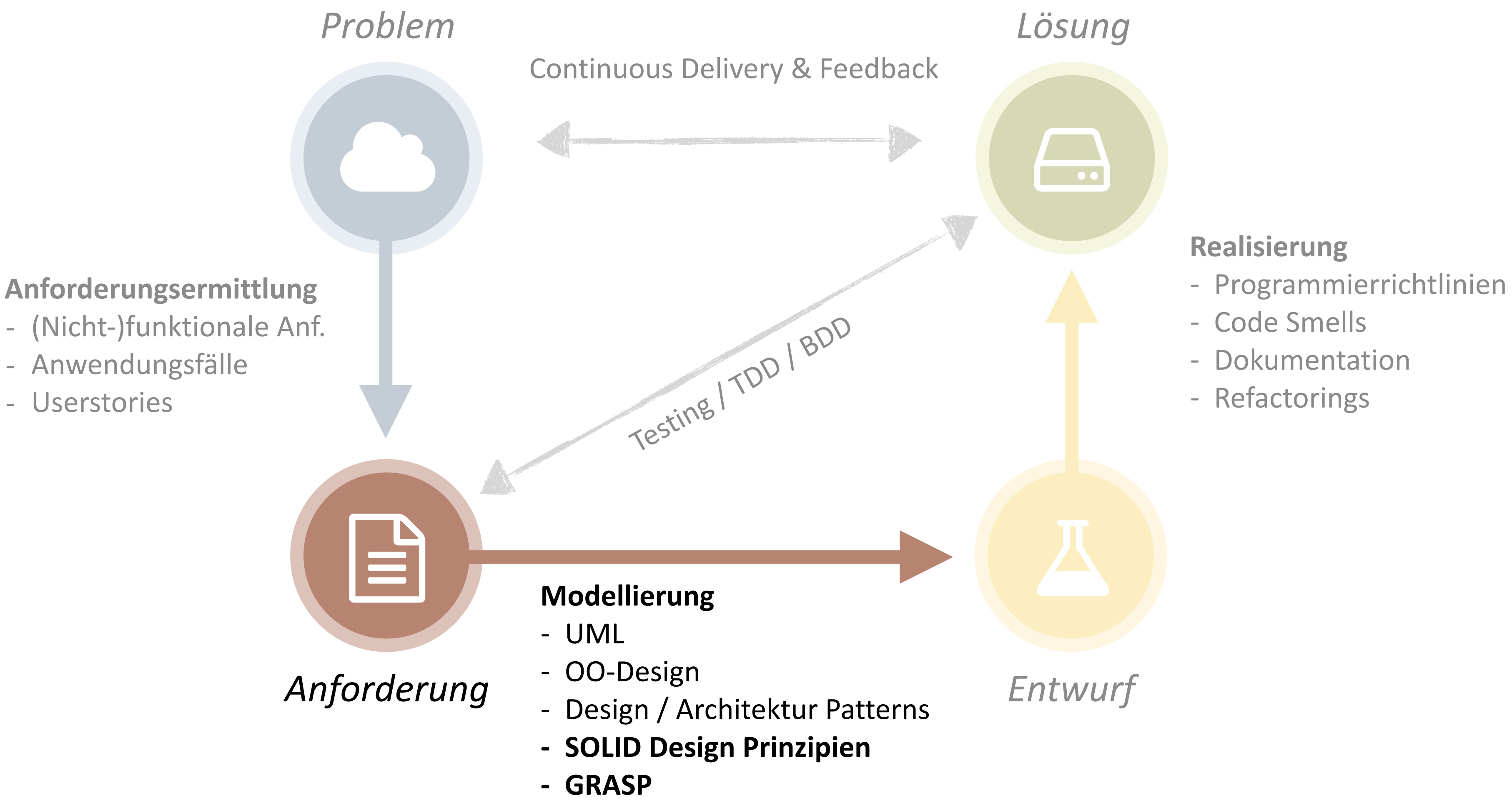
Software Engineering
5. Software Design und Design Prinzipien

Jonathan Brachthäuser

Einordnung



Anforderungsermittlung
- (Nicht-)funktionale Anf.
- Anwendungsfälle
- Userstories

Realisierung
- Programmierrichtlinien
- Code Smells
- Dokumentation
- Refactorings

Modellierung
- UML
- OO-Design
- Design / Architektur Patterns
- **SOLID Design Prinzipien**
- **GRASP**

Software Design

Goal of Software Design

- For each desired program behavior there are infinitely many programs that implement this behavior
  - What are the **differences** between the variants?
  - Which variant should we choose?
- Since we usually have to synthesize (i.e. create) rather than choose the solution…
  - How can we **design** a variant that has the desired properties?

Example - Variant A

- Sorting with configurable order, variant A

```
void sort(int[] list, String order) {
    …
   boolean mustswap;
   if (order.equals("up")) {
     mustswap = list[i] < list[j];
   } else if (order.equals("down")) {
     mustswap = list[i] > list[j];
   }
   …
}
```

Example - Variant B

- Sorting with configurable order, variant B

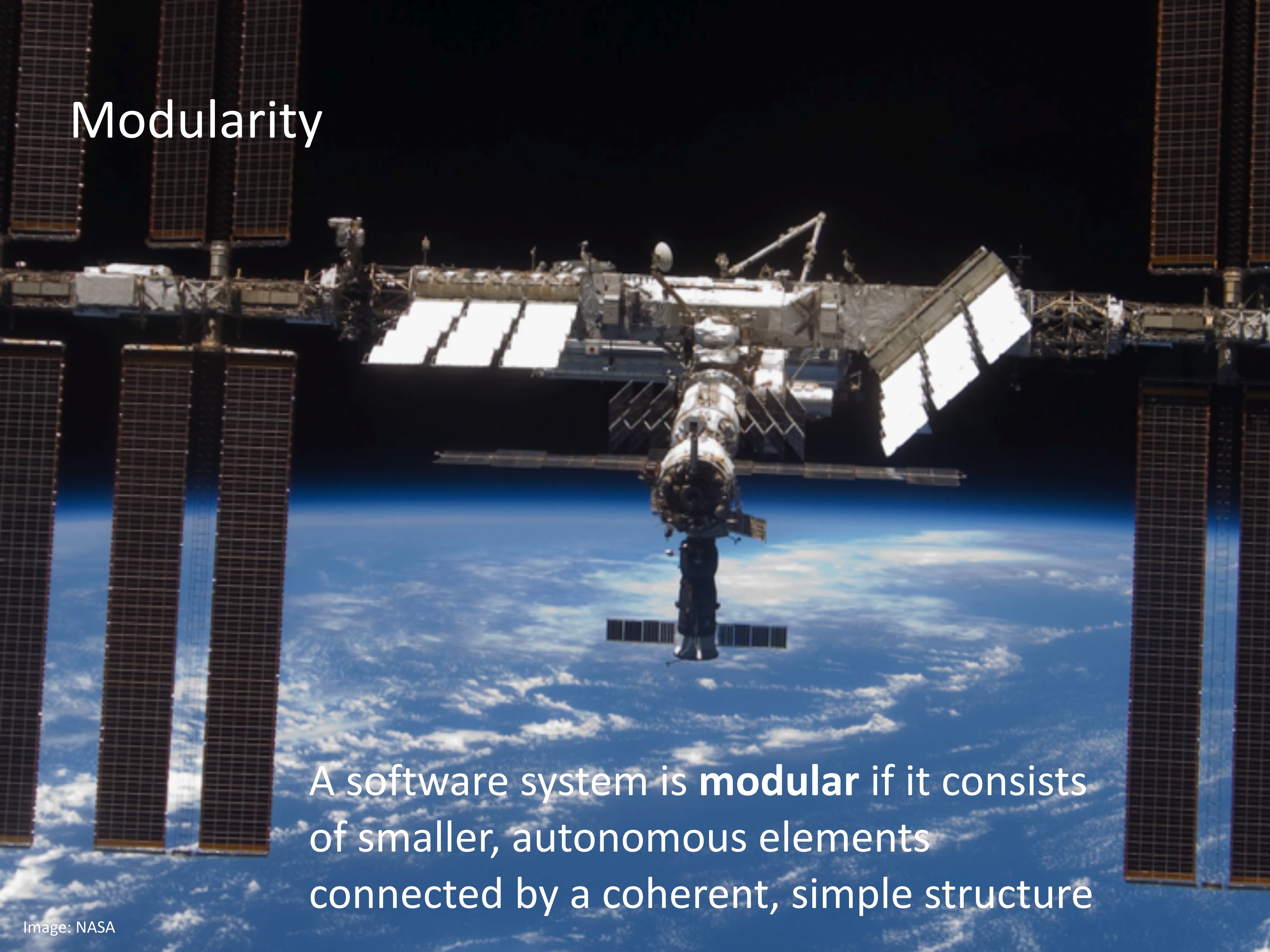
```
void sort(int[] list, Comparator cmp) {
    …
  boolean mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int i, int j) { return i < j; }}

class DownComparator implements Comparator {
  boolean compare(int i, int j) { return i > j; }}
```

(by the way, this design is called “strategy pattern”)

Quality of a Software Design

- How can we measure the **internal quality** of a software design?
  - Extensibility, Maintainability, Understandability, Readability, …
  - Robustness to change
  - Low Coupling & High Cohesion
  - Reusability
  - All these qualities are typically summarized by the term **modularity**
- …as opposed to **external quality**
  - Correctness: Valid implementation of requirements
  - Ease of Use
  - Resource consumption
  - Legal issues, political issues, …

Modularity



A software system is **modular** if it consists of smaller, autonomous elements connected by a coherent, simple structure

Image: NASA

Modularity

- In the following we’ll elaborate on that:
  - Five Criteria
  - Five Rules

Five Criteria: Modular Decomposability (1)

A software construction method satisfies **Modular Decomposability** if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.

