Chapter 17-18-25-26
Responsibilities + GRASP + GOF

A popular way of thinking about the design of SW objects and also larger-scale components is in terms of responsibilities, roles and collaborations. This is part of a larger approach called responsibility-driven design **RDD**.

Responsibility→ In RDD, we think of SW objects as having responsibilities, an abstraction of what they do.
UML define responsibility as→ a contract or obligation of a classifier. It is related to the behavior of an obj. in terms of its role. Responsibilities are one of two types: doing & knowing.

- **Doing responsibilities include**
  - ✓ Doing sth itself such as creating an obj. or doing a calculation
  - ✓ Initiating action in other obj.
  - ✓ Controlling & coordinating activities in other obj.
- **Knowing responsibilities include**
  - ✓ Knowing about private encapsulated data
  - ✓ Knowing about related obj.
  - ✓ Knowing about things it can derive or calculate

RDD→ assign responsibilities to classes of obj. during obj. design. Example, Sale is responsible for creating SaleLineItems (a doing), Sale is responsible for knowing its total (a knowing)

Low-representational gap→ Domain obj. in the domain model inspires relevant responsibilities related to "knowing" because of the attributes and associations that they have. Example, Sale obj. in the DM has the attribute "time", it's nature by low-representational gap that a sw Sale class knows its time.

Granularity of responsibility→ the translation of responsibilities into classes and methods is influenced by granularity of responsibility. Three types:

- Small scale: a single object fulfills the responsibility
- Medium scale: a group of objects collaborate
- Large scale: an entire system or sub-system required

> A responsibility is direct metaphor with human workers in an organization. It is not the same thing as method, it is an abstraction, but methods fulfill responsibilities.

Collaboration→ RDD includes the idea of collaboration. Responsibilities are implemented as methods that either act as alone or collaborate with other methods and objects. Example, Sale has res. to know its total, so has a method getTotal to fulfill this res. Sale collaborates with other obj. sending getSubTotal message to each SalesLineItem asking for its subtotal.

GRASP→General Responsibility Assignment Software Patterns (or Principles)→

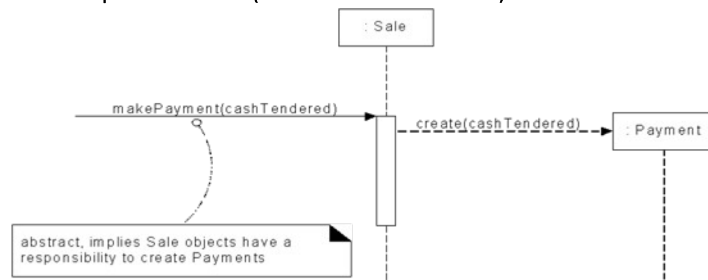
- Fundamental techniques for effective RDD
- Must understand GRASP to do good OOD

GRASP is a learning aid to grasp OO design with responsibilities, to help in understanding essential object design and applying design reasoning in a methodical, rational and explainable way. 9 GRASP Patterns:

1 Creator
2 Information Expert
3 Low Coupling
4 Controller
5 High Cohesion
6 Polymorphism
7 Indirection
8 Pure Fabrication
9 Protected Variations

What's the connection between responsibilities, GRASP and UML diagrams?

We think about assigning responsibilities to obj. while coding or while modeling. Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods).



Someone sends Sale a message telling it to make a payment
Implies Sale has the responsibility to create a Payment object
Doesn't necessarily mean Sales creates Payment directly

Therefore, when we draw a UML interaction diagram (SD or CD), we are deciding on responsibility assignment.

What are patterns and how patterns are related to SW design?

In OO design, a pattern is a named description of a problem and its solution that can be applied to new context; ideally, a pattern advices us on how to apply its solution in varying circumstances and considers the forces, trade-offs, implementations, variations and so on.
Patterns effectiveness confirmed by prior experience and they eliminate "reinvention of the wheel".
Naming a pattern, design idea or principle has the following advantages:

- support chunking and incorporating that concept into our understanding and memory.
- Facilitate communication.

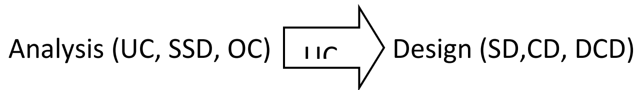
UCR→ describes how a particular UC is realized within the design model, in terms of collaborating objects.
More precisely, a designer can describe the design of one or more scenarios of a UC; each of these is called a UC realization (or scenario realization).
UCR is a UP term used to remind us of the connection btw the requirements expressed as UCs and the object design that satisfies the requirements (Traces reqs into OO model).

UML diagrams are a common language to illustrate UCR. We can apply principles and patterns of object design, such as information Expert and Low coupling during UCR design work.

UCR & other artifacts→

[1] The UC suggests the system **operations** that are shown in **SSD**.
[2] The system operations become the starting messages entering the **Controllers** for **domain layer** interaction diagrams.
[3] Domain layer interaction diagrams illustrate how objects interact to fulfill the required tasks→the UCR

Thus, Sequence diagrams model collaboration detail. As same in SSD, if use communication diagrams to illustrate UCRs, we will draw a different communication diagram to show the handling of each system operation message.

