Objects Analysis Threads Design 15-214

Principles of Software Construction: Objects, Design and Concurrency

Design I (Conceptual Modeling, GRASP)

15-214 toad

Spring 2013

Christian Kästner Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

With slides from Klaus Ostermann

The four course themes

- Threads and Concurrency
  - Concurrency is a crucial system abstraction
  - E.g., background computing while responding to users
  - Concurrency is necessary for performance
  - Multicore processors and distributed computing
  - Our focus: application-level concurrency
  - Cf. functional parallelism (150, 210) and systems concurrency (213)
- Object-oriented programming
  - For flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses
- Analysis and Modeling
  - Practical specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.
- Design
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems

Learning Goals

- What is software design?
- Conceptual Modeling
  - able to model a domain and their relationships
- Design Goals (Modularity, …)
  - able to critique designs, discuss tradeoffs
- Design Considerations with GRASP Principles
  - justify designs
  - toolkit to perform design decisions

Goal of Software Design

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

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];
   }
   …
}
```

Sorting with configurable order, variant B

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  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")

Tradeoffs

```
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];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
    …
  boolean mustswap;
  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; }}
```

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
  - Testability
  - => **modularity**
- …as opposed to external quality
  - Correctness: Valid implementation of requirements
  - Ease of Use
  - Resource consumption
  - Legal issues, political issues, …

The bad news

it depends

(see context)

depends on what?
what are scenarios?
what are tradeoffs?

"**Software engineering** is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind.
Software engineering entails making **decisions** under constraints of limited time, knowledge, and resources. […]

Engineering quality resides in engineering judgment. […]

Quality of the software product depends on the engineer's faithfulness to the engineered artifact. […]

Engineering requires reconciling conflicting constraints. […]

Engineering skills improve as a result of careful systematic reflection on experience. […]

Costs and time constraints matter, not just capability. […]

Software Engineering for the 21st Century: A basis for rethinking the curriculum
Manifesto, CMU-ISRI-05-108

Conceptual Design

Conceptual Modeling / Domain Models