Five Criteria: Modular Decomposability (1)

- Modular Decomposability implies: Division of Labor possible!
- Example: Top-Down Design
- Counter-Example:

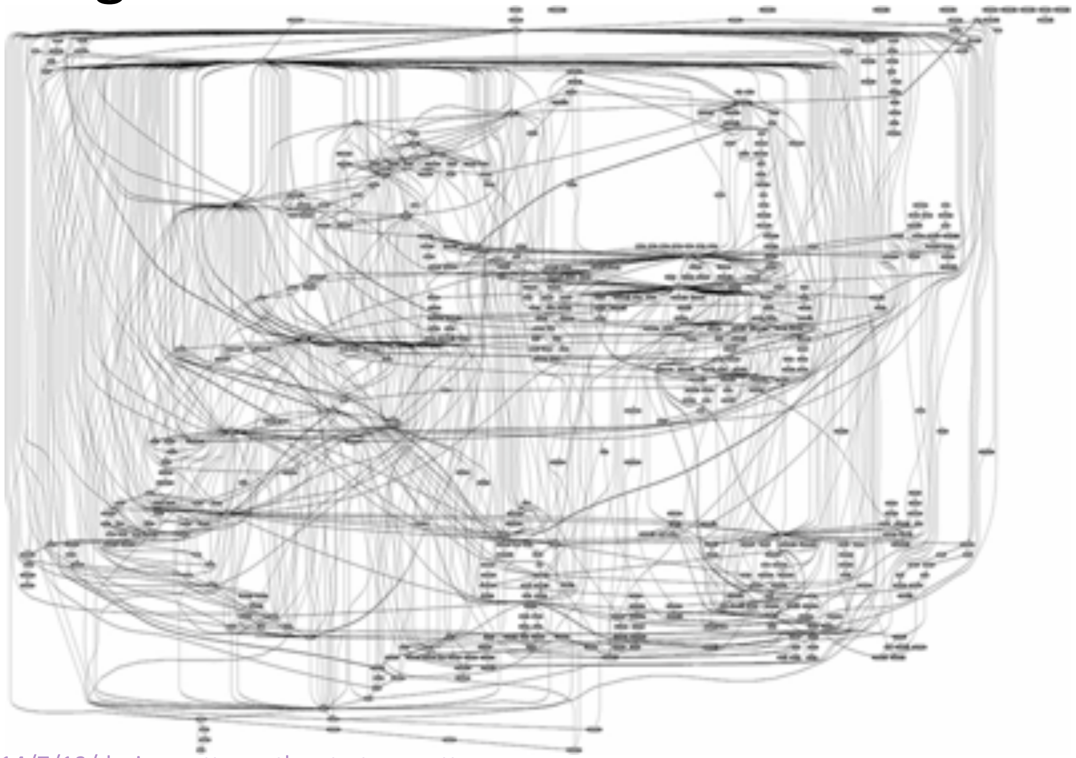


Image Source: http://www.darxstudios.com/darx-studios/2014/7/19/design-patterns-the-strategy-pattern

Five Criteria: Modular Composability (2)

A method satisfies **Modular Composability** if it favors the products of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

Five Criteria: Modular Composability (2)

- Is “dual” to modular decomposability
- Is directly connected with **reusability**
  - Old “dream” of programming: programming as construction box activity
- Well-defined interfaces are of utmost importance to achieve modular composability
- Example 1: Libraries have been reused successfully in countless domains
- Example 2: Unix Shell Commands

Five Criteria: Modular Understandability (3)

A method favors **Modular Understandability** if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

Five Criteria: Modular Understandability (3)

- Important for maintenance
- Applies to all software artifacts, not just code
- Again: Interfaces are needed to protect from unnecessary details
- Counter-examples:
  - Complex call graphs between many different modules
  - Complex use of object state and side effects across module boundaries

Five Criteria: Modular Continuity (4)

A method satisfies **Modular Continuity** if, in the software architectures that it yields, a small change in the problem specification will trigger a change of just one module, or a small number of modules.

Five Criteria: Modular Continuity (4)

- Example 1: Symbolic constants (as opposed to magic numbers)
- Example 2: Hiding data representation behind an interface
- Counter-Examples:
  - Program designs depending on fragile details of hardware or compiler

Five Criteria: Modular Protection (5)

A method satisfied **Modular Protection** if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

Five Criteria: Modular Protection (5)

- Motivation: Big software will always contain bugs etc., failures unavoidable
- Example: Defensive Programming
- Counter-Example: An erroneous null pointer in one module leads to an error in a different module

Five Rules

- Five Rules will follow which we must observe to ensure high-quality design

Five Rules: Direct Mapping

The modular structure devised in the process of **building a software system** should remain compatible with any modular structure devised in the process of **modeling the problem domain**.

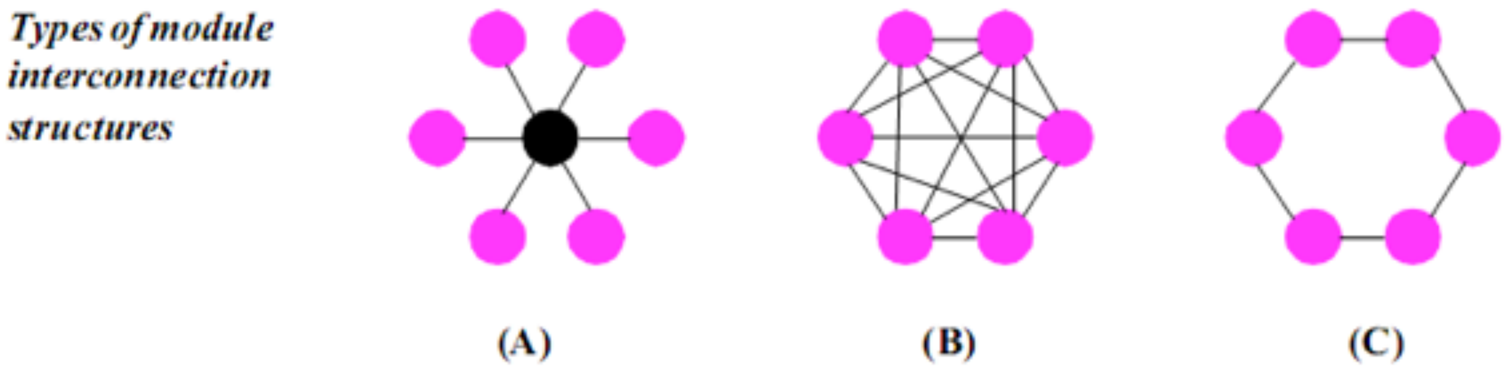
Five Rules: Direct Mapping

- Follows from continuity and decomposability
- A.k.a. “low representational gap”[C. Larman]

Five Rules: Few Interfaces

If two modules communicate, they should use **as few interfaces as possible**

Five Rules: Few Interfaces



- We want topology with few connections
- Follows from continuity and protection; otherwise changes/errors would propagate more

Five Rules: Small Interfaces

If two modules communicate, they should exchange **as little information as possible**

Five Rules: Small Interfaces

- Follows from continuity and protection, required for composability
- The more detailed the exchanged information is, the higher the coupling

Five Rules: Explicit Interfaces

Whenever two modules A and B communicate, **this must be obvious from the interface** of A or B or both.

Five Rules: Explicit Interfaces

- Counter-Example 1: Global Variables
- Counter-Example 2: Aliasing – mutation of shared heap structures