Analysis (UC, SSD, OC) 

Design (SD,CD, DCD)

Although Startup UC is first to be invoked
It should be modeled last
You need to know what 'root' objects need to be created
Therefore, Process Sale is modeled prior to Startup System
See example 325-334

GRASP Patterns

1. Creator

Problem	Who should create object A?
Solution (advice)	Assign class B the responsibility to create an instance of class A if one of these is true: ✓ B contains or compositely aggregates A ✓ B records A ✓ B closely uses A ✓ B has the initializing data for A

Note that we turned to the concept of **composition** in considering the creator Pattern. A composite obj. is an excellent candidate to make its parts.

Creator benefits

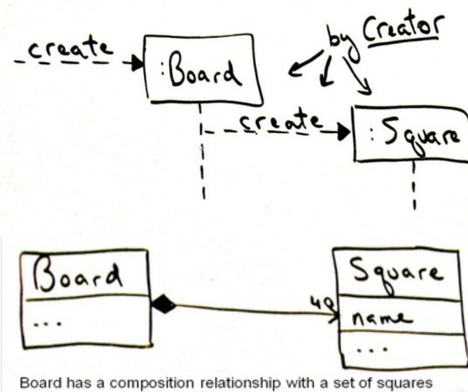
- Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
- Coupling is properly not increased because the *created class* is likely already visible to the *creator* class due to the existing associations that motivated its choice as creator.

Related Patterns

- Low coupling
- Concrete Factory & Abstract Factory

1.1 Monopoly example

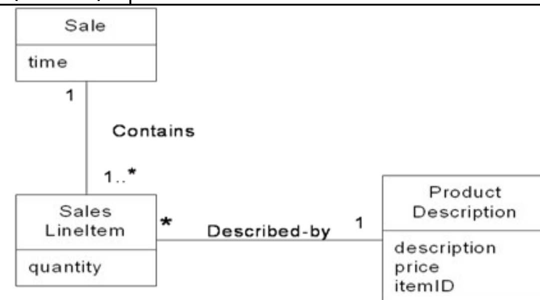
Problem	When you start a game, who creates the squares for the board?
Solution (advice)	Let Board create them since it *contains* the squares

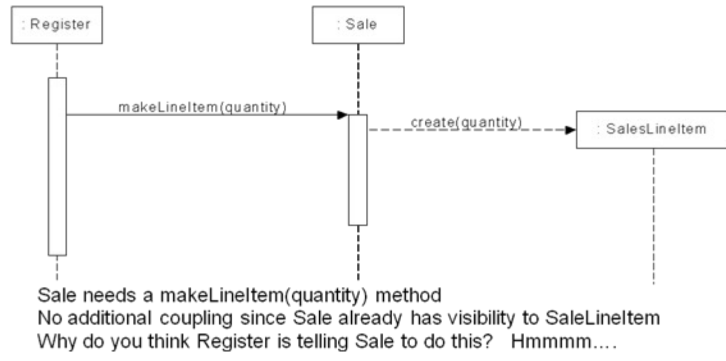


Board has a composition relationship with a set of squares

1.2 NextGen POS Example

Problem	Who should create a SalesLineItem?
Solution (advice)	Let Sale do it because Sale contains SaleLineItems





2. Expert

Problem	What is a basic principle of RDD by which to assign responsibilities to objects?
Solution (advice	Assign responsibility to the object (class) that has the required information to fulfill it. "Tell the expert to do it!"

Expert benefits

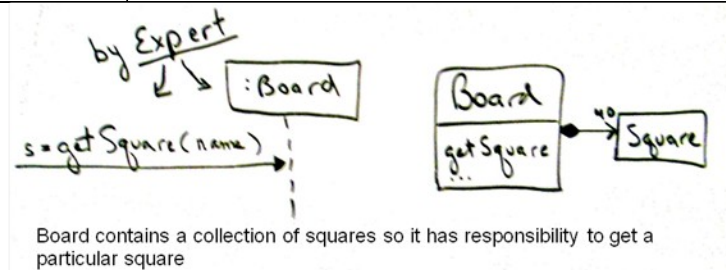
- Information encapsulation is maintained since obj. use their own info to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems.
- Behavior is distributed across the classes that have the required info, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain.