- Find the concepts in the problem domain
  - Real-world abstractions, not necessarily software objects
- Establish a common vocabulary
- Common documentation, big pictutre
- For communication!
- Often using UML class diagrams as (informal) notation
- Starting point for finding classes

Running Example

Running Example

- **Point of sale** (**POS**) or **checkout** is the place where a retail transaction is completed. It is the point at which a customer makes a payment to a merchant in exchange for goods or services. At the point of sale the merchant would use any of a range of possible methods to calculate the amount owing - such as a manual system, weighing machines, scanners or an electronic cash register. The merchant will usually provide hardware and options for use by the customer to make payment. The merchant will also normally issue a receipt for the transaction.
- For small and medium-sized retailers, the POS will be customized by retail industry as different industries have different needs. For example, a grocery or candy store will need a scale at the point of sale, while bars and restaurants will need to customize the item sold when a customer has a special meal or drink request. The modern point of sale will also include advanced functionalities to cater to different verticals, such as inventory, CRM, financials, warehousing, and so on, all built into the POS software. Prior to the modern POS, all of these functions were done independently and required the manual re-keying of information, which resulted in a lot of errors.

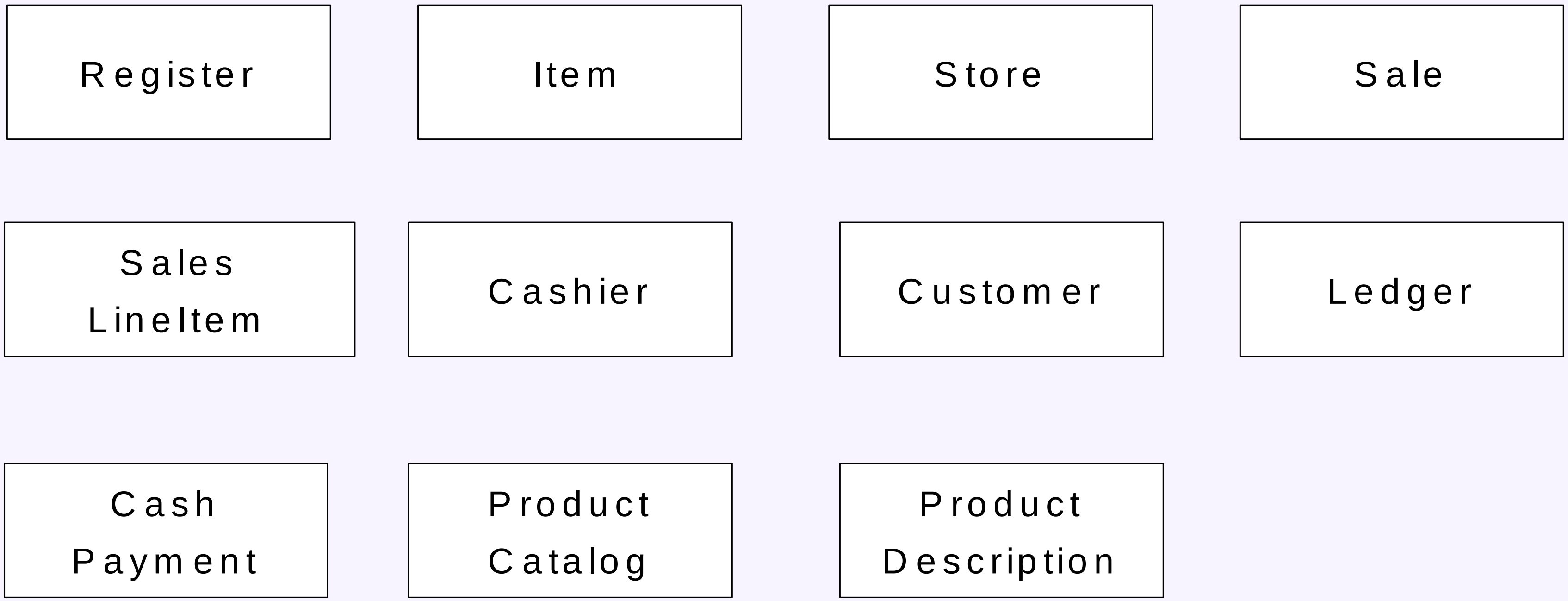
http://en.wikipedia.org/wiki/Point_of_sale

Read description carefully, look for nouns and verbs

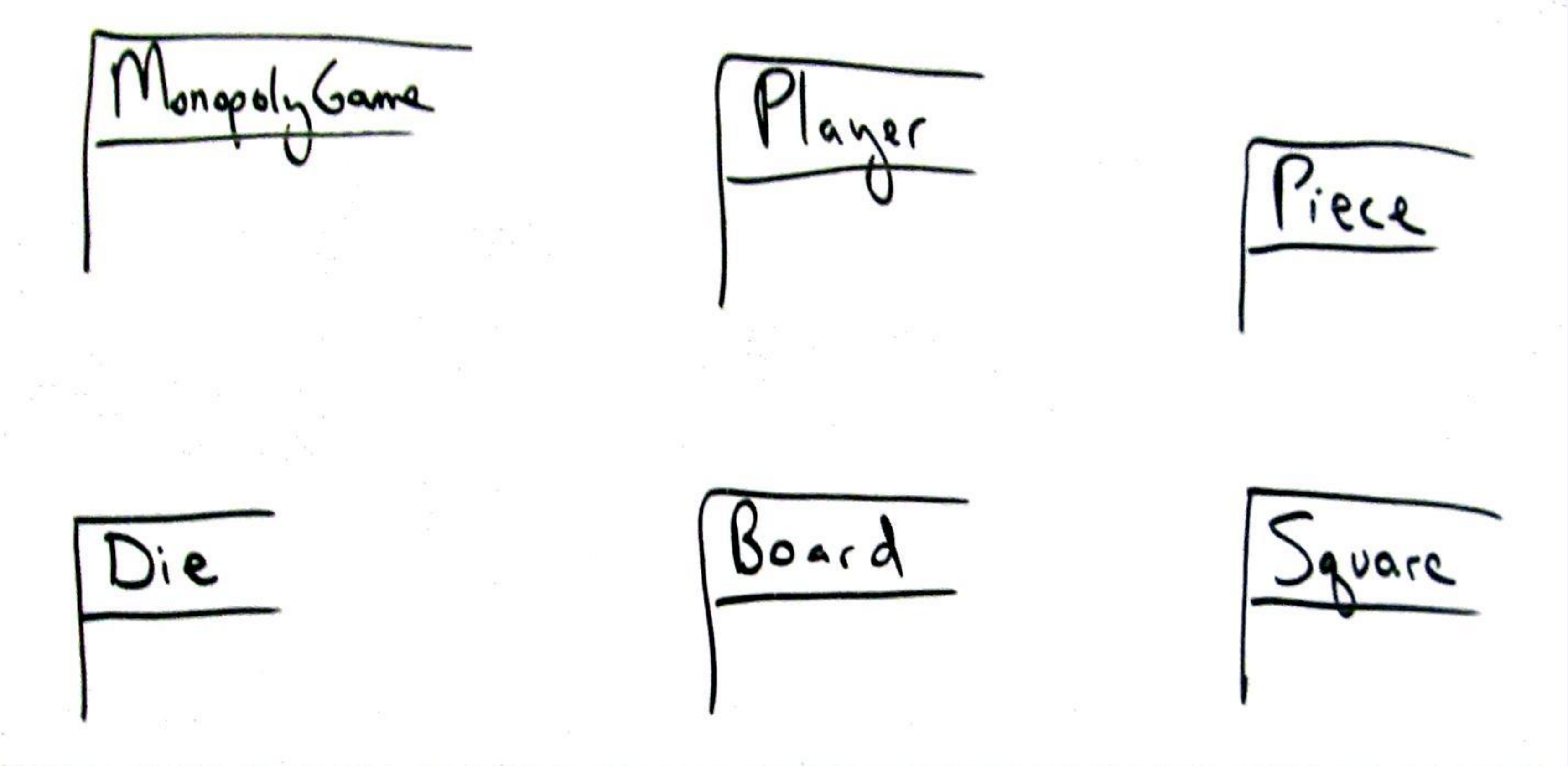
- **Point of sale** (**POS**) or **checkout** is the place where a retail transaction is *completed*. It is the point at which a customer makes a payment to a merchant in *exchange* for goods or services. At the point of sale the merchant would use any of a range of possible methods to *calculate* the amount owing - such as a manual system, weighing machines, scanners or an electronic cash register. The merchant will usually provide hardware and options for use by the customer to *make* payment. The merchant will also normally *issue* a receipt for the transaction.
- For small and medium-sized retailers, the POS will be customized by retail industry as different industries have different needs. For example, a grocery or candy store will need a scale at the point of sale, while bars and restaurants will need to *customize* the item sold when a customer has a special meal or drink request. The modern point of sale will also include advanced functionalities to cater to different verticals, such as inventory, CRM, financials, warehousing, and so on, all built into the POS software. Prior to the modern POS, all of these functions were done independently and required the manual re-keying of information, which resulted in a lot of errors.