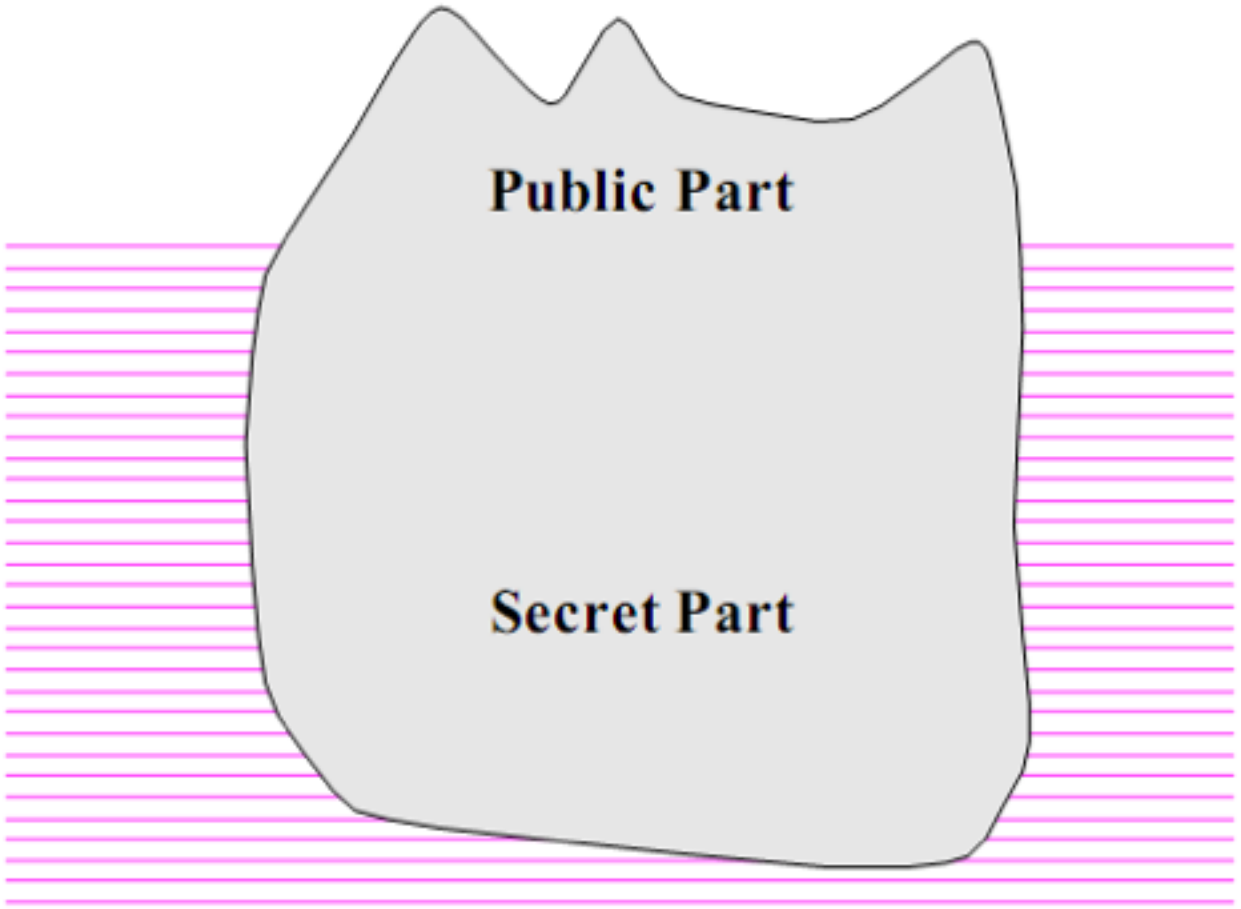
Intermezzo: Law of Demeter (LoD)

- LoD: Each module should have only limited knowledge about other units: only units "closely" related to the current unit
- In particular: Don’t talk to strangers!
- For instance, no a.getB().getC().foo()
- Motivated by continuity

Five Rules: Information Hiding

The designer of every module must select a s**ubset of the module's properties** as the official information about the module, to be made available to authors of client modules.

Five Rules: Information Hiding



Five Rules: Information Hiding

- Idea: hide implementation details that are likely to change
- Implied by continuity
- The iceberg analogy is slightly misleading, since an interface also *abstracts* over the implementation

GRASP

GRASP Patterns

- Object Design:
  - “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.”
  - But how?
    - What method belongs where?
    - How should the objects interact?
    - This is a critical, important, and non-trivial task

GRASP Patterns

- The GRASP patterns are a learning aid to
  - help one understand essential object design
  - apply design reasoning in a methodical, rational, explainable way.
- This approach to understanding and using design principles is based on patterns of assigning **responsibilities**

GRASP - Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - knowing
  - doing
- Doing responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software
- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
  - (technically "GRASP Patterns" is hence redundant but it sounds better)
- Describe fundamental principles of object design and responsibility
- Expressed as patterns

Nine GRASP patterns:

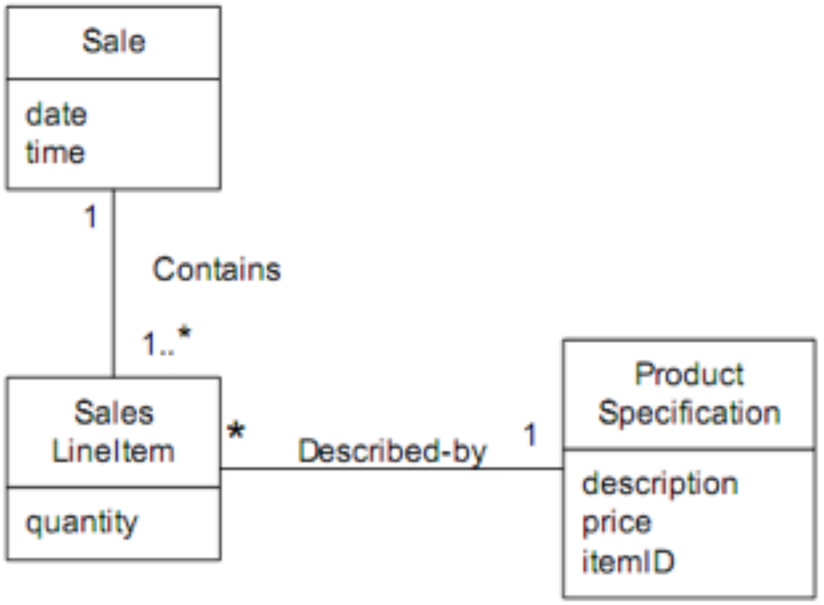
- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert
  - the class that has the information necessary to fulfill the responsibility
- Start assigning responsibilities by clearly stating responsibilities!
- For instance, in a POS application a statement might be: *“Who should be responsible for knowing the grand total of a sale"?*

Information Expert

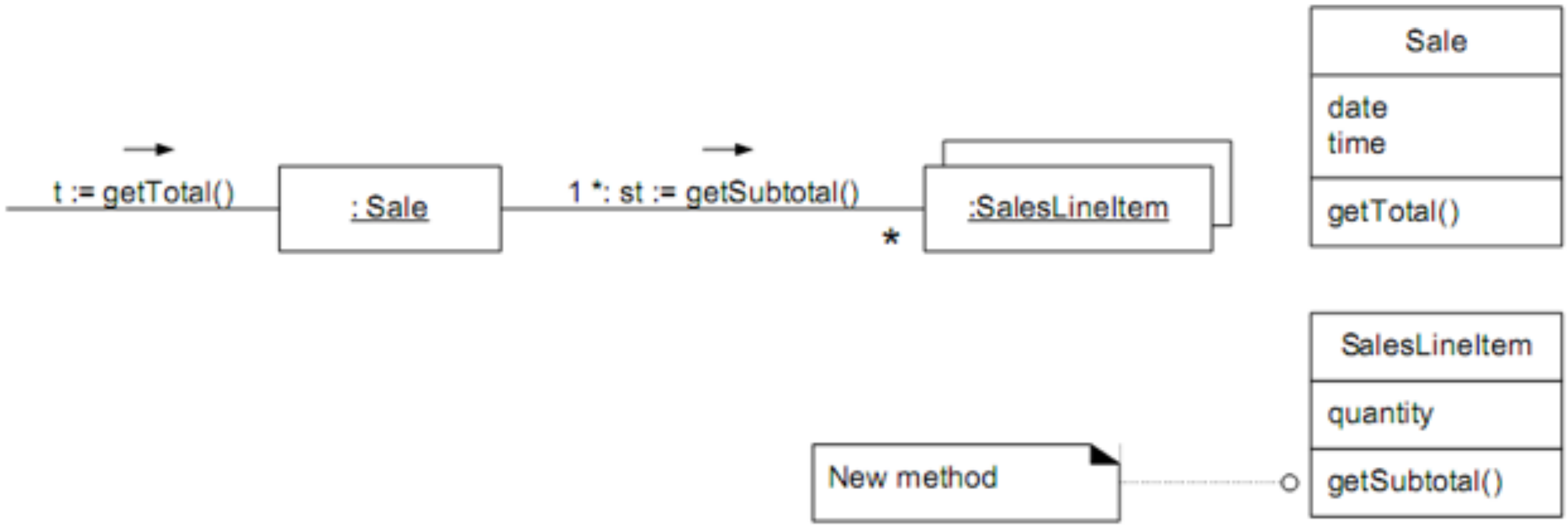
- What information is needed to determine the grand total?



- *Sale* is the information expert for this responsibility.

Information Expert

- What information is needed to determine the line item subtotal?

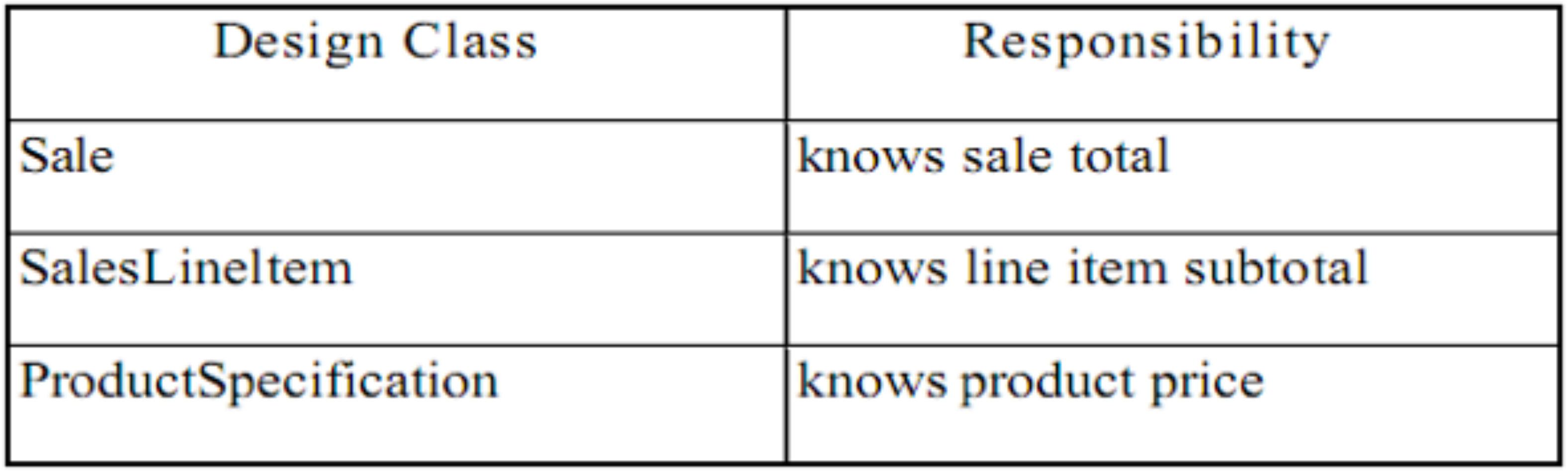


Information Expert

- To fulfill the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is an information expert on answering its price; therefore, a message must be sent to it asking for its price.