Related Patterns

- Low coupling
- High cohesion

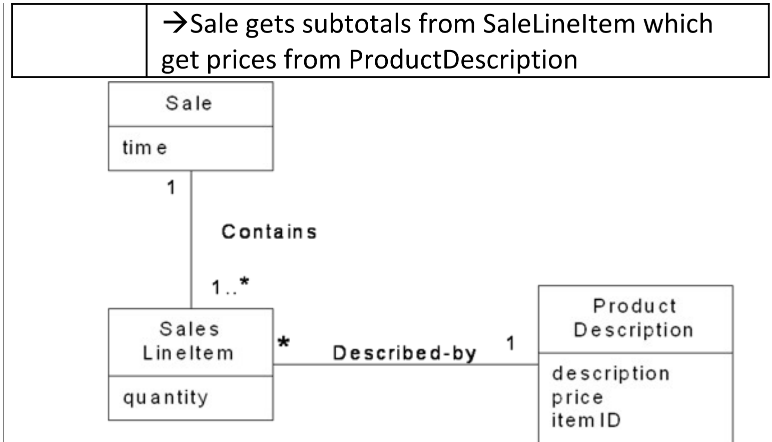
2.1 Monopoly example

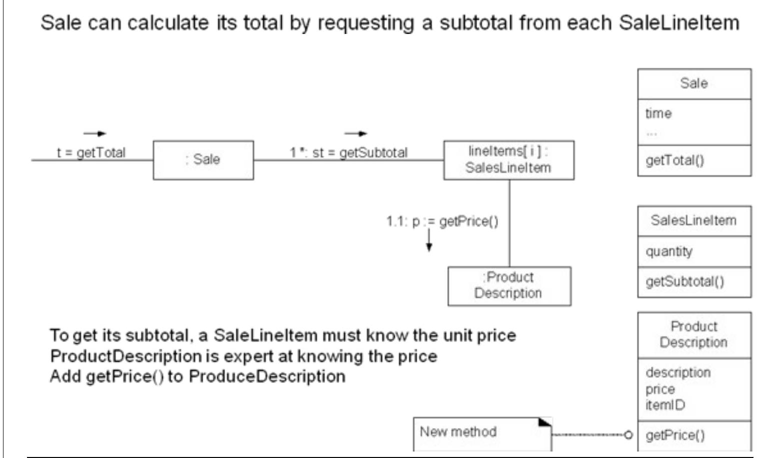
Problem	Who should get a square given a unique ID?
Solution (advice	Let the Board do it because it knows about the squares



2.2 NextGen POS Example

Problem	Who should be responsible for knowing the total of a sale?
Solution (advice)	Let Sale do it because it knows all of its SaleLineItems. For example, to get the sale total
	→Sale gets subtotals from SaleLineItem which get prices from ProductDescription





3. Low Coupling

Coupling→ is a measure of how strongly one element is connected to, has knowledge of, or depends on other elements. If there is coupling or dependency, then when the depended-upon element changes, the dependent may be affected.

Problem	How to reduce the impact of change?
Solution (advice	✓ Assign responsibility so that (unnecessary) coupling between objects remain low. ✓ Evaluate design alternatives ✓ Choose option that minimizes coupling ✓ "Simple chain of command"

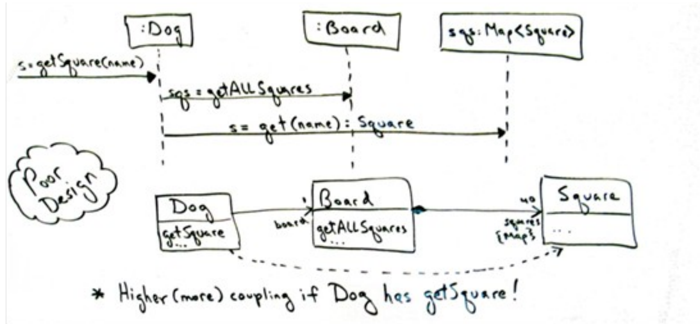
Low Coupling benefits

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse
- **Classes are**
  - More independent
  - Easier to reuse
  - Easier to understand
  - Easier to maintain

Related Patterns

- Protected Variation

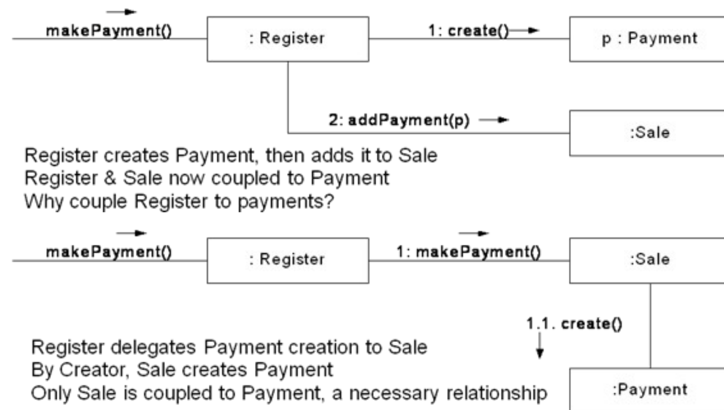
3.1 Monopoly example



If you let Dog get the Square, then Dog & Board are coupled to Square!

3.2 NextGen POS Example

Problem	Who should create a Payment and associate it with a Sale?
Solution (advice)	By **Expert**, let Register create Payment info, then associate it with the Sale ✓ Register & Sale now coupled to Payment By **Low Coupling**, Register delegates Payment creation to Sale ✓ Only Sale coupled to Payment



Common coupling (types X & Y)→

[1] X has an attribute of Y
[2] X has a method that references Y
  a. Method parameter of type Y
  b. Local variable of type Y
  c. Method call return of type Y
[3] X is a direct/indirect subclass of Y
[4] X implements a Y interface

---

4. Controller

Controller→ is the first obj. (class) beyond the UI layer that is responsible for receiving or handling a system operation message.

Problem	The UI & domain layers should be loosely coupled. What is the 1st object should receive and coordinate messages (system operation) between the UI layer and other domain objects?
Solution (advice)	Assign responsibility to an object representing one of these choice: ✓ The overall 'system' or a 'root' object (Store, Bank) ✓ A device that the SW is running within (BankATM) ✓ A major subsystem (AccountingSystem) ✓ A major use case scenario (GameHandler) ✓ 'Single channel of communication between layers'