http://en.wikipedia.org/wiki/Point_of_sale

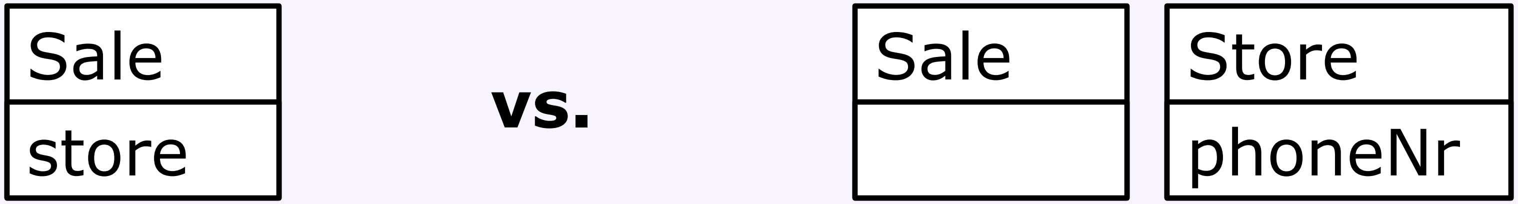
First Steps toward a Domain Model



Domain Model

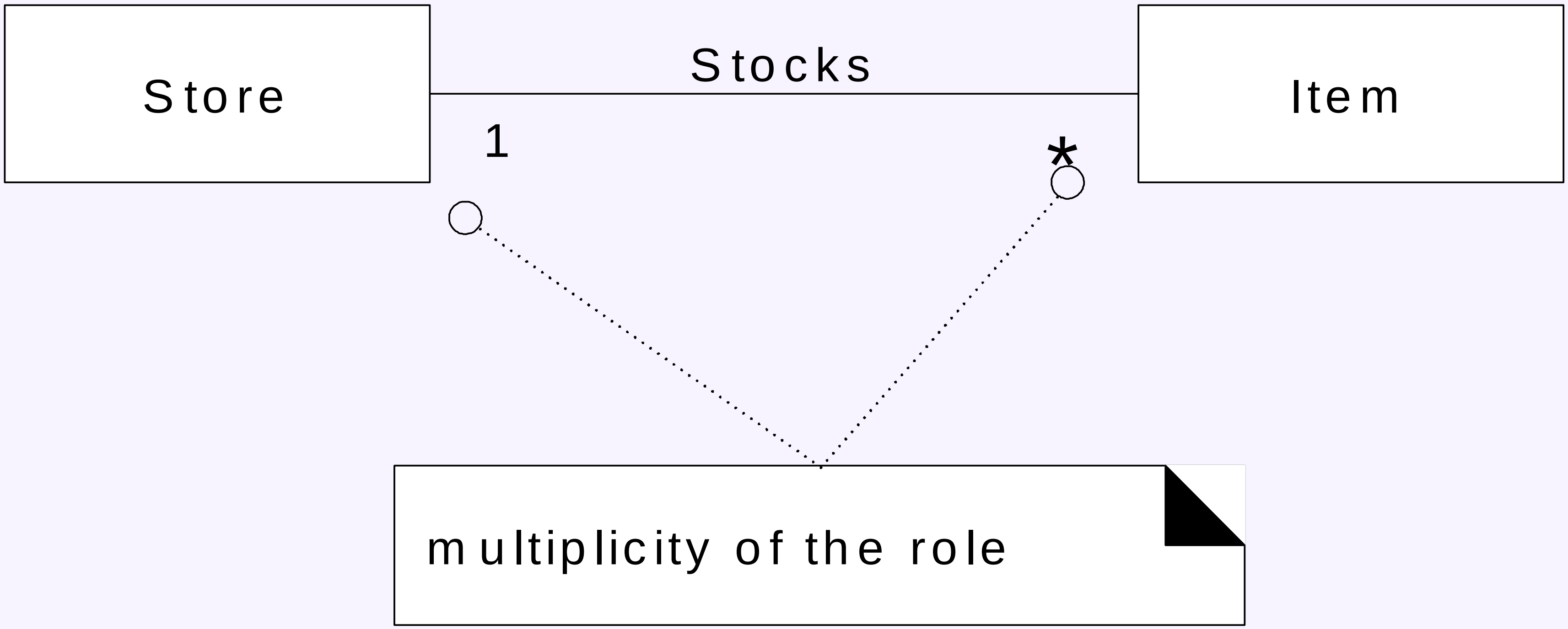


Classes vs. Attributes



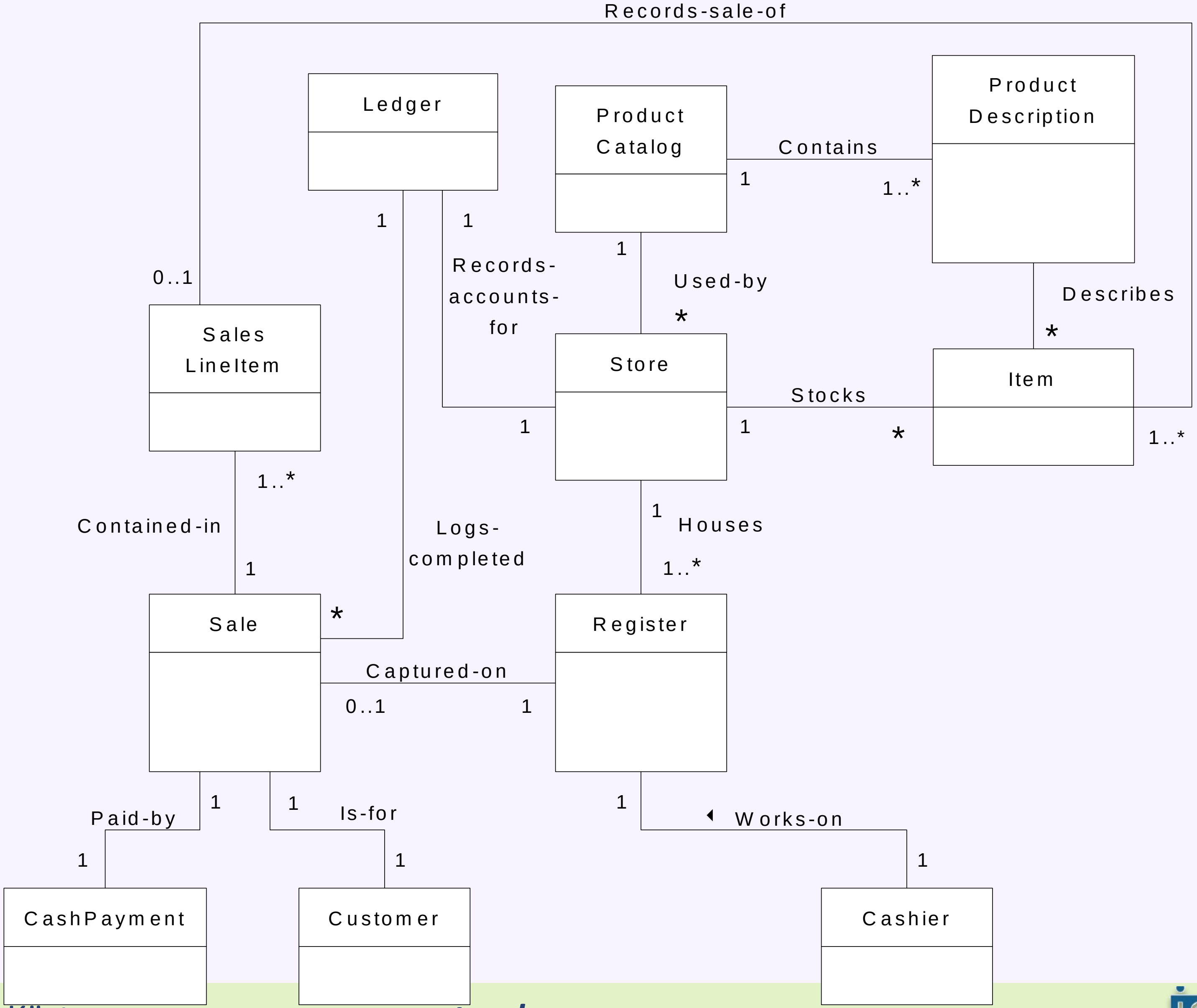
- "If we do not think of some conceptual class X as text or a number in the real world, it's probably a conceptual class, not an attribute"

Associations



- When do we care about a relationship between two objects? (in the real world)

Domain Model (excerpt)



Note

- Focus on concepts, not software classes, not data
  - ideas, things, objects
  - Give it a name, define it and give examples (symbol, intension, extension)
  - Add glossary
  - Some might be implemented as classes, other might not
- There are many choices
- Agree on a single vocabulary
- The model will never be perfectly **correct**
  - that’s okay
  - start with a partial model, model what's needed
  - extend with additional information later
  - communicate changes clearly
  - otherwise danger of "analysis paralysis"

Three perspectives of class diagrams

- Conceptual: Draw a diagram that represents the concepts in the domain under study
  - Little or no regard for the software that might implement it
- Specification: Describing the interfaces of the software, not the implementation
