L18 - Introduction to detailed design patterns (2)

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

18.1 Reminding past issues and mapping them to current topics

Where are we now?

⇒ · · · · · ·

⇒ Elaboration (second UP phase)

⇒ · · · · · ·

⇒ Elaboration Design

⇒ Architectural design & patterns

⇒ Detailed design

⇒ Catalogue of design patterns: GoF patterns

⇒ Catalogue of design patterns: GoF patterns (selection)

⇒ · · · · · ·

18.2 The adapter pattern

GoF pattern No 1: *adapter* (spec)

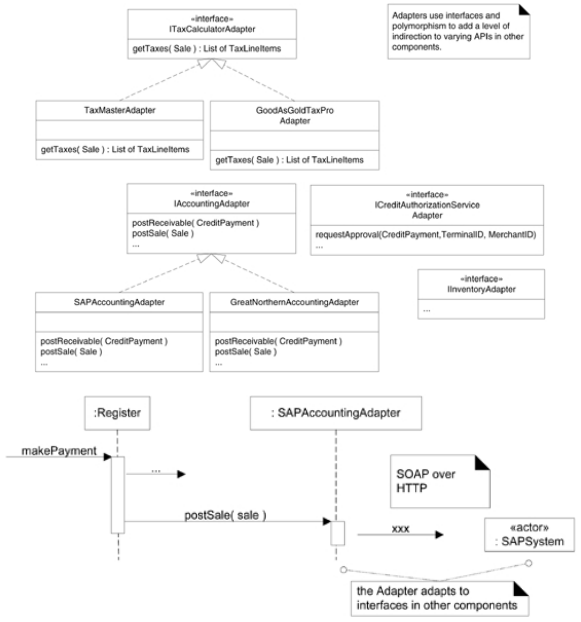
Name: Adapter

Problem: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution: Convert the original interface of a component into another interface, through an intermediate adapter object.

Notes: Adapters use *interface* and *polymorphism* to add a level of indirection; the same warning about 'future-proofing' thus holds here.

GoF pattern No 1: *adapter* (example)



18.3 The factory pattern

GoF pattern No 2: *factory* (spec)

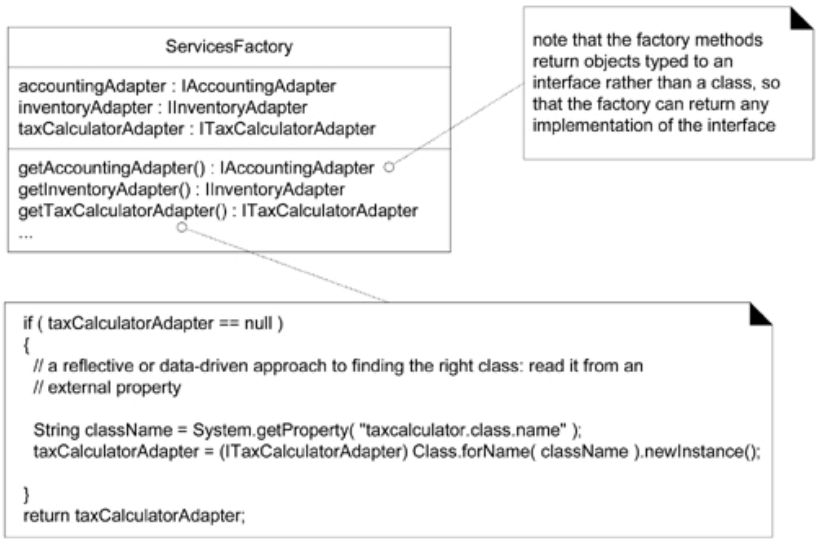
Name: Factory

Problem: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution: Create a *pure fabrication* object called a *factory* that handles the creation.

Notes: Factories use *pure fabrication*; so the same warning against the creation of behaviour objects *whose responsibilities are not co-located with the information required for their fulfillment* thus holds here.