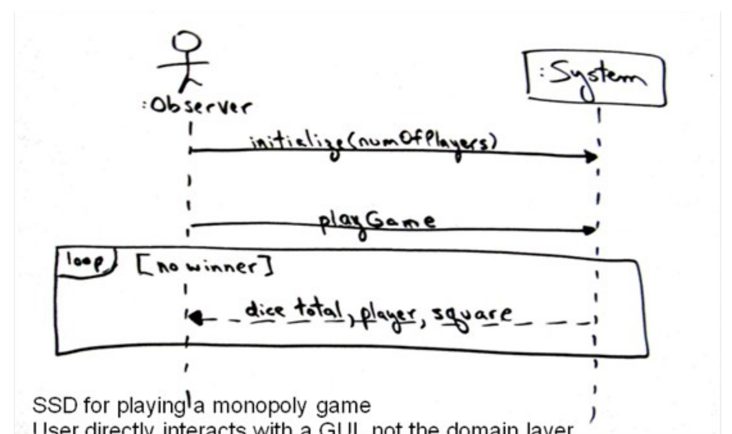
Controller benefits

- Increased potential for reuse and pluggable interfaces
- Opportunity to reason about the state of the UC

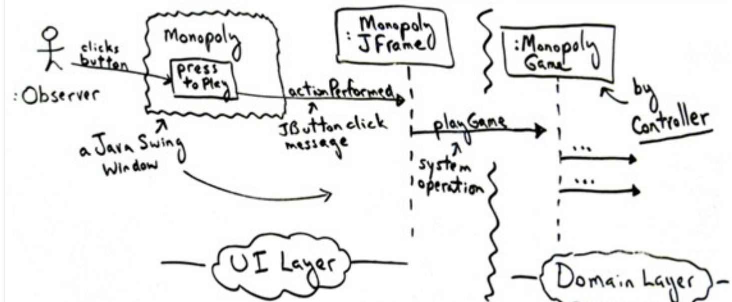
Related Patterns

- Command: In a msg handling sys., each msg may be represented and handled by a spate Command obj
- Façade
- Layers (POSA pattern) placing domain logic in the domain layer.
- Pure Fabrication

4.1 Monopoly example



Which object should relay system operations from UI to domain layer?

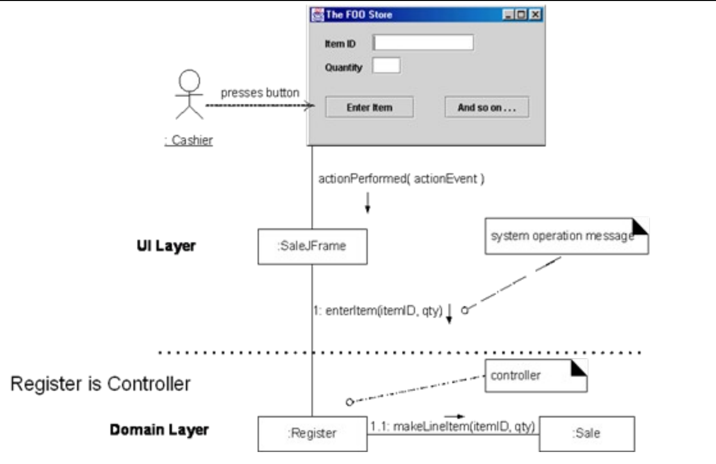


Let MonopolyGame be controller
It represents the system & there aren't many system operations

4.2 NextGen POS Example

Problem	Which object in the domain layer should be

	responsible for the system operations?
Solution (advice)	✓ Select the 'root' object in the POS domain model (Store) ✓ Select an object that represents a key device (Register) ✓ Create an object that handles the use case (SaleHandler)



5. High Cohesion

Cohesion→ measures how functionality related the operations of a SW elements are, and also measures how much work a SW element is doing. A class with low cohesion does unrelated things or too much work. Such classes are undesirable; they suffer from the following problems:

[1] Hard to comprehend
[2] Hard to reuse
[3] Hard to maintain
[4] Delicate; constantly affected by change

Low cohesion classes represented a very "large grain" of abstraction, or have taken on responsibilities that should have been delegated to other obj.

Problem	How to keep objects focused, understandable, manageable, maintainable and a side effect? How to support low coupling?
Solution (advice)	✓ Assign responsibilities so that cohesion remains high and object's responsibilities are closely related ✓ Evaluate alternatives to optimize cohesion ✓ 'Don't spread yourself too thin' ✓ 'Teamwork'

High Cohesion benefits

- Clarity and ease of comprehension of the design is increased
- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

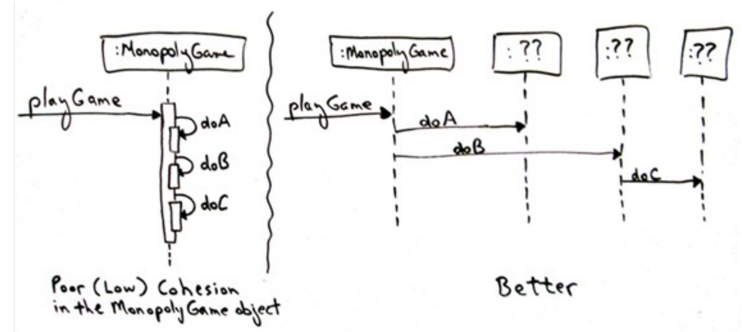
Related Patterns

- Low coupling

Cohesion vs. Coupling→

- Inherent trade-offs
- To minimize coupling, a few objects have all responsibility: Objects will be non-cohesive
- To maximize cohesion, a lot of objects have limited responsibility: Objects will be highly coupled to each other
- Choose best trade-off from alternative designs

