L17 - Introduction to detailed design patterns (1)

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

17.1 Reminding past issues and mapping them to current topics

Where are we now?

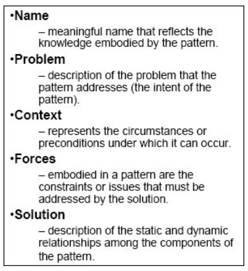
⇒ · · · · · ·

- ⇒ Elaboration (second UP phase)
  - ⇒ · · · · · ·
  - ⇒ Elaboration Design
    - ⇒ Architectural design & patterns
    - ⇒ Detailed design
      - ⇒ Detailed design patterns
        - ⇒ Catalogue of design patterns: GRASP patterns

⇒ · · · · · ·

Detailed design patterns - a gentle reminder

- Patterns are three-part rules that express a relation between:
  - a context
  - a *system of forces* which occurs repeatedly in that context
  - a *software configuration* allowing these forces to resolve themselves
- Templates are used to document patterns
- Different template styles emphasise different pattern aspects
- No consensus view on most appropriate template!
- There is agreement on a minimal template:



- **Name**
  - – meaningful name that reflects the knowledge embodied by the pattern.
- **Problem**
  - – description of the problem that the pattern addresses (the intent of the pattern).
- **Context**
  - – represents the circumstances or preconditions under which it can occur.
- **Forces**
  - – embodied in a pattern are the constraints or issues that must be addressed by the solution.
- **Solution**
  - – description of the static and dynamic relationships among the components of the pattern.

17.2 The creator pattern

GRASP pattern No 1: *creator* (spec)

Name: Creator

Problem: Who should be responsible for creating a new instance of a class `A`?

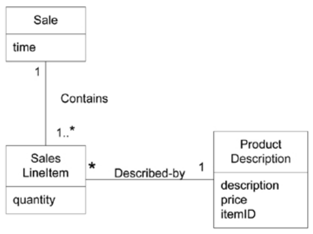
Solution: Assign class `B` the responsibility to create an instance of class `A` if one of these is true (the more the better) -

- `B` ‘contains’ or compositely aggregates `A`.
- `B` records `A`.
- `B` closely uses `A`.
- `B` has the initialising data for `A` that will be passed to `A` when it is created. Thus `B` is an *Expert* with respect to creating `A`.

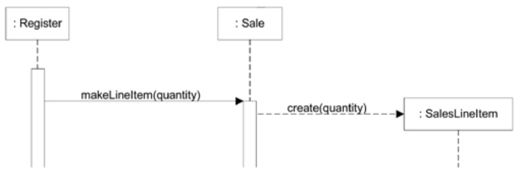
`B` is a *creator* of `A` objects.

Note: This pattern focuses on responsibility assignment

GRASP pattern No 1: *creator* (example)



Who creates a new SalesLineItem during a Sale? (Product Description already exists)

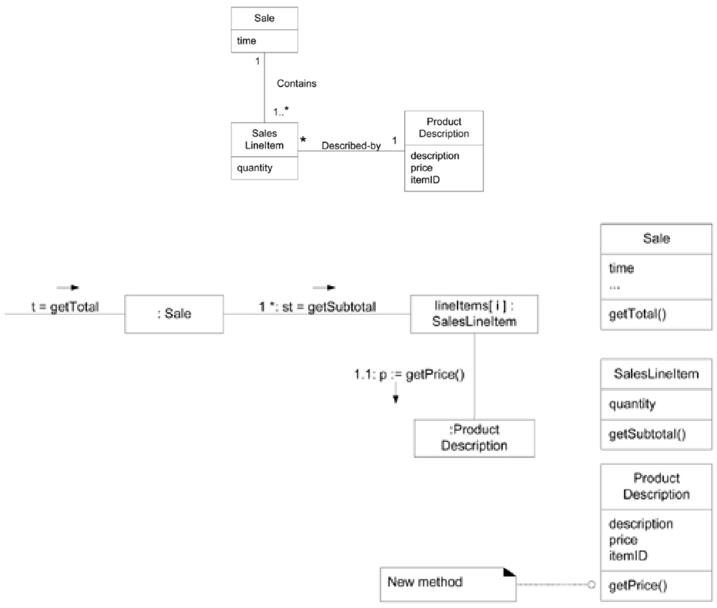


17.3 The expert pattern

GRASP pattern No 2: *expert* (spec)

Name:	[Information] Expert
Problem:	What is a general principle of assigning responsibilities to objects?
Solution:	Assign a responsibility to the expert (*i.e.*, the class that has the information necessary to fulfill the responsibility).
Note:	An expert is also an expert in creations, so to some extent this second pattern can be considered as a subtype of the creator pattern