GoF pattern No 2: *factory* (example)



18.4 The singleton pattern

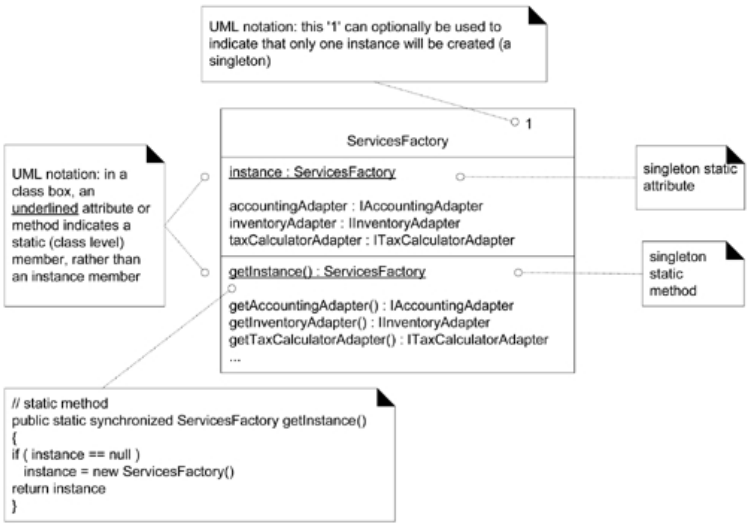
GoF pattern No 3: *singleton* (spec)

Name: Singleton

Problem: How can a class be constructed having only one, globally accessible instance (a *singleton*)? Some applications require a class to have one instance only. However, making object a global variable is not a good design choice. Using static operations and attributes limits extensibility.

Solution: Define a static method of the class that returns the singleton

GoF pattern No 3: *singleton* (example)



18.5 The strategy pattern

GoF pattern No 4: *strategy* (spec)

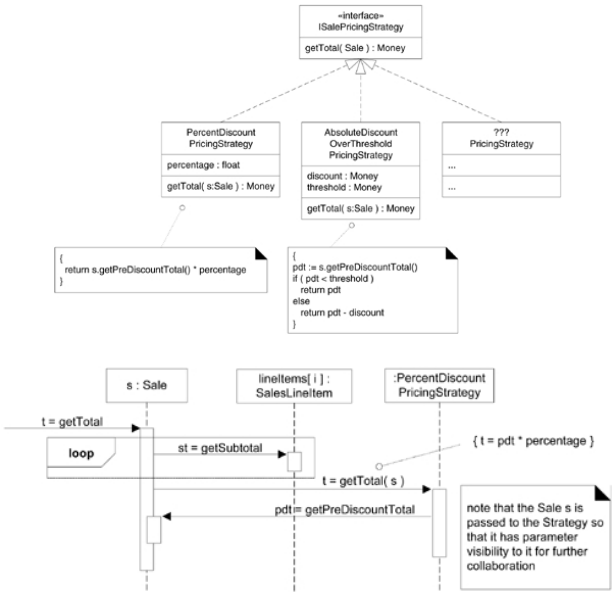
Name: Strategy

Problem: How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

Solution: Define each algorithm / policy / strategy in a separate class, with a common interface.

Notes: Strategy is based on *polymorphism*, and provides *protected variations* with respect to changing algorithms. Strategies are often created by a *factory*. All the corresponding warnings thus apply.

GoF pattern No 4: *strategy* (example)



18.6 The composite pattern

GoF pattern No 5: *composite* (spec)

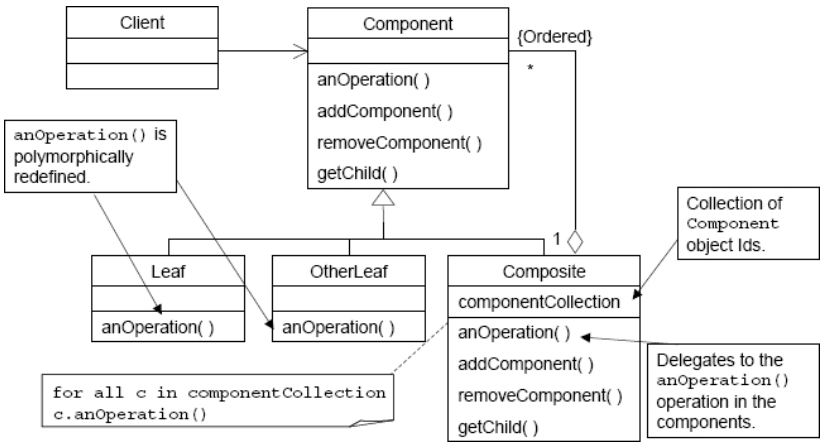
Name: Composite

Problem: How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object? This is a requirement to represent whole-part hierarchies; whole/part (composite and component) objects should offer same interface & behaviour.