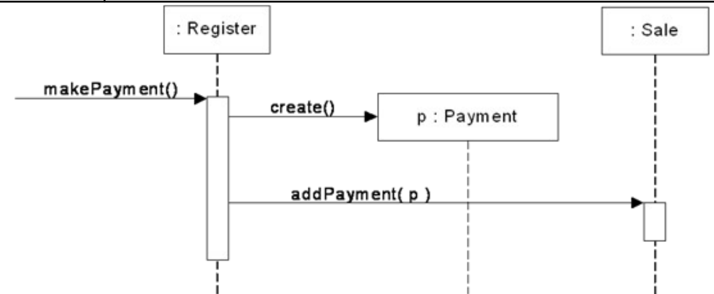
5.1 Monopoly example



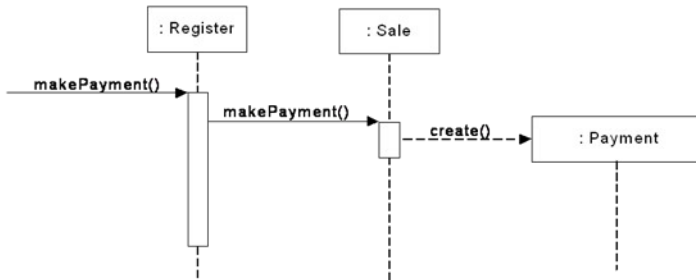
Design alternatives to support high cohesion

5.2 NextGen POS Example

Problem	Who should be responsible for creating a Payment?
Solution (advice)	✓ By Creator, Register should since it has the info ✓ Register may become incohesive if it does too much (and it's a Controller!) ✓ Let Register delegate to Sale



Register (a Controller), also creates a Payment, leading to lower cohesion
Register must also add the Payment to a Sale



Register delegates Payment creation to Sale
Register is more cohesive & there's less coupling

Levels of cohesion→

- Very low: class has lots of unrelated responsibilities
- Low: responsible for complex task in one area
- Moderate: sole responsibilities in a few areas related to the class but not to each other
- High: moderate responsibilities in one area & collaborates with others

6. Polymorphism

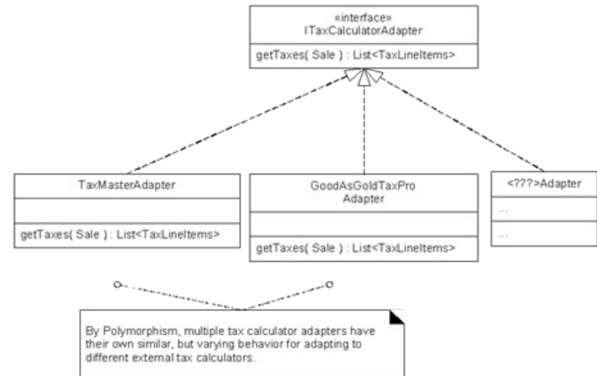
Problem	• How to handle alternatives (variations) based on type? • How to insulate the client from changes in the server? (i.e., pluggable SW)
Solution (advice)	• When behavior varies based on type, use polymorphic operations to encapsulate the variation • Eliminates lots of conditional logic (if/else/switch) based on type

Polymorphism benefits

- Extensions required for new variants are easy to add.
- New implementations can be easily introduced w/o affecting clients.

Related Patterns

- PV
- A number of popular GOF design patterns including Adapter, Command, Composite, Proxy, State & Strategy



Each tax calculator has a unique API. NextGen (client) should be immune to any specific calculator
Abstract interface has getTaxes() method. Specific adapters inherit from interface.
Each concrete adapter calls the actual tax calculator

- Using interfaces vs. superclasses
  - Some OO languages are single inheritance
  - Only get 1 chance to subclass
  - Classes may implement multiple interfaces
  - Interfaces give you multiple chances to use polymorphism on a single class
  - Polymorphism can be overused
    - Speculative future-proofing

7. Pure Fabrication

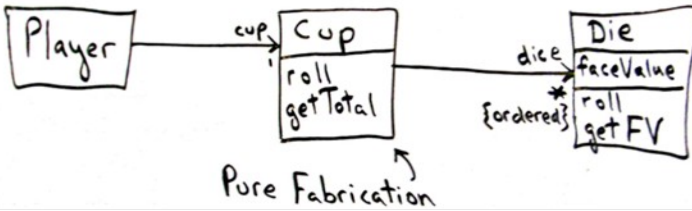
Problem	• What object should be responsible when Expert causes high coupling/low cohesion?
Solution (advice)	• Assign responsibility to a class that does not represent a concept in the domain • Create a fictitious class

Pure Fabrication benefits

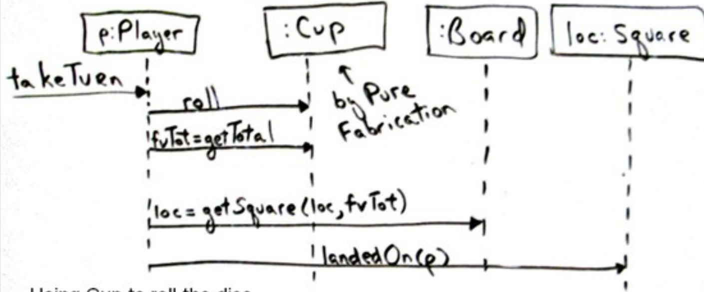
- High cohesion is supported because responsibilities are factored into a fine-grained class that only focuses on a very specific set of related tasks.
- Reuse potential may increase because of the presence of fine-grained Pure Fabrication classes whose responsibilities have applicability in other applications.

Related Patterns

- Low coupling
- High cohesion
- PF usually takes on responsibilities from the domain class that would be assigned those responsibilities based on the Expert pattern
- All GOF design patterns including Adapter, Command, Strategy and so on are Pure Fabrications



If Player rolls dice, then dice rolling behavior not very reusable
Cup is a Pure Fabrication responsible for rolling the dice
Can you think of a better name for this Pure Fabrication?



