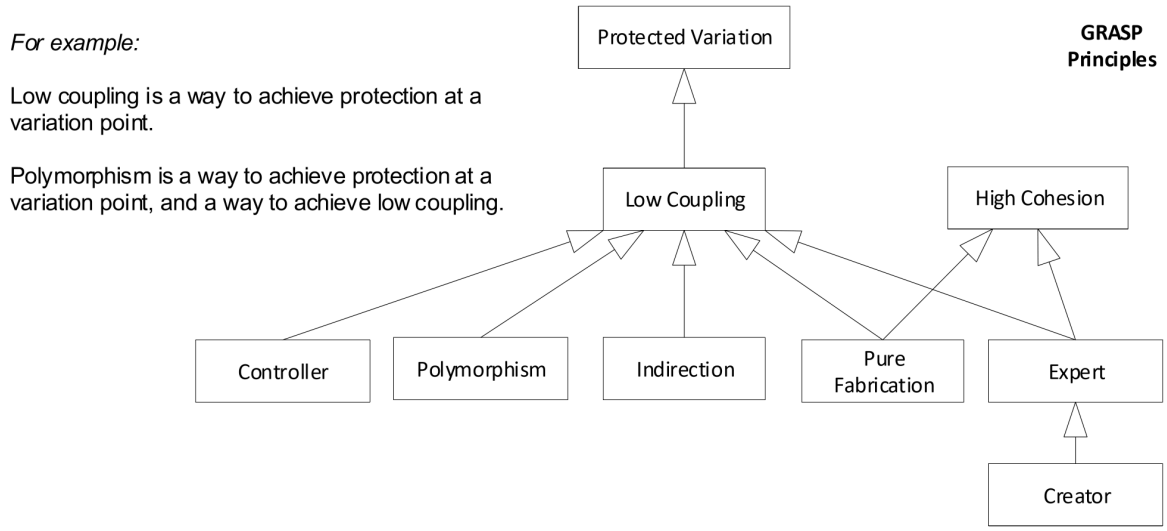

Figure 1: Relationship between grasp principles

Responsibilities

- **responsibility**: contract/obligation
- types of responsibility:
  - *doing*: an object could
    - do something itself, e.g. create an object, perform a calculation
    - initiate action in other objects
    - control/coordinate activities in other objects
  - *knowing*: an object could
    - know about private encapsulated data
    - know about related objects
    - know about things it can derive/calculate
- **low representational gap**: domain model can be used to inspire *knowing* responsibilities
- **granularity**: big responsibilities may take hundreds of classes/methods, while small ones may take a single method
- responsibilities are distinct from methods: responsibilities are an abstraction, but methods fulfill responsibilities
- **collaboration**: responsibiilities may involve multiple objects working together to fulfill a responsibility

- **Responsibility Driven Design:** way to think about assigning responsibilities inOO software design, where the design comprises a community of collaborating responsible objects

Creator

Problem: who should be responsible for object creation?

Solution: Assign class B responsibility to create class A if: - B contains/aggregates A - B records A - B closely uses A - B has initialising data for A

The more of these that hold, the stronger the implication.

Benefits: Low coupling

Contraindications: - complex object creation, e.g. from a family of classes. Instead delegate to Factory

Information Expert

Problem: How to decide which class to assign a responsibility to?

Solution: - Assign X the responsibility if X has the necessary information

Benefits: classes are - understandable - maintainable - extendible

Contraindications: - the solution suggested by Information Expert may introduce problems with coupling and cohesion

Low Coupling

Problem: how to support low dependency, low change impact, and increased reuse?

Solution: - assign responsibilities such that coupling remains low. - use this to differentiate alternatives - **coupling:** degree of connection to other elements (whether knowledge/reliance on)

Benefits: code becomes - maintainable - efficient - reusable

Contraindications: - high coupling can be okay with stable code, e.g. standard libraries

High Cohesion

Problem: How to keep objects focused, understandable, manageable, while suppporting low coupling? - **functional cohesion:** how strongly related and focused the responsibilities of an element

are - **low cohesion:** class performs too many unrelated tasks. Code is hard to comprehend, reuse, maintain

Solution: - choose between alternatives by assigning the reponsibility to X for maximum cohesion

Benefits: code becomes - easy to comprehend - maintainable - reusable

Contraindications: - non-functional requirements may require low cohesion, e.g. reduce processing overheads in high- performance computing

Controller

Problem: What first object beyond the UI layer receives and coordinates (i.e. controls) system operation?