  - Often confused in OO since classes combine both interfaces and implementation
- Implementation: Diagram describes actual implementation classes

Understanding the intended perspective is crucial to drawing and reading class diagrams, even though the lines between them are not sharp

Goals for Object Design

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

Modularity

- A software construction method is modular if it helps designers to produce software systems made of autonomous elements connected by a coherent, simple structure

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

Five Criteria: Modular Decomposability

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

- POS Example:
  - Data Model
  - User Interface
  - Printing Receipts
  - Tax Accounting
  - Admin Interface
  - Connecting Scales…
- Modular Decomposability implies: Division of Labor possible!

Five Criteria: Modular Composability

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

- Is dual to modular decomposability
- Is directly connected with reusability
- Example 1: Libraries have been reused successfully in countless domains
- Example 2: Unix Shell Commands
- POS: Examples:
  - Reuse existing Storage Management System
  - Connect to CRM System
- Counter-Example: Preprocessors

Five Criteria: Modular Understandability

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

- Important for maintenance
- Applies to all software artifacts, not just code
- Counter-example: Sequential dependencies between modules

Five Criteria: Modular Continuity

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

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

Five Criteria: Modular Protection

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

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

Five Rules: Direct Mapping / Low Representational Gap

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.

Follows from continuity and decomposability

Low Representational Gap

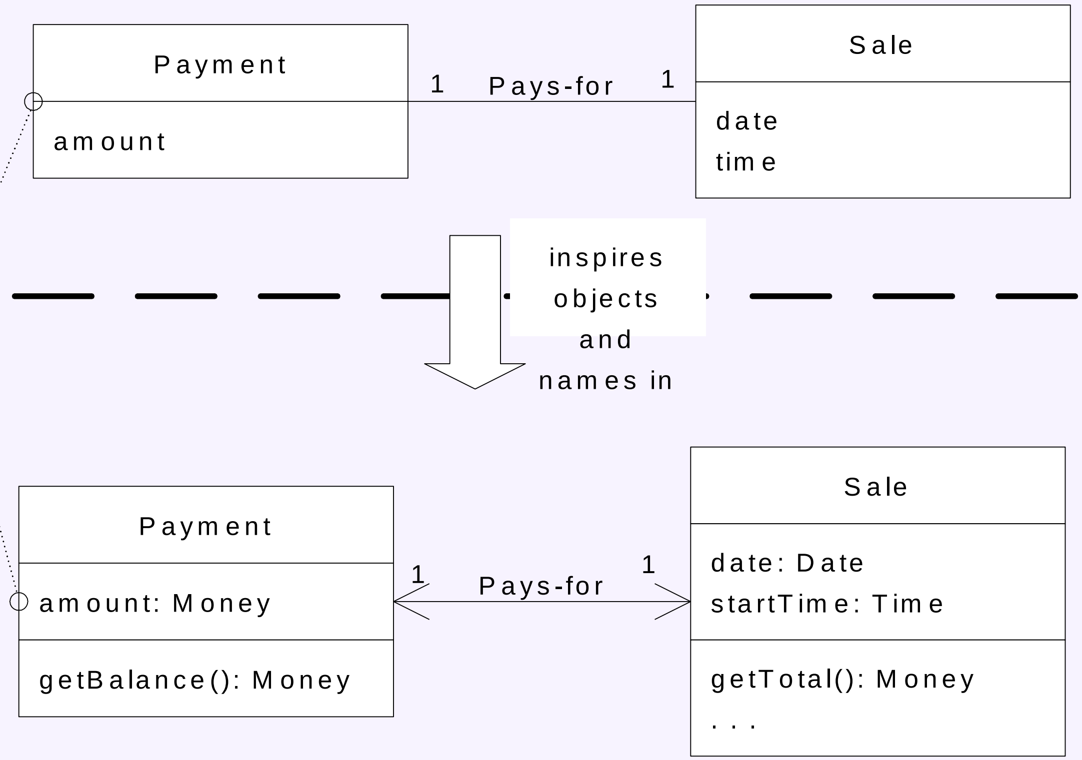
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.



UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

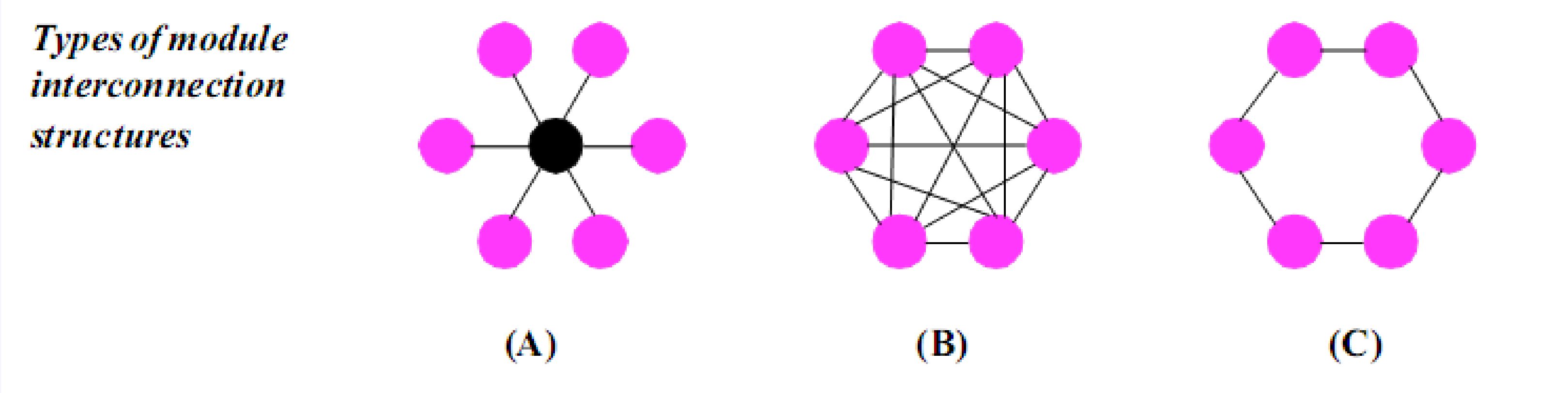
Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Five Rules: Few Interfaces

Every module should communicate with as few others as possible

institute for SOFTWARE RESEARCH

Five Rules: Few Interfaces



- 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
- Counter-Example: Big Interfaces ☺

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

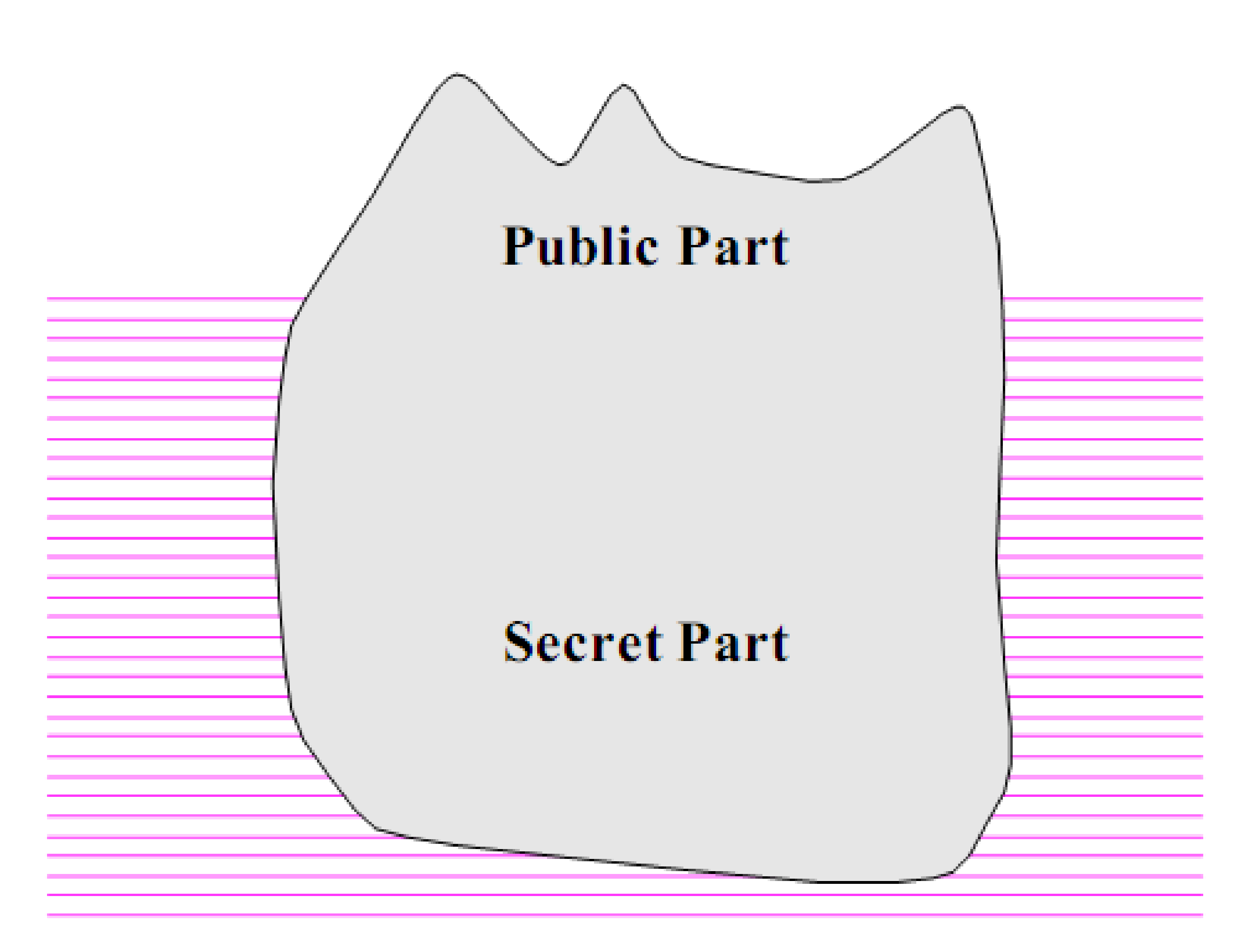
Intermezzo: Law of Demeter (LoD)

- LoD (or Principle of Least Knowledge): 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 subset 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



Monopoly Example