Using Cup to roll the dice
See any inconsistency WRT previous figure?

- Two basic approaches to design obj.
  - Representation decomposition
    - Design objects based on what they represent in the domain
    - What we've been doing to date
  - Behavioral decomposition
    - Design objects based on what they do
    - Function-centric, encapsulate an algorithm or special method
    - Commonly involve a Pure Fabrication
- Commonly used to encapsulate an algorithm that doesn't fit well in other classes
- When overused, model has lots of classes that encapsulate a single method

8. Indirection

Problem	• How to assign responsibility in order to avoid direct coupling that is undesirable? • Especially when server object is highly unstable or could result in vendor lock-in (e.g., proprietary API)
Solution (advice)	• Assign to an intermediate object that mediates between client-server

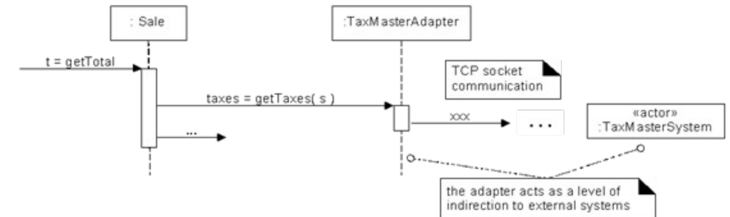
Indirection benefits

- Lower coupling btw components

Related Patterns

- Protected Variations
- Low Coupling
- Many GOF design patterns including Adapter, Bridge, Façade, Observer and Mediator

- Many Indirection intermediaries are Pure Fabrication



TaxMasterAdapter is a Pure Fabrication offering a level of Indirection
Shields client (Sale) from variable server (proprietary tax calculator system)

- a ubiquitous architectural principle
- "Most problems in CS can be solved by adding another level of indirection"
- "Most performance issues can be solved by removing a level of indirection"

9. Protected Variations

Problem	• How to design a server (object, module, system. etc.) so that clients are shielded from variations in the server. • How to design obj,, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on another elements?
Solution (advice)	• Create a stable server interface that shields clients from points of instability. • Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

PV benefits

- Extensions required for new variations are easy to add.
- New implementations can be introduced w/o affecting clients
- Coupling is lowered
- The impact or cost of changes can be lowered

Related Patterns

- Most designs principles and patterns are mechanisms for PV, including Polymorphism, Interfaces, Indirection, Data Encapsulation
- GOF patterns
- Variation & evolution points are called "hot spots"

A foundational design principle

- Used in conjunction with Polymorphism & Indirection
- Advanced examples
- **Data-Driven design**
  - Protect from variation in program parameters
  - Read parameters from external file during runtime (e.g., Java properties file)

- **Service Look-up**
  - Protect from variation in location of services
  - Find service by using external naming directory (e.g., JNDI)
- **Interpreter-Driven design**
  - Protect from variation in business logic
  - Rule-based system handles business rules
  - Separates rules (variable) from work flow (stable)
- **Reflective (meta-) design**
  - Special case of data-driven design
- **Uniform access design**
  - Protect from variation in type of object member
  - Methods & attributes invoked the same way (C#)

GoF Patterns

- GoF = Gang-of-Four Patterns are *design* patterns
- Used to solve design-related issues
- Patterns simplify but proliferation of patterns adds complexity.
- It illustrates that obj. design and the assignment of responsibilities can be explained and learned based on the application of patterns.
- Some GRASP principles can be viewed as a Generalization of GoF Patterns.
  1. Adapter
  2. Factory
  3. Singleton
  4. Strategy
  5. Composite
  6. Façade
  7. Observer

1. Adapter – One of the most useful

Problem	• How to resolve incompatible interfaces? • How to provide a stable interface to similar components with different interfaces?
Solution (advice)	• Convert the original interface of a component into another interface, through an intermediate adapter object. • Hide the incompatible/unstable interface behind the adapter's interface • Client collaborates with stable adapter • Adapter relays messages to unstable interface • Uses Protected Variations, Polymorphism, Indirection

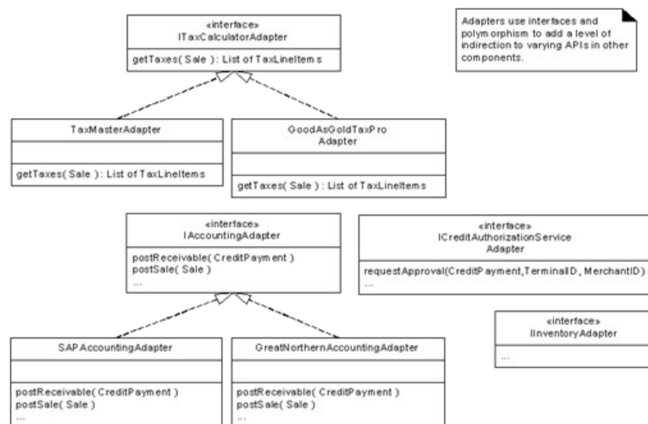
Adapter benefits

- The previous use of the Adapter pattern can be viewed as a specialization of some GRASP building blocks:

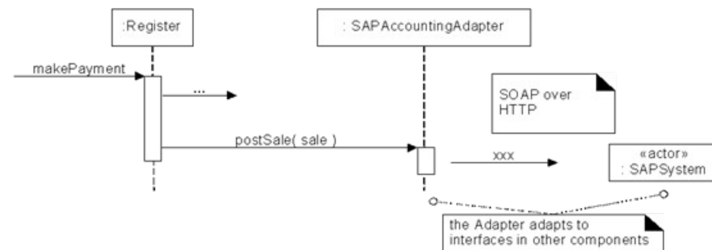
"*Adapter* supports *Protected Variations* with respect to changing external interfaces or third-party packages through the use of *Indirection* object that applies interfaces and *Polymorphism*"

Related Patterns

- A resource Adapter that hides an external system may also be considered as a Façade object , as it wraps access to the subsystem or system with a single object (the essence of Façade). The motivation here to call it Adapter exists when the wrapping obj. provides adaptation to various external interfaces.