GRASP pattern No 2: *expert* (example)

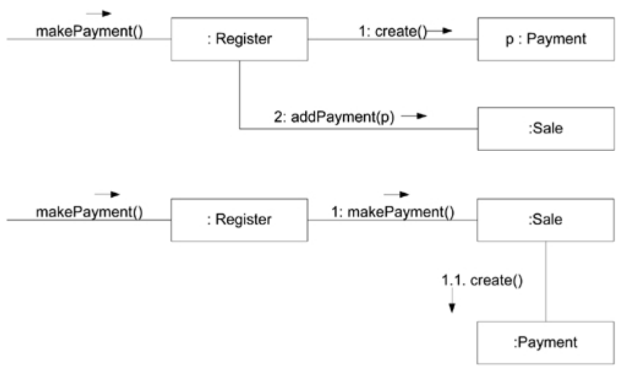


17.4 The low coupling pattern

GRASP pattern No 3: *low coupling* (spec)

Name:	Low coupling
Problem:	How to support low dependency, low change impact, and increased reuse?
Solution:	Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.
Note:	Low coupling is an evaluative principle that you apply while evaluating all design decisions. However, high coupling to stable elements and to pervasive elements is seldom a problem.

GRASP pattern No 3: *low coupling* (example)



17.5 The controller pattern

GRASP pattern No 4: *controller* (spec)

Name: Controller

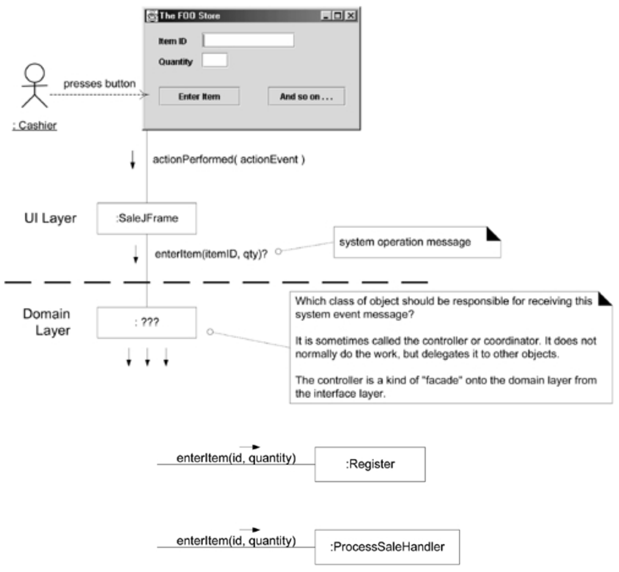
Problem: What first object beyond the User Interface layer receives and coordinates (namely, *controls*) a system operation?

Solution: Assign the responsibility to one of the following:

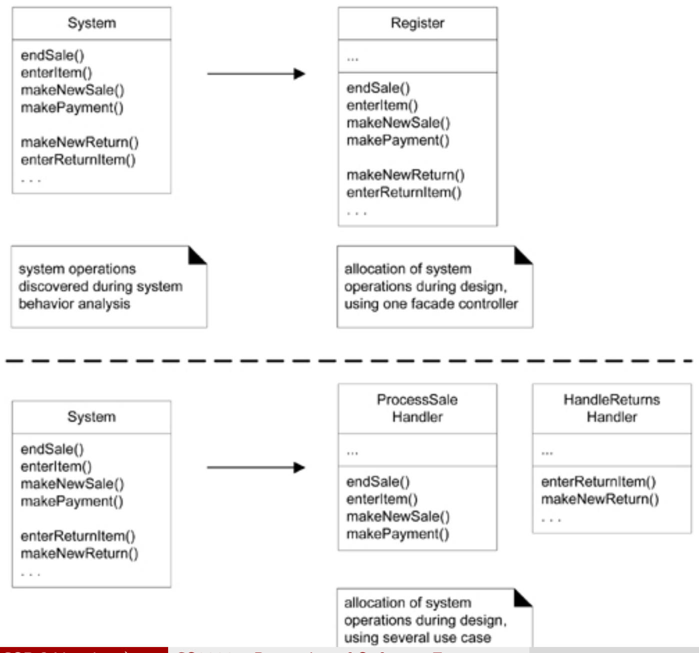
- A class representing the overall system, a root object, a device the software is running within, a major subsystem. These are all variations of a *GoF facade* controller
- A class that represents a use case scenario within which the system event occurs

Notes: Use the same controller class for all system events in the same use case scenario