Solution: - **facade controller:** assign responsibility to a class representing the overall system - **use case/session controller:** assign responsibility to a class representing a use case scenario that deals with the event, named something like <`UseCaseName`><`Handler`|`Coordinator`|`Session`>

Benefits: prevent coupling between UI and application logic

Issues: - **bloated controller**: controllers with too many responsibilities (low cohesion) - break facade controller into multiple use case controllers - delegate work to other objects: only handle control in the controller itself

Polymorphism

Problem: - how to handle alternatives based on *type* (class)? - conditional variation using **`switch`**–**`case`** statements requires heavy modification when new alternatives are added - how to create pluggable software components? - viewing components in a client-server relationship, how can you replace a server component without affecting the client?

Solution: when related alternatives/behaviours vary by type (class), assign responibility for the behaviour using polymorphic operations to the types (classes) for which the behaviour varies. - i.e. give the same name to services in different objects - i.e. inheritance with generalisation/specialisation, or interfaces

Corollary: avoid testing the type of an object as part of conditional logic to perform varying alternatives based on type (class).

Guideline: unless there is a default behaviour in the superclass, declare a polymorphic operation in the superclass to be **`abstract`**.

Guideline: when should you consider using an interface?
- introduce one when you want to support polymorphism without being committed to a class hierarchy

Benefits: - easy extension of code: you can introduce new implementations without affecting clients

Contraindications: - avoid premature optimisation: consider future proofing with respect to realistic likelihood of variability before investing time in increased flexibility.

Pure Fabrication

Problem: what object should have a responsibility, where you don't want to violate high cohesion/low coupling, etc., but guidance from Expert etc. is not appropriate?

Solution: assign a highly cohesive set of responsibilities to an artificial/convenience class that doesn't exist in the problem domain.

Benefits: - high cohesion - reuse potential

Contraindications: - overuse where each class is basically a single function: produces high coupling and lots of message passing

Indirection

Problem: - where to assign responsibility to avoid direct coupling between 2+ things? - how to decouple to support low coupling and reuse potential?

Solution: - assign responsibility to an intermediary, creating indirection between components - e.g. `Adapter` to protect inner design against external variation - "Most problems in computer science can be solved by another layer of indirection"

Benefits: - reduced coupling

Contraindications: - high performance may need to reduce amount of indirection - "Most problems in performance can be solved by removing another layer of indirection"

Protected Variations

Problem: How to design objects/systems so that variation in these elements doesn't impact other elements?

Solution: - identify points of predicted variation/instability - assign responsibilities to create a stable interface (in the broad sense of an access view) around them - points of change: - **variation point:** variation in existing system/requirements - **evolution point:** speculative variations that may arise in the future - equivalent to **Open-Closed principle**: objects should be open for extension, and closed to modification that affects clients - equivalent to **Information Hiding**

Benefits: - extensible - new implementations don't affect clients - low coupling - low cost of change

Contraindications: - cost of future-proofing can outweigh benefits - reworking a brittle design as needed may be easier

Guidance: - novice developers produce brittle designs - intermediate developers produce overly fancy/flexible, generalised designs that never get used - expert developers choose with insight, balancing the cost of changing a simple/brittle design against its likelihood

Liskov Substitution Principle

- software (methods, classes, …) referring to a type `T` (interface, abstract superclass) should work properly with any substituted implementation or subclass of `T`

Don't talk to strangers

Avoid creating designs that traverse long object structure paths and/or send messages to distant, indirect (stranger) objects. doing so makes designs fragile with respect to changes in object structures.

Within a method, messages should only be passed to:

1. **`this`** object (`self`)
2. parameter of the method
3. attribute of **`this`**, (or element of collection that is an attribute of **`this`**)
4. object created in the method

Intent:: avoid coupling between client and knowledge of indirect objects, and connections between objects.

```
public void doX() {
    // avoid this
    F someF = foo.getA().getB().getC().getD().getE().getF();
    // this is better
    F someF = foo.getFfromFoo();
}
```

Guideline: The farther along a path one traverses, the more fragile it will be. Instead add a public operation to direct objects that hides how the information is obtained.