Clients send message to Adapter; Enables Plug n' Play WRT 3rd party SW!
Notice adherence to good naming conventions



Register posts the sale to an Adapter
Adapter communicates with the SAP accounting system
SAP accounting system exposes functionality as a *web service*

2. Factory (aka Simple or Concrete Factory)→

Simplified version of GOF Abstract Factory

Problem	• Who should be the Creator when creation causes incohesiveness, or involves complex creation logic
Solution (advice)	• Create a Pure Fabrication object called a Factory that handles the creation. • Commonly implemented via the Singleton pattern

Factory advantages

- Separate the responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic.
- Allow introduction of performance-enhancing memory management strategies, such as object cashing or recycling.

Related Patterns

- Factories are often accessed with the Singleton pattern

3. Singleton

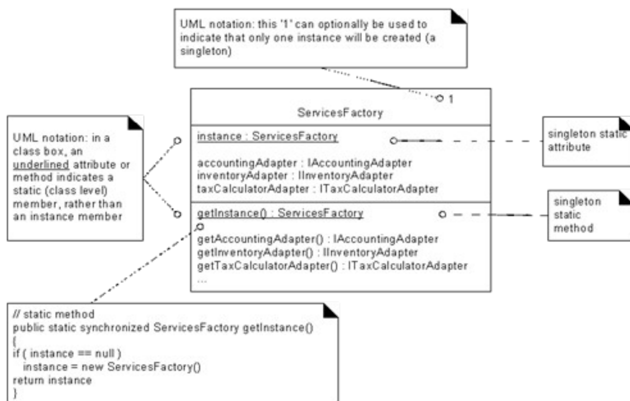
Problem	• Exactly one instance of a class is needed/allowed • Other objects need single, global point of access to it. • Who should create the Factory? The Singleton!
Solution (advice)	• Define a static method of a class that returns the Singleton • The static method can only create one instance

Singleton benefits

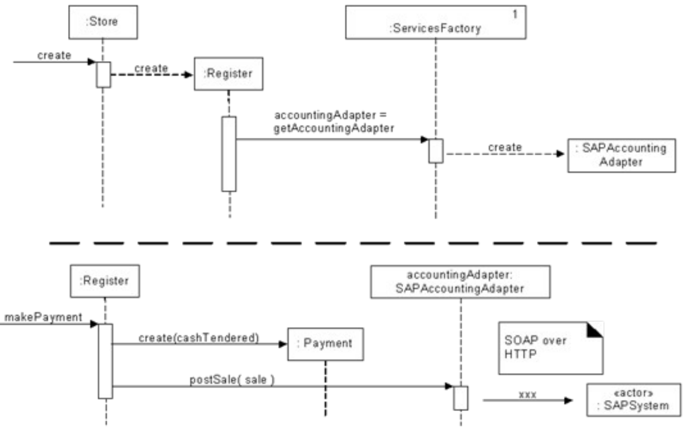
- Provides global visibility via static method
- Avoids passing factory reference to many clients

Related Patterns

- Singleton pattern often used to for Factor and Façade objects.



All clients have global visibility to static method getInstance()
First call to getInstance() creates a ServicesFactory singleton
Subsequent calls just access the singleton



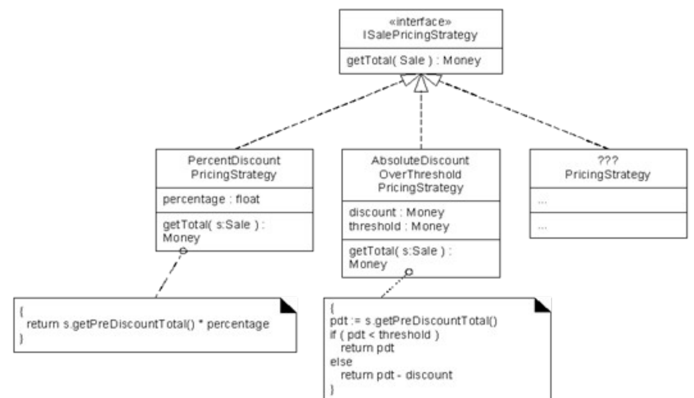
Adapter, Factory & Singleton used together

4. Strategy

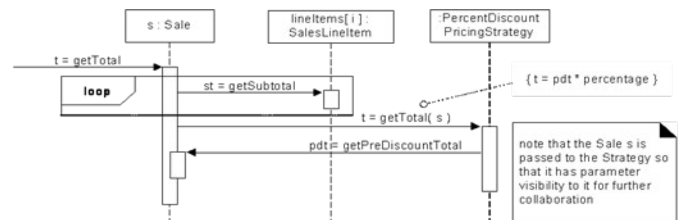
Problem	• How to design for varying, but related, algorithms, policies or business rules? • How to design for the ability to change these algorithms • Yet insulate clients from algorithmic details
Solution (advice)	• Define separate class for each algorithm, policy or rule with a common interface