GRASP pattern No 4: *controller* (example)


The FOO Store
Item ID
Quantity
Enter Item
And so on ...
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.
enterItem(id, quantity)
:Register
enterItem(id, quantity)
:ProcessSaleHandler

GRASP pattern No 4: *controller* (operation allocations)



17.6 The high cohesion pattern

GRASP pattern No 5: *high cohesion* (spec)

Name: High cohesion

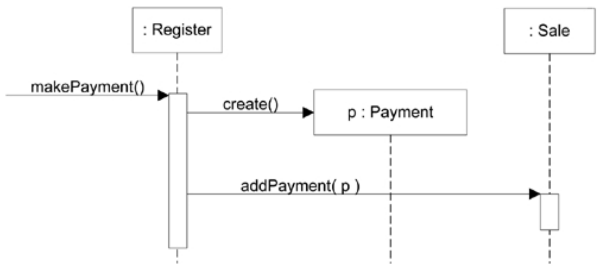
Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support *low coupling*?

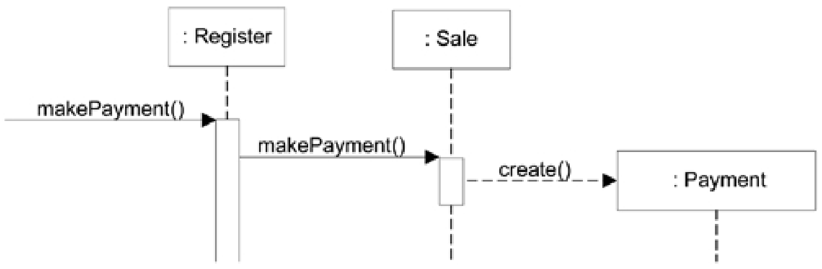
Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

Notes: In a few cases, accepting lower cohesion is justified:

- Grouping of responsibilities or code into one class or component to simplify maintenance by one person
- Distributed server objects. Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations

GRASP pattern No 5: *high cohesion* (example)



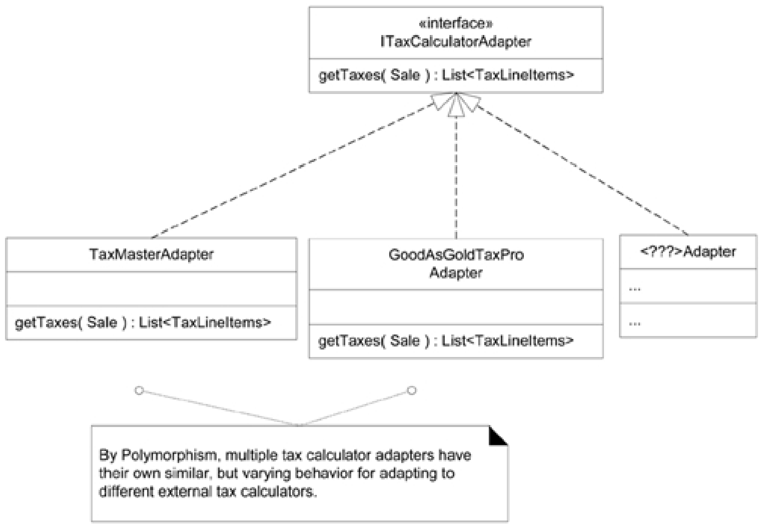


17.7 The polymorphism pattern

GRASP pattern No 6: *polymorphism* (spec)

Name:	Polymorphism
Problem:	How handle alternatives based on type? How to create pluggable software components?
Solution:	When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour — using polymorphic operations — to the types for which the behaviour varies
Notes:	Sometimes systems are designed with interfaces and polymorphism for speculative 'future-proofing' against an unknown possible variation. Unless this is very probable, it should be avoided

GRASP pattern No 6: *polymorphism* (example)


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List<TaxLineItems>
TaxMasterAdapter
getTaxes(Sale) : List<TaxLineItems>
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List<TaxLineItems>
<???>Adapter
...
...
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

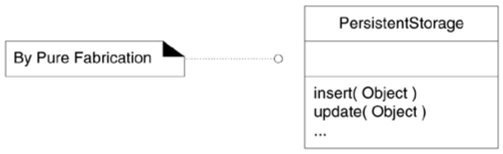
17.8 The pure fabrication pattern

GRASP pattern No 7: *pure fabrication* (spec)

Name:	Pure Fabrication
Problem:	What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
Solution:	Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept — but something made up — in order to support high cohesion, low coupling, and reuse. Such a class is a fabrication of the imagination.
Notes:	If overused, pure fabrication could lead to too many behaviour objects whose responsibilities are not co-located with the information required for their fulfillment, which can adversely affect coupling. The usual symptom is that most data inside the objects is being passed to other objects to reason with it.