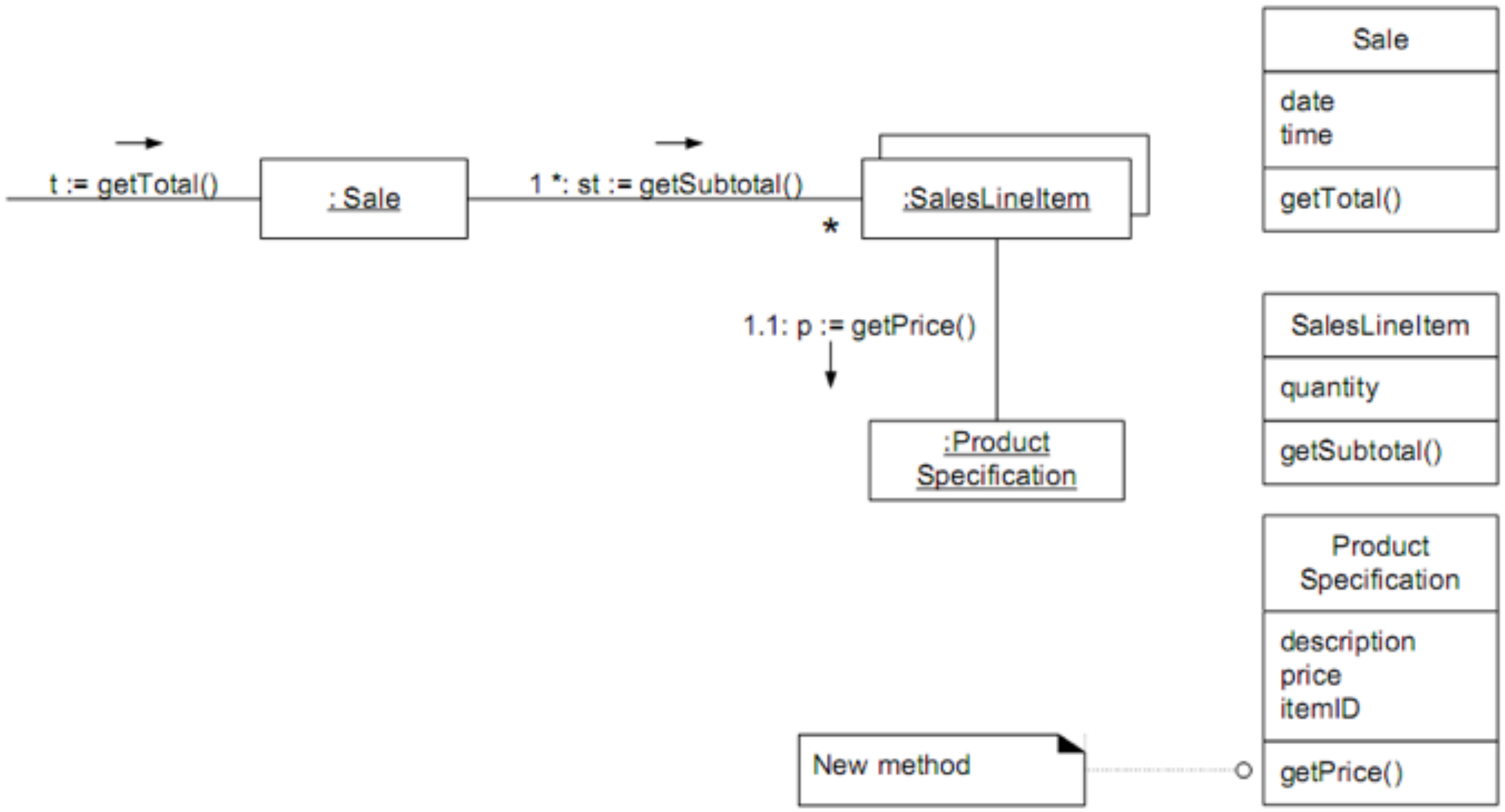
Information Expert

- To fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Information Expert: Final Design



Information Expert: Discussion

- Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
  - a sale does not tell you its total; it is an inanimate thing
- In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
- They do things related to the information they know.

Information Expert: Discussion

- Contraindication: Conflict with separation of concerns
  - Example: Who is responsible for saving a sale in the database?
  - Adding this responsibility to Sale would distribute database logic over many classes → low cohesion
- Contraindication: Conflict with late binding
  - Late binding is available only for the receiver object
  - But maybe the variability of late binding is needed in some method argument instead
  - Example: Support for multiple serialization strategies

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Creator

Problem:

Assign responsibility for creating a new instance of some class?

Solution:

Determine which class should create instances of a class based on the relationship between potential creator classes and the class to be instantiated.

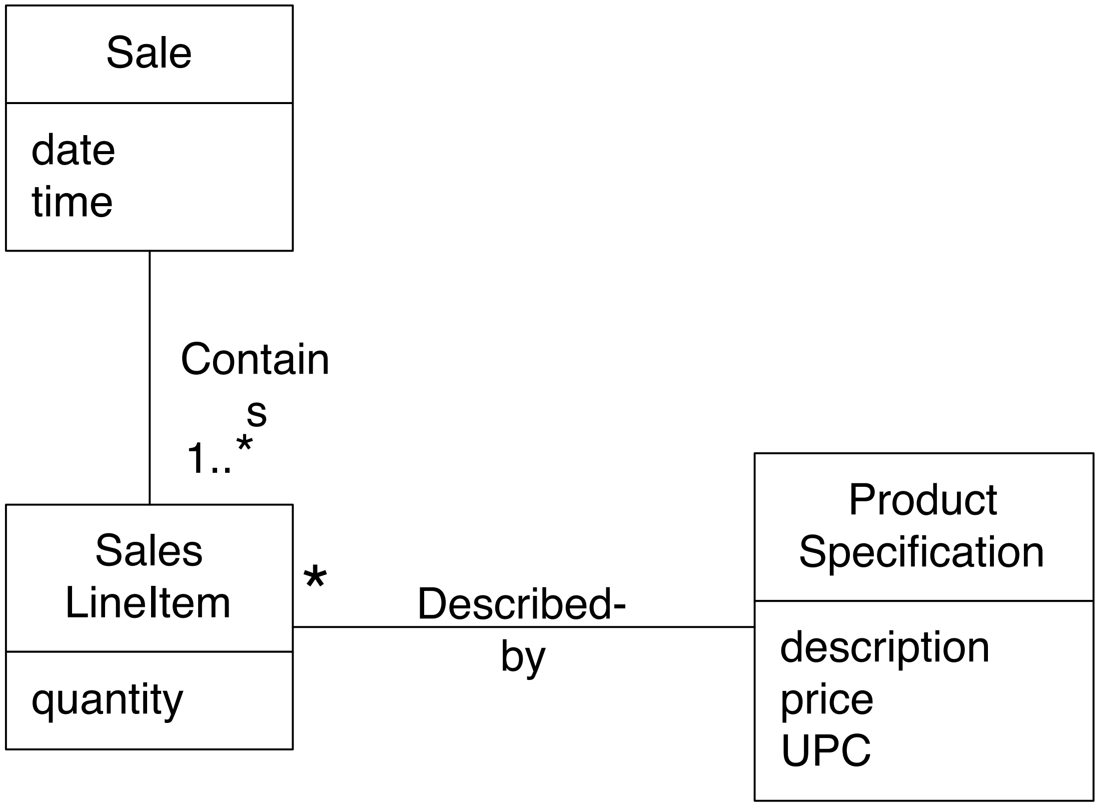
Creator

- who has responsibility to create an object?
- By creator, assign class B responsibility of creating instance of class A if
  - B *aggregates* A objects
  - B *contains* A objects
  - B *records* instances of A objects
  - B *closely* uses A objects
  - B *has the initializing data* for creating A objects
- where there is a choice, prefer
  - B *aggregates* or *contains* A objects

Creator : Example

Who is responsible for creating *SalesLineItem* objects?