Solution: Combine inheritance and aggregation hierarchies.

Notes: Shared interface implies same inheritance hierarchy; part-whole hierarchy indicates aggregation structure. Composite is based on *polymorphism* and provides *protected variations* to a client so that it is not impacted if its related objects are atomic or composite.

GoF pattern No 5: *composite* (example)



18.7 The facade pattern

GoF pattern No 6: *facade* (spec)

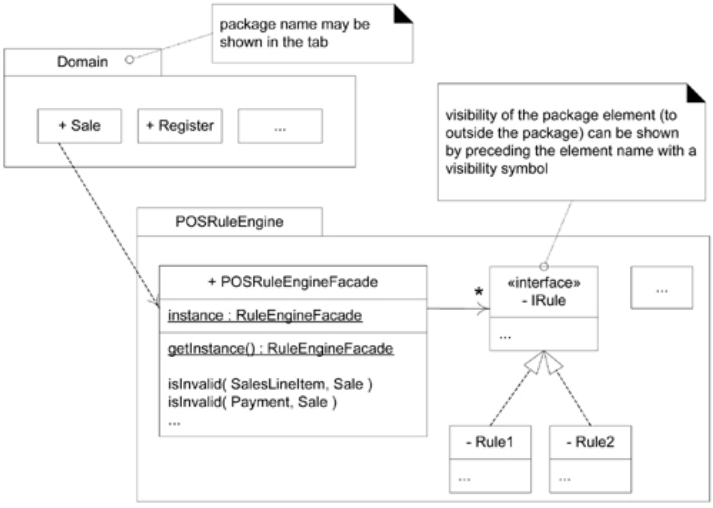
Name: Facade

Problem: A common, unified interface to a disparate set of implementations or interfaces — such as within a subsystem — is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?

Solution: Define a single point of contact to the subsystem, *i.e.*, a *facade* object that wraps the subsystem. This object presents a single unified interface and is responsible for collaborating with the subsystem components.

Notes: Subsystem hidden by the facade object could contain dozens or hundreds of classes of objects, or even a non-object-oriented solution, yet as a client to the subsystem, we see only its one public access point.

GoF pattern No 6: *facade* (example)



18.8 The observer (publish - subscribe) pattern

GoF pattern No 7: *observer [publish - subscribe]* (spec)

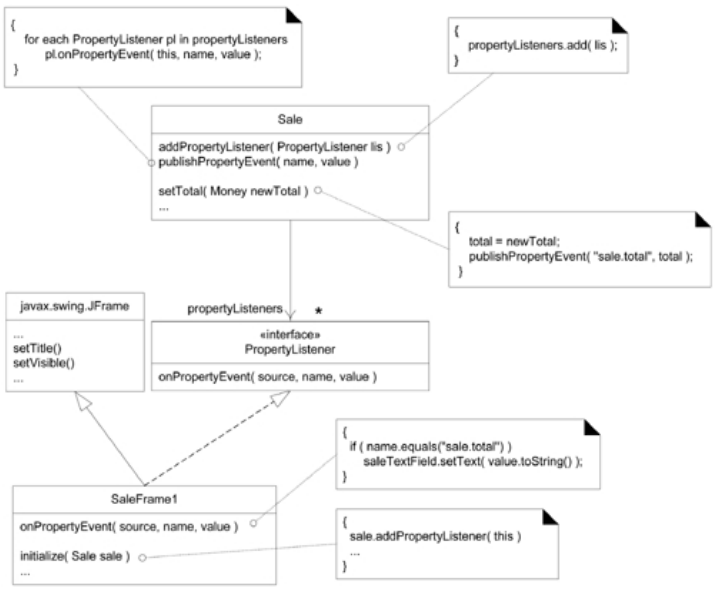
Name: Observer (publish - subscribe)

Problem: Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?

Solution: Define a 'subscriber' or 'listener'interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

Notes: Also called the *Delegation Event Model* in Java because the publisher delegates handling of events to 'listeners' (subscribers).

GoF pattern No 7: *observer [publish - subscribe]* (example)



Next week...

From design to implementation

More specifically, we will focus on:

- Construction; Software Quality Assurance; standards
- Testing strategies
- Black-box and white-box testing