Related Patterns

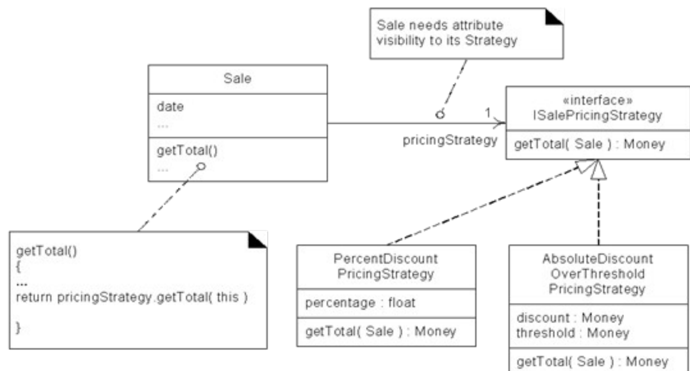
- Strategy is based on polymorphism and provides Protected Variations with respect to changing algorithms. Strategies are often created by a Factory.



Each pricing strategy subclass implements a different pricing algorithm
All implement the getTotal(..) method
Strategy object attached to a context object (Sale, in this case)



Strategy collaboration
Client tells Sale to getTotal(), Sale gets subtotals from all SaleLineItems
Sale passes 'this' reference to its strategy object
So strategy can obtain needed data from its context object, Sale



Strategy collaboration class diagram
Sale has attribute visibility to its strategy object
Strategy has parameter visibility back to its context object, Sale

5. Composite

Problem	• How to treat a group of objects the same way as though they were a single object?
Solution (advice)	• Define a composite object that has the same interface as the atomic objects • Composite object can contain many atomic objects…all of which can be invoked by same method

Composite benefits

- A strategy object implements a specific atomic algorithm/rule/policy
- To handle compound rules, we need an object that is composed of several atomic strategy objects
- Allows customized rules based on blending of atomic rules
- Composite object has an attribute that is a list of atomic objects

Related Patterns

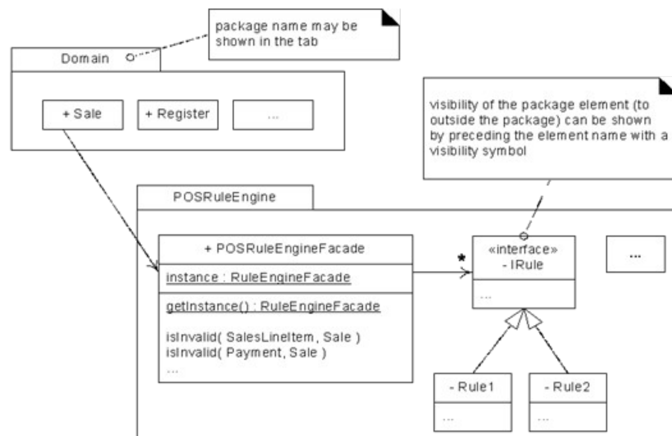
- Often used with Strategy and Command patterns.
- Campsite is based on polymorphism and provide PV to a client so that it is not impacted if its related objects are atomic or composite.

6. Facade

Problem	• Need a common interface to a variable set of interfaces, perhaps within a subsystem • Want to minimize coupling to the interfaces • Want to expose only a subset of the interfaces' functionality
Solution (advice)	• Define an object that represents a single point-of-contact to the set of interfaces • Clients communicate with 'front-end' Façade object • Essentially an Adapter for multiple interfaces

Related Patterns

- Usually accessed via Singleton pattern. They provide PV from the implementation of a subsystem , by adding an Indirection object to help support low coupling. External objects in a system are coupled to one point: the façade



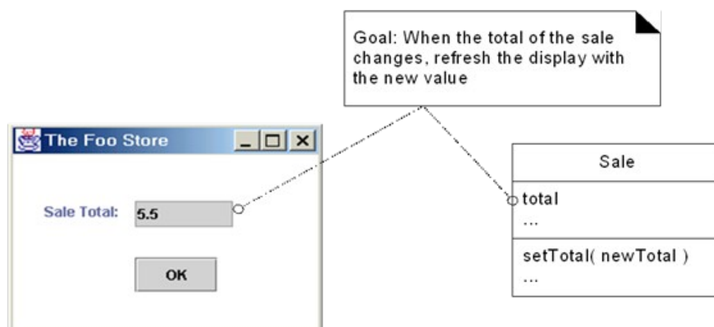
Facades often accessed via a Singleton (single point of contact!)
Notice all objects needed by rule engine passed as parameters
Uses Indirection via a Pure Fabrication to achieve Low Coupling

7. Observer/Publish-Subscribe/Delegation

Problem	• An observer (e.g., a GUI) needs to know about state changes in a publisher (e.g. a domain object) without direct coupling between objects
Solution (advice)	• Subscribers implement a 'Listener' interface • Publishers dynamically register listeners • Publishers automatically notify listeners when event occurs • Publishers coupled to generic interface instead of concrete object

Related Patterns

- Observer is based on Polymorphism; and provide PV in terms of protecting the publisher from knowing the specific class of object, and a number of objects.



How to update GUI when a Sale total changes w/o directly coupling both?
Want to maintain plug n' play WRT UIs