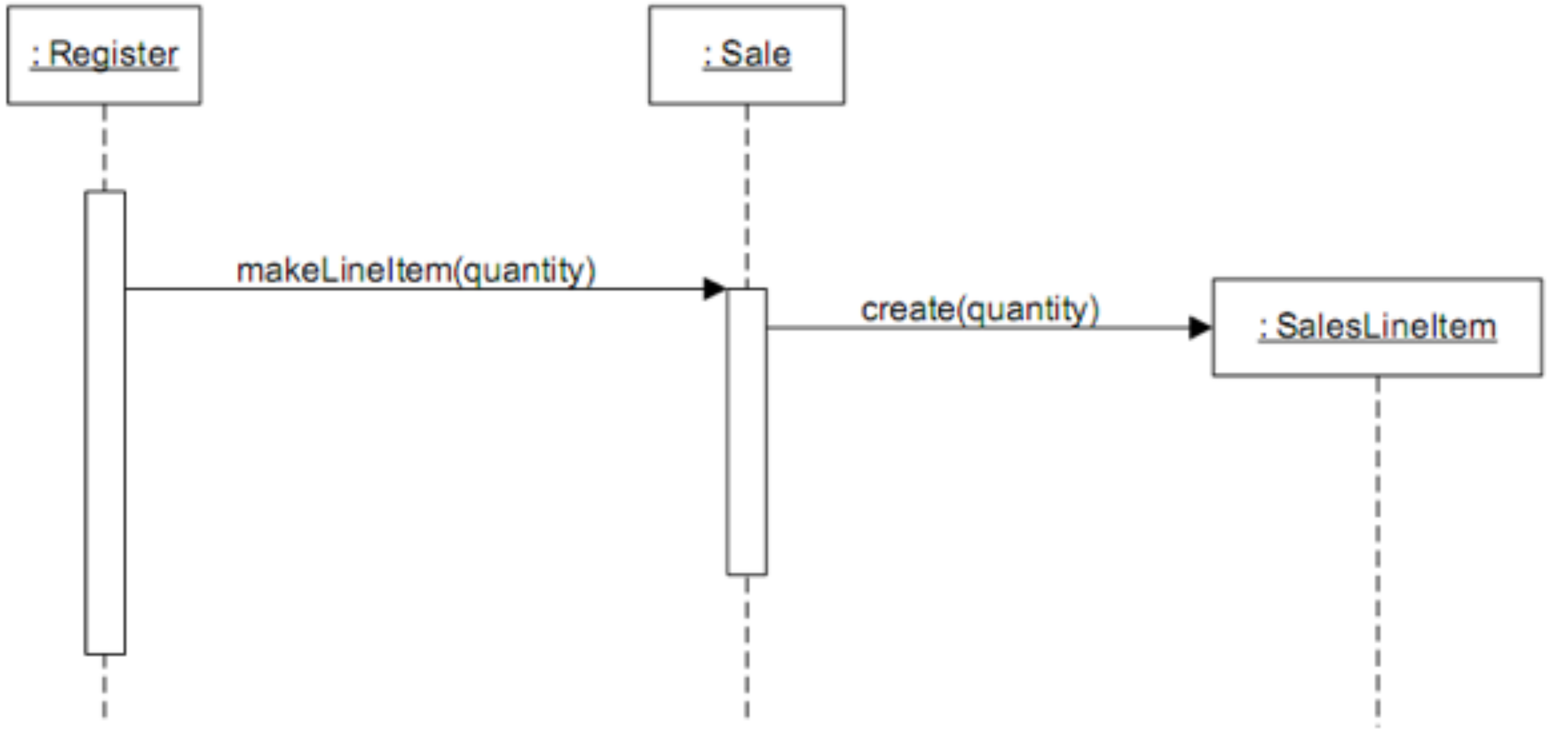
Look for a class that aggregates or contains *SalesLineItem* objects.



Creator : Example

Creator pattern suggests Sale.

Collaboration diagram is



Creator

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference
- By creating the objects themselves, they avoid being dependent on another class to create the object for them

Creator: Discussion

- Contraindications:
  - creation may require significant complexity, such as
    - using recycled instances for performance reasons
    - conditionally creating an instance from one of a family of similar classes based upon some external property value
    - Sometimes desired to outsource object wiring ("dependency injection")
- Related patterns:
  - Abstract Factory, Singleton, Dependency Injection

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Low Coupling

Problem:

How to support low dependency, low change impact, and increased reuse.

Solution:

Assign a responsibility so that coupling remains low.

Why High Coupling is undesirable

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements (classes, subsystems, …)
  - "too many" is context-dependent
- A class with high (or strong) coupling relies on many other classes.
  - Changes in related classes force local changes.
  - Such classes are harder to understand in isolation.
  - They are harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Low Coupling

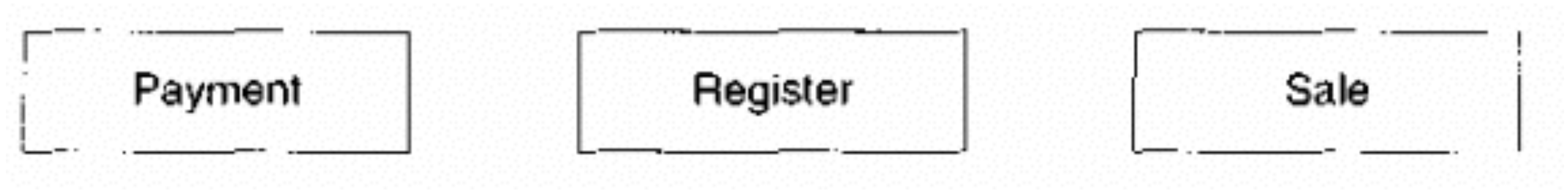
How can we make classes independent of other classes?

changes are localised

easier to understand

easier to reuse

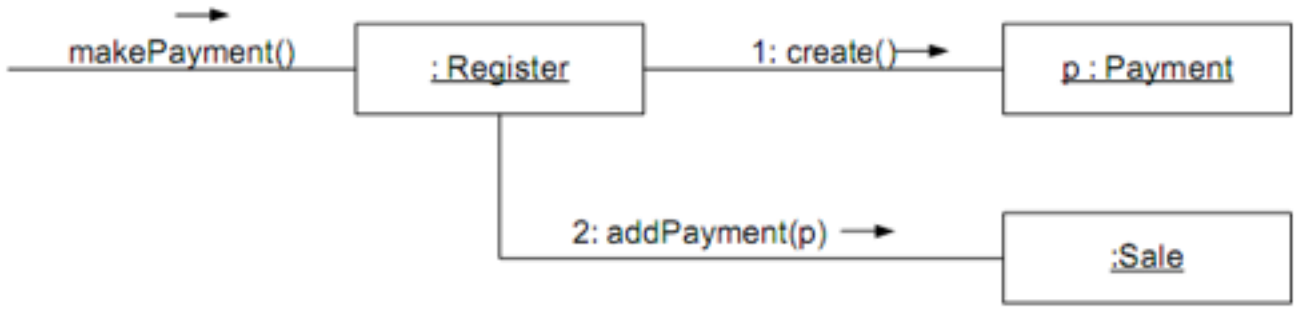
Who has responsibility to create a *payment and associate it to a sale*?



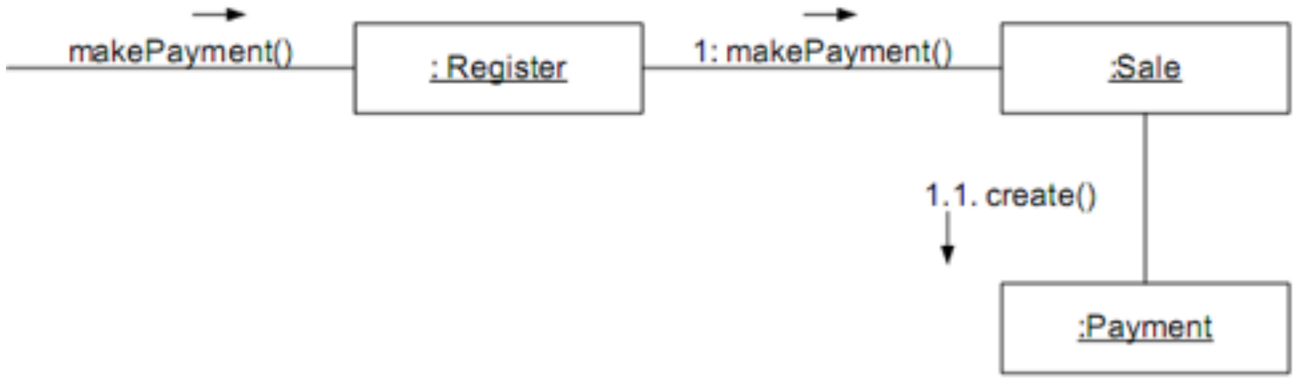
Low Coupling

Two possibilities:

1. Register



2. Sale



Low coupling suggests *Sale* because *Sale* has to be coupled to *Payment* anyway (*Sale* knows its *total*).

Common Forms of Coupling in OO Languages

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- TypeX has a method which references an instance of TypeY, or TypeY itself, by any means.
  - Typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

Low Coupling: Discussion

- Low Coupling is a principle to keep in mind during all design decisions
- It is an underlying goal to continually consider.
- It is an *evaluative principle* that a designer applies while evaluating all design decisions.
- Low Coupling supports the design of classes that are more independent
  - reduces the impact of change.
- Can't be considered in isolation from other patterns such as Expert and High Cohesion
- Needs to be included as one of several design principles that influence a choice in assigning a responsibility.

Low Coupling: Discussion

- Subclassing produces a particularly problematic form of high coupling
  - Dependence on implementation details of superclass
  - “Fragile Base Class Problem” [see SE Design Lecture]
- Extremely low coupling may lead to a poor design
  - Few incohesive, bloated classes do all the work; all other classes are just data containers
- Contraindications: High coupling to very stable elements is usually not problematic

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

High Cohesion

Problem:

How to keep complexity manageable.