GRASP pattern No 7: *pure fabrication* (example)

- You need to save `Sale` instances in a relational DB
- This includes supporting a range of DB-oriented operations, none of which is strictly related to a `Sale`, which would thus become *incohesive*
- The `Sale` class has to be coupled to the relational DB interface (*e.g.*, through JDBC), so its coupling goes up
- Saving objects in a relational DB is a very general task for which many classes need support. Placing these responsibilities in `Sale` implies poor reuse or lots of duplication in other classes that do the same thing.
- Create a new class solely responsible for saving objects in the relational DB (namely, `PersistentStorage`, which is a *pure fabrication*)

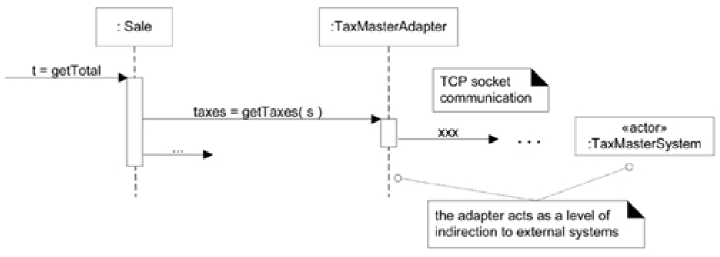


17.9 The low indirection pattern

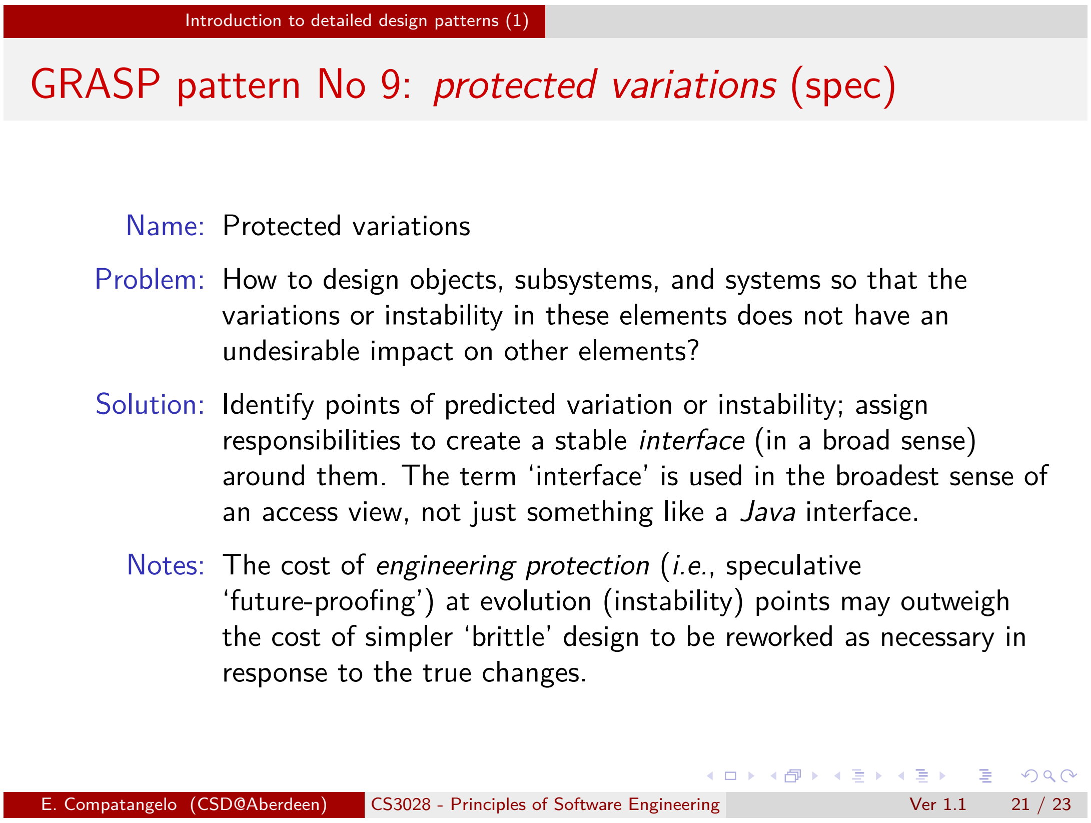
GRASP pattern No 8: *indirection* (spec)