Solution:

Assign responsibilities so that cohesion remains high.

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion

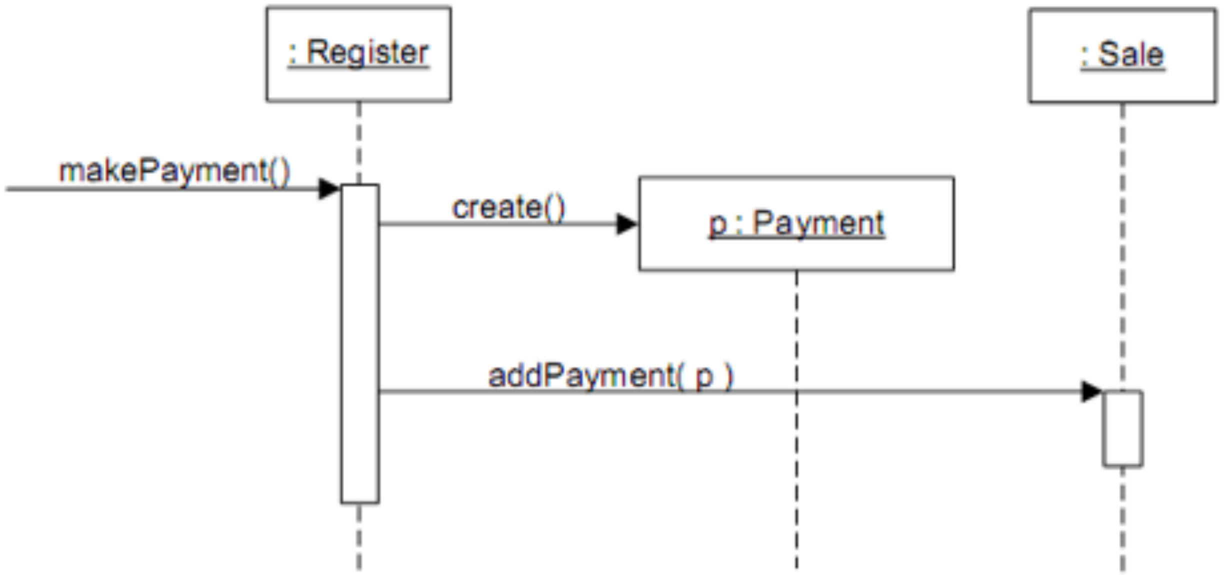
High cohesion

- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility

High Cohesion

Who has responsibility to create a *payment*?

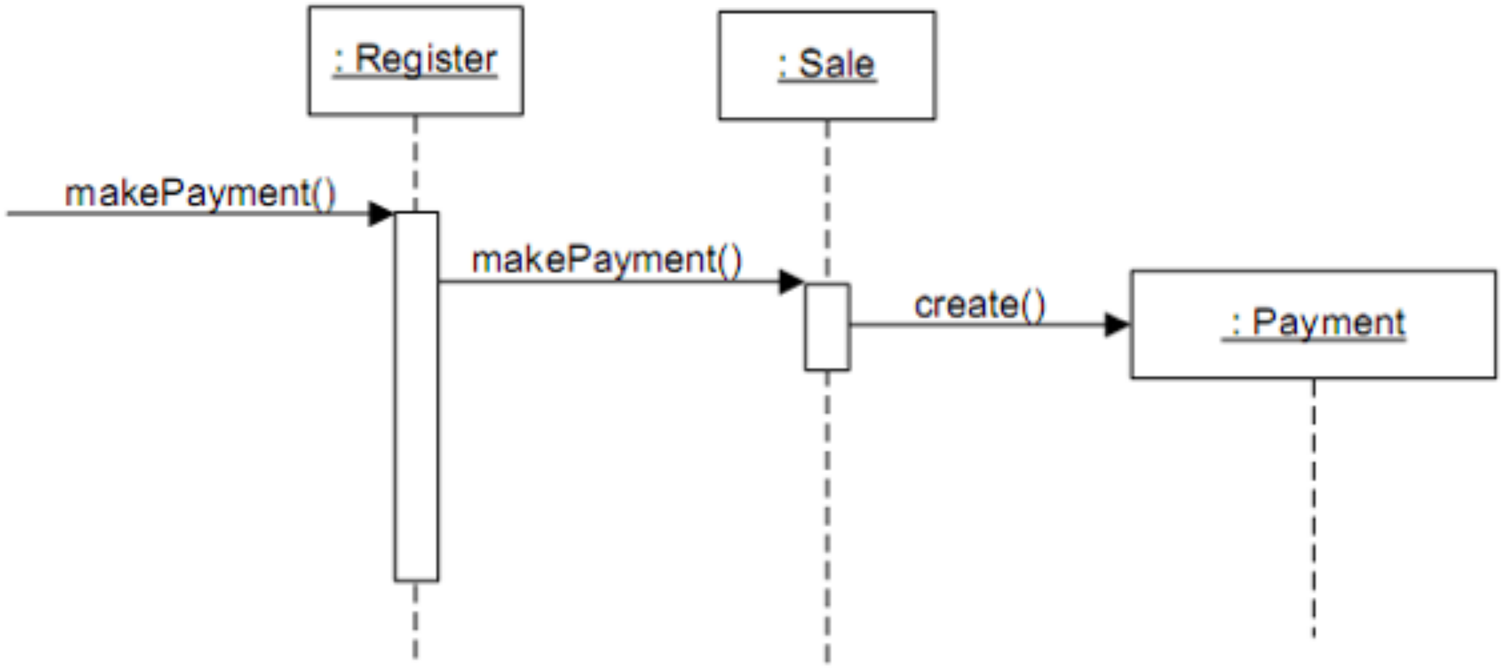
1. Register



looks OK if *makePayement* considered in isolation, but adding more system operations, *Register* would take on more and more responsibilities and become less cohesive.

High Cohesion

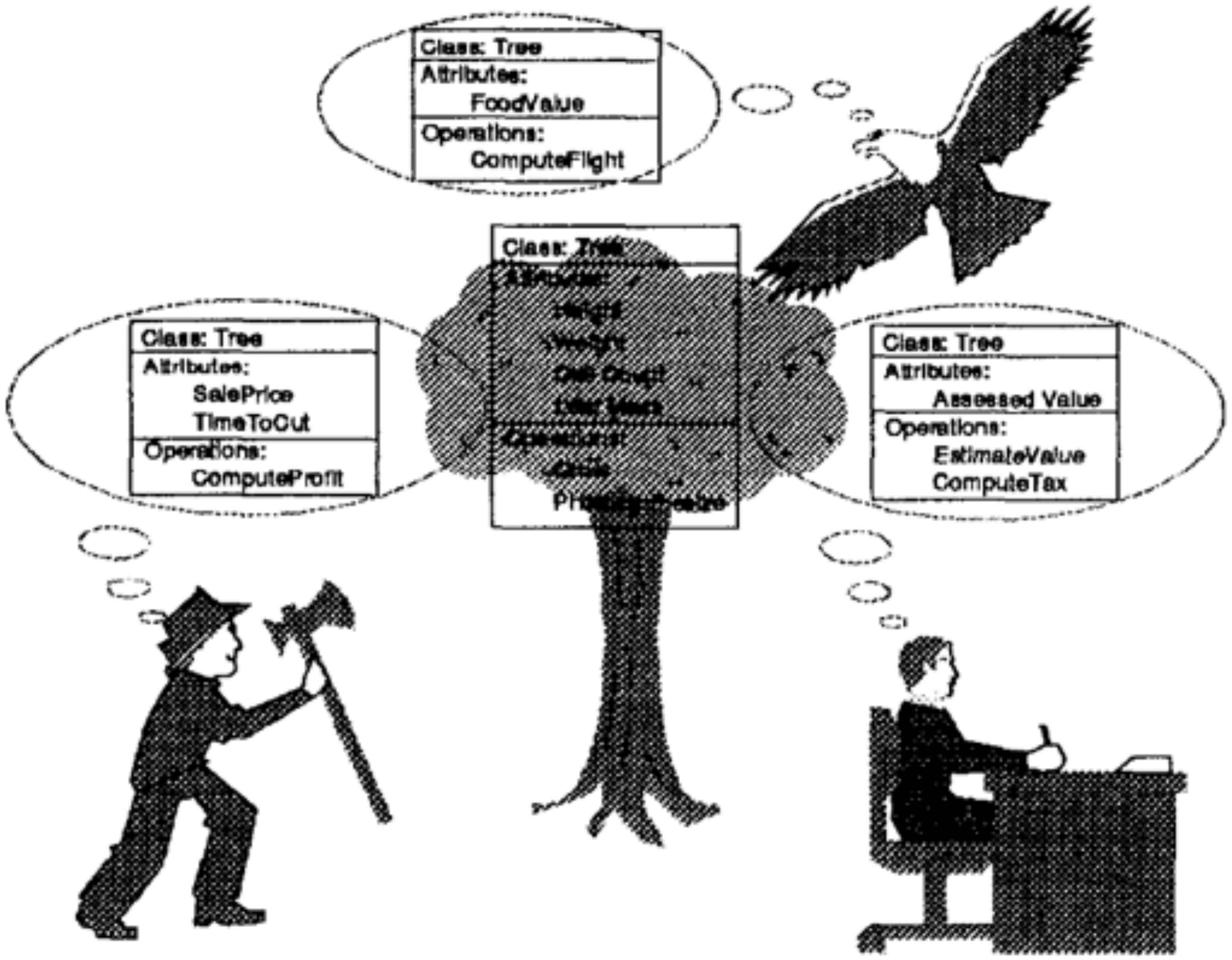
Giving responsibility to *Sale* supports higher cohesion in *Register*, as well as low coupling.



High Cohesion: Discussion

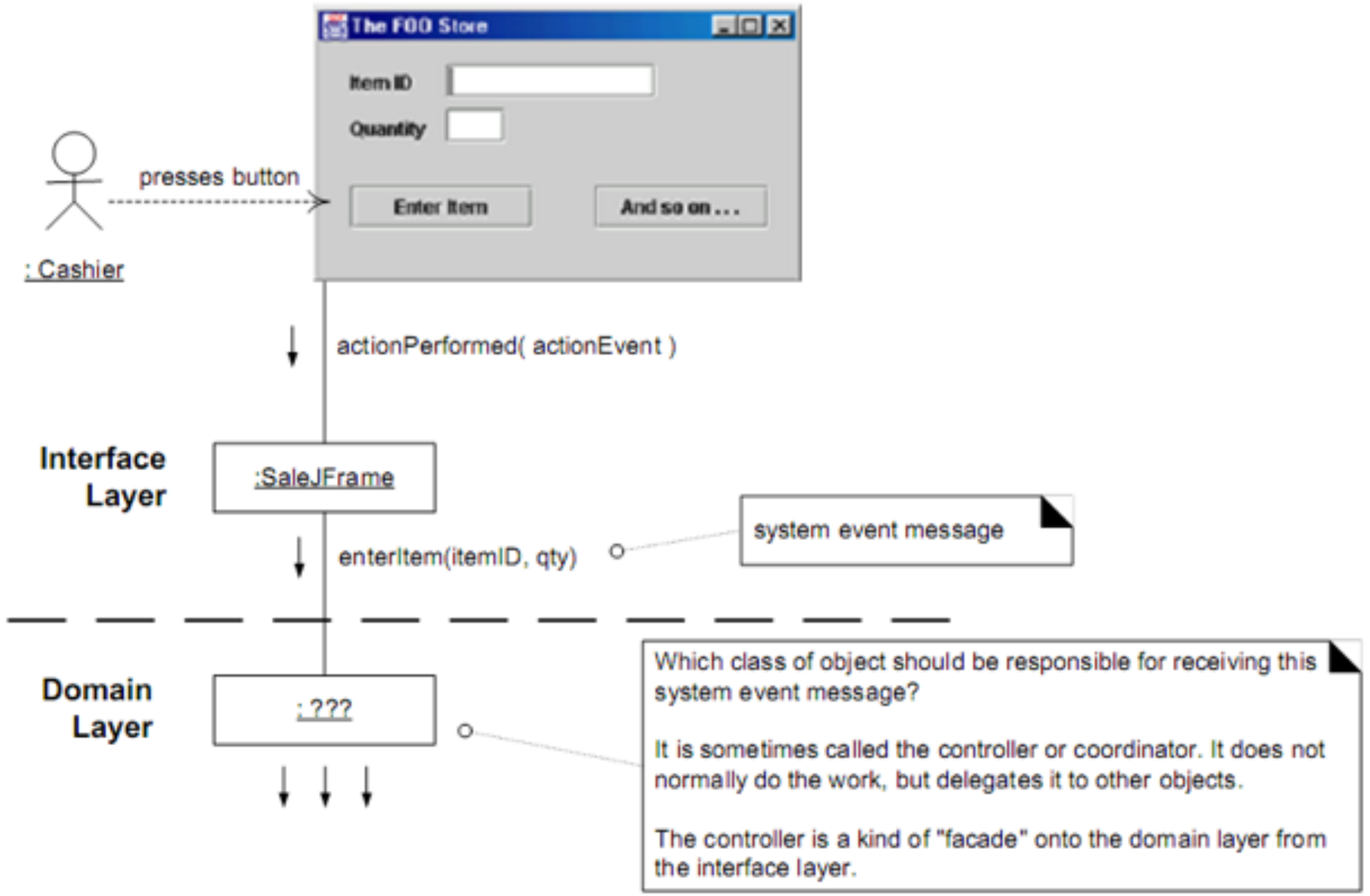
- Scenarios:
  - Very Low Cohesion: A Class is solely responsible for many things in very different functional areas
  - Low Cohesion: A class has sole responsibility for a complex task in one functional area.
  - High Cohesion. A class has moderate responsibilities in one functional area and collaborates with classes to fulfil tasks.
- Advantages:
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility
- Rule of thumb: a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work.

Problem: High Cohesion and Viewpoints



[Harrison&Ossher '93]

Controller: Example



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.

Controller: Example

- By the Controller pattern, here are some choices:
- *Register, POSSystem*: represents the overall "system," device, or subsystem
- *ProcessSaleSession, ProcessSaleHandler*: represents a receiver or handler of all system events of a use case scenario

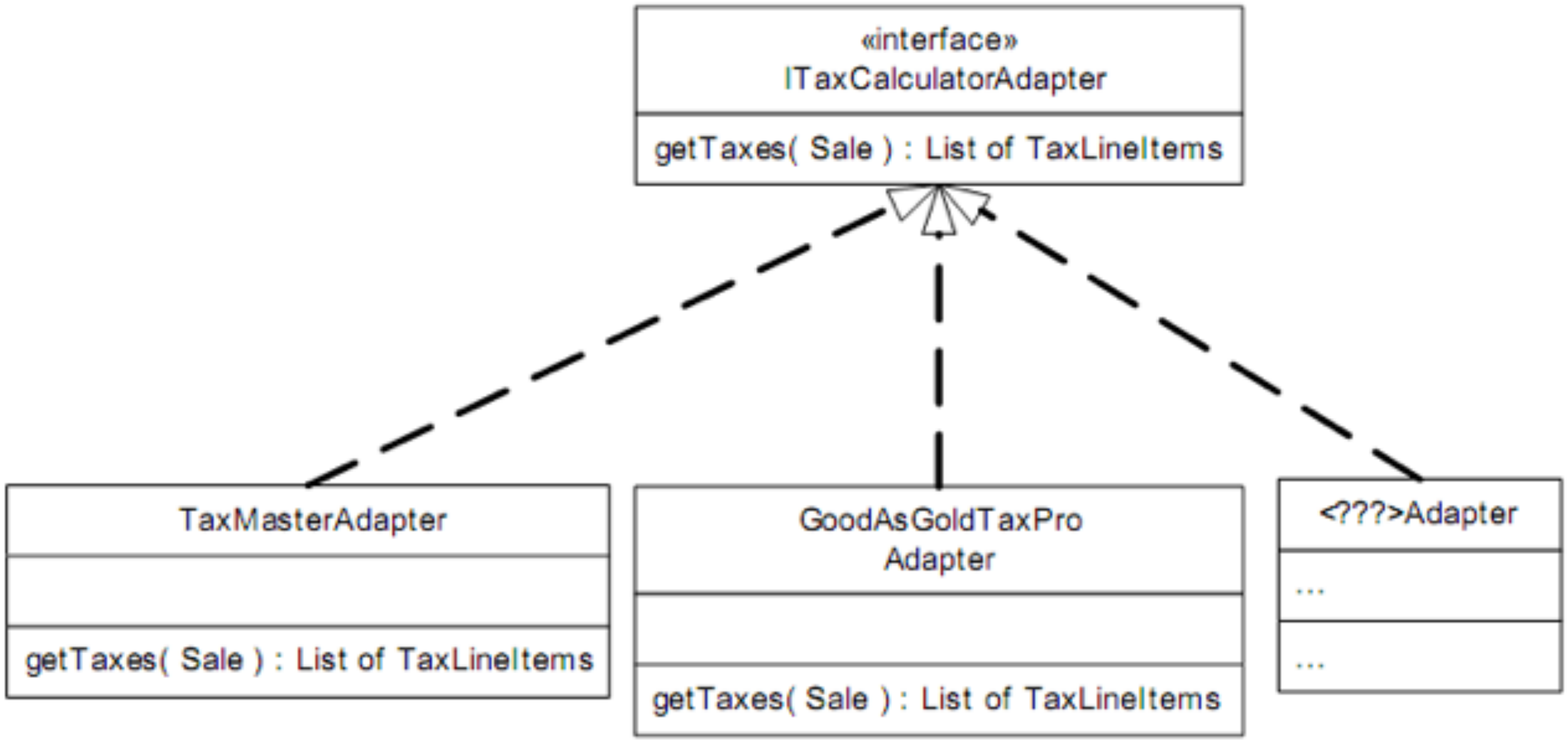
Controller: Discussion

- Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.
- Facade controllers are suitable when there are not "too many" system events
- A use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling
  - typically when the facade controller is becoming "bloated" with excessive responsibilities.

Controller: Discussion

- Benefits
  - Increased potential for reuse, and pluggable interfaces
    - No application logic in the GUI
  - Dedicated place to place state that belongs to some use case
    - E.g. operations must be performed in a specific order
- Avoid bloated controllers!
  - E.g. single controller for the whole system, low cohesion, lots of state in controller
  - Split into use case controllers, if applicable
- Interface layer does not handle system events

Polymorphism: Example


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...