Name:	Indirection
Problem:	Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to decouple objects so that low coupling is supported and reuse potential remains higher?
Solution:	Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an *indirection* between the other components.
Notes:	Sometimes too many levels of indirection can be created, which decrease design simplicity and understanding as well as system performance

GRASP pattern No 8: *indirection* (example)



17.10 The protected variations pattern

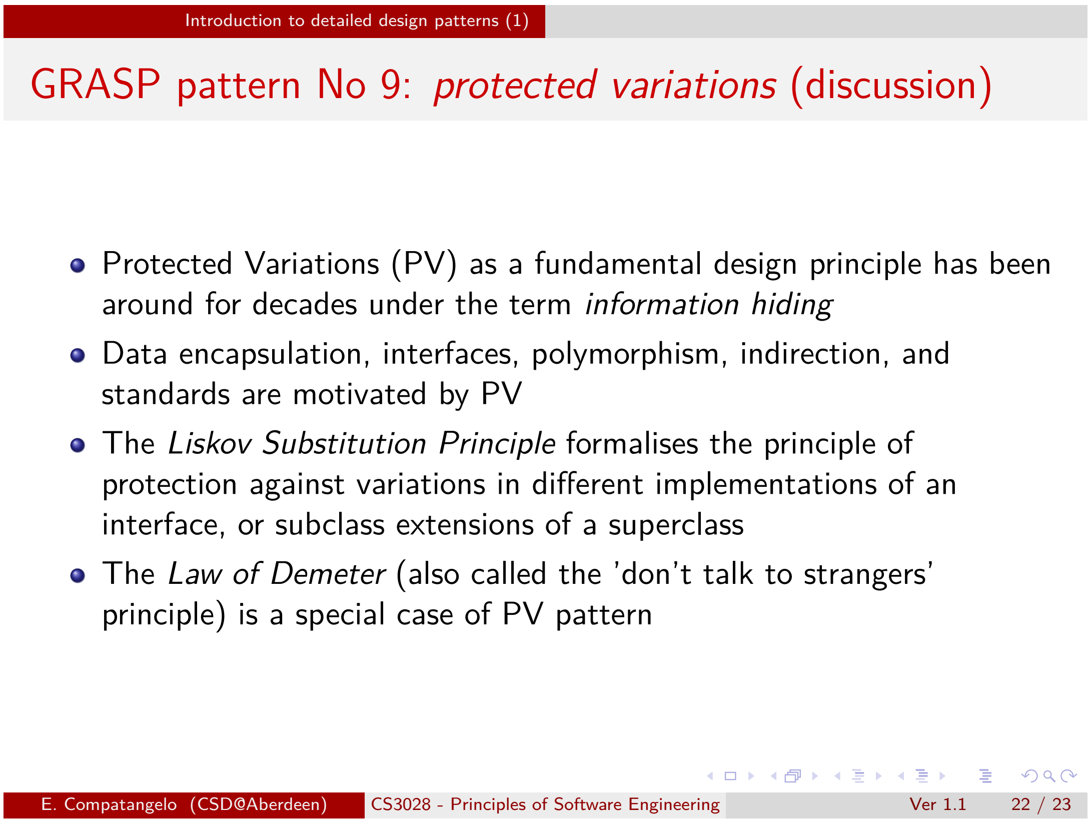


Introduction to detailed design patterns (1)

GRASP pattern No 9: *protected variations* (spec)

Name:	Protected variations
Problem:	How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution:	Identify points of predicted variation or instability; assign responsibilities to create a stable *interface* (in a broad sense) around them. The term 'interface' is used in the broadest sense of an access view, not just something like a *Java* interface.
Notes:	The cost of *engineering protection* (*i.e.*, speculative 'future-proofing') at evolution (instability) points may outweigh the cost of simpler 'brittle' design to be reworked as necessary in response to the true changes.

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 21 / 23



Introduction to detailed design patterns (1)

GRASP pattern No 9: *protected variations* (discussion)

- Protected Variations (PV) as a fundamental design principle has been around for decades under the term *information hiding*
- Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV
- The *Liskov Substitution Principle* formalises the principle of protection against variations in different implementations of an interface, or subclass extensions of a superclass
- The *Law of Demeter* (also called the 'don't talk to strangers' principle) is a special case of PV pattern

E. Compatangelo (CSD@Aberdeen) CS3028 - Principles of Software Engineering Ver 1.1 22 / 23

Next lecture. . .

Selected Gang of Four patterns

More specifically, we will focus on:

- The *Adapter* pattern
- The *Factory* pattern
- The *Singleton* pattern
- The *Strategy* pattern
- The *Composite* pattern
- The *Facade* pattern
- The *Observer* pattern