Polymorphism: Discussion

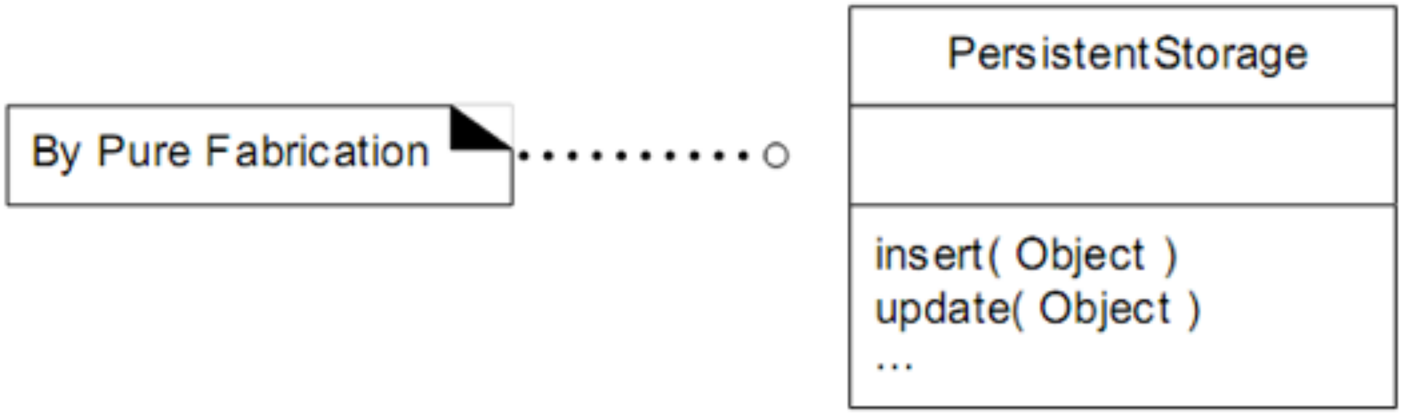
- Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations.
- Properties:
  - Easier and more reliable than using explicit selection logic
  - Easier to add additional behaviors later on
  - Increases the number classes in a design
  - May make the code less easy to follow
- Using the principle excessively for “future-proofing” against yet unknown potential future variations is a bad idea
  - Agile methods recommend to do no significant “upfront design” and add the variation point only when the need arises

Pure Fabrication: Example

- In the point of sale example support is needed to save Sale instances in a relational database.
- By Expert, there is some justification to assign this responsibility to Sale class.
- However, the task requires a relatively large number of supporting database-oriented operations and the Sale class becomes incohesive.
- The sale class has to be coupled to the relational database increasing its coupling.
- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Pure Fabrication : Example

- Solution: create a new class that is solely responsible for saving objects in a persistent storage medium
- This class is a Pure Fabrication



- The Sale remains well-designed, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object

Pure Fabrication: Discussion

- The design of objects can be broadly divided into two groups:
  - Those chosen by representational decomposition (e.g. Sale)
  - Those chosen by behavioral decomposition (e.g. an algorithm object such as TOCGenerator or PersistentStorage)
- Both choices are valid designs, although the second one corresponds less well to the modeling perspective on objects
- If overused, it can lead to a non-idiomatic design, namely a separation of the code into data and behavior as in procedural programming
  - Coupling of data and behavior is central to OO design

Nine GRASP patterns:

- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Indirection

Problem:

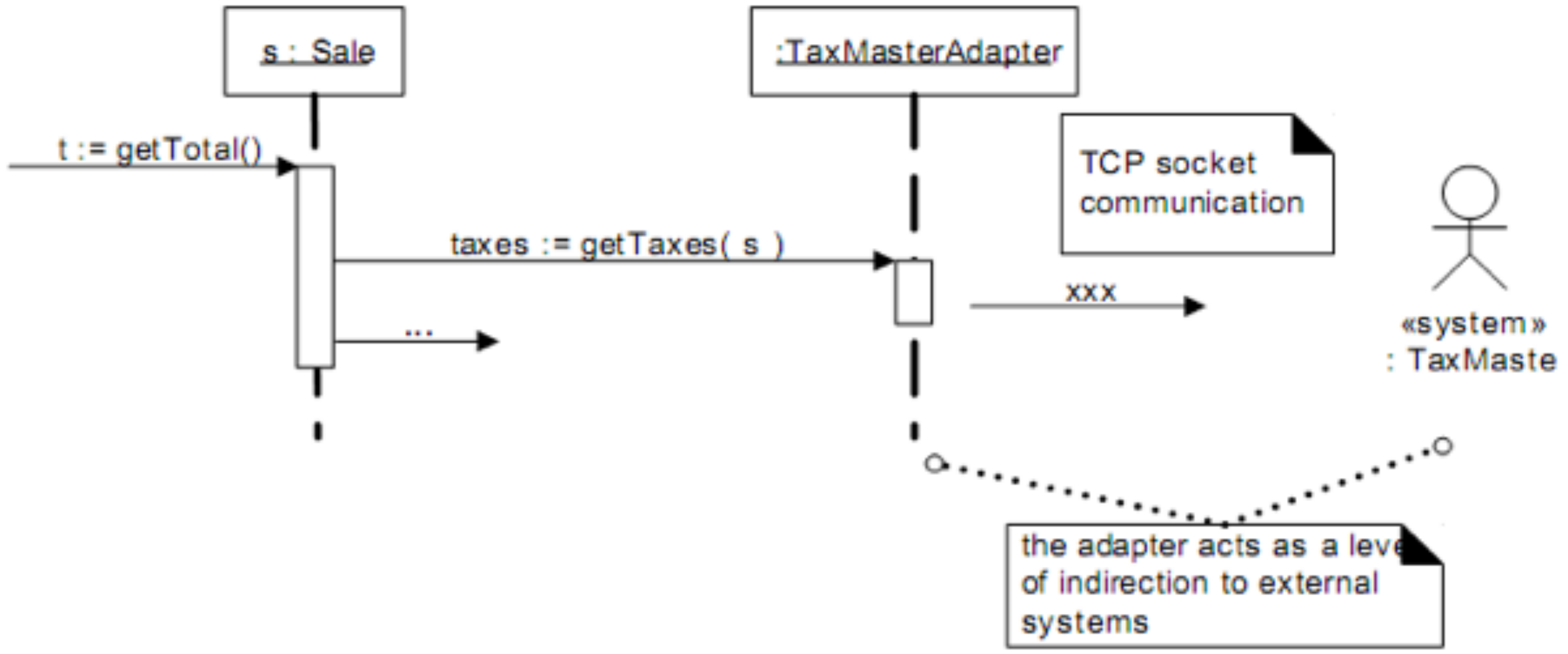
Where to assign a responsibility, to avoid direct coupling between two (or more) things?
How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Solution:

Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

"Most problems in computer science can be solved by another level of indirection"

Indirection: Example



By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces

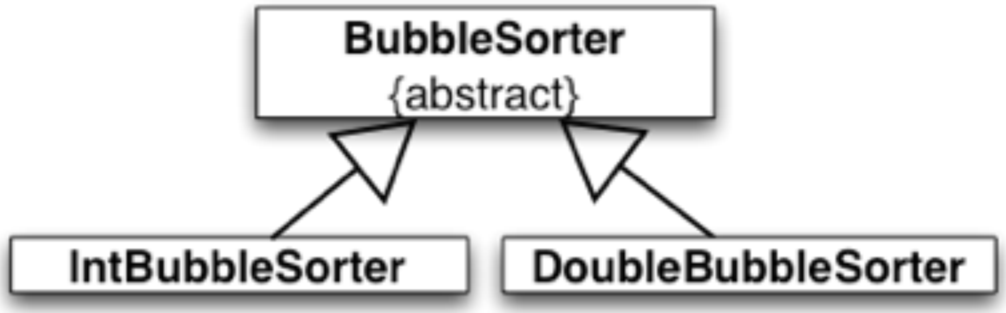
Protected Variation: Examples

- Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV.
- Virtual machines are complex examples of indirection to achieve PV
- Service lookup: Clients are protected from variations in the location of services, using the stable interface of the lookup service.
- Uniform Access Principle
- Law of Demeter
- …

Literature

- Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  - Chapter 16+17+22 introduce GRASP

Using the Template Method Pattern for Bubble-Sort



```
public abstract class BubbleSorter {

    protected int length = 0;

    protected void sort() {
        if (length <= 1) return;
        for (int nextToLast = length - 2; nextToLast >= 0; nextToLast--)
            for (int index = 0; index <= nextToLast; index++)
                if (outOfOrder(index))
                    swap(index);
    }

    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

Policy

Mechanisms

Filling the Template for Specific Sorting Algorithms

```
public class IntBubbleSorter extends BubbleSorter {
  private int[] array = null;

  public void sort(int[] theArray) {
        array = theArray;
        length = array.length;
        super.sort();
  }
  protected void swap(int index) {
        int temp = array[index];
        array[index] = array[index + 1];
        array[index + 1] = temp;
  }
  protected boolean outOfOrder(int index) {
        return array[index] > array[index + 1];
  }
}
```

Mechanisms

Consequences

- Template method forces detailed implementations to extend the template class.
- Detailed implementation depend on the template.
- Cannot re-use detailed implementations‘ functionality. (E.g., swap and out-of-order are generally useful.)
- If we want to re-use the handling of integer arrays with other sorting strategies we must remove the dependency
  - this leads us to the Strategy Pattern.