```
class Player {
        Board board;
        Square getSquare(String name) {
                for (Square s: board.getSquares())
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

```
class Player {
        Board board;
        Square getSquare(String n) { board.getSquare(n); }
}
class Board{
        List<Square> squares;
        Square getSquare(String name) {
                for (Square s: squares)
                        if (s.getName().equals(name))
                                return s;
                return null;
}}
```

Summary Design Goals

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection
- Direct Mapping / Low Representational Gap
- Few Interfaces
- Small Interfaces
- Explicit Interfaces
- Information Hiding

Grasp Patterns

GRASP Patterns / Principles

- 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
- Describe fundamental principles of object design and responsibility
- General principles, may be overruled by others

Patterns/Principles aid Communication

Fred: "Where do you think we should place the responsibility for creating a SalesLineItem? I think a Factory."

Wilma: "By Creator, I think Sale will be suitable."

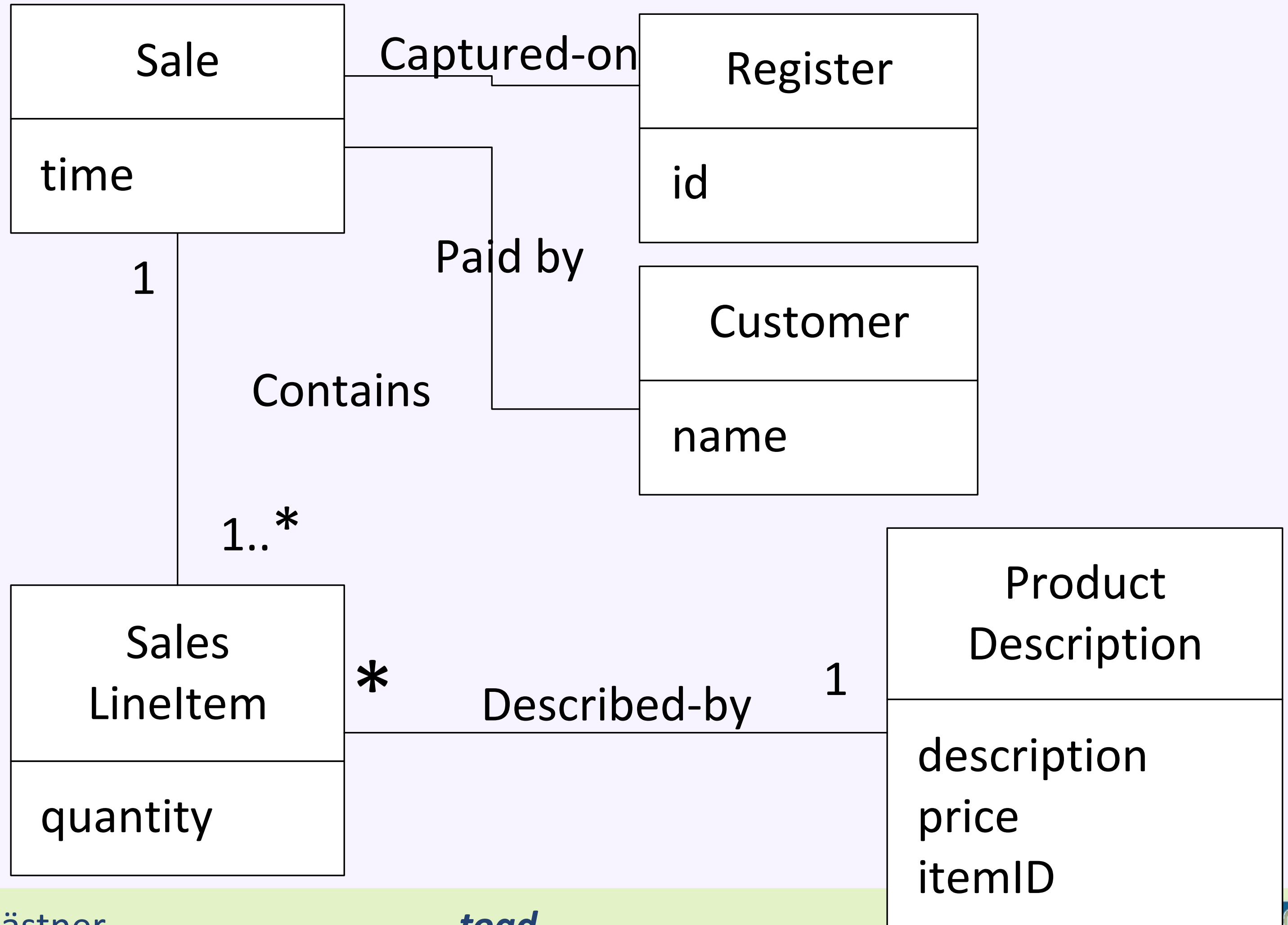
Fred: "Oh, right - I agree."

Nine GRASP Principles:

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

Information Expert Principle

- Who should be responsible for **knowing** the grand total of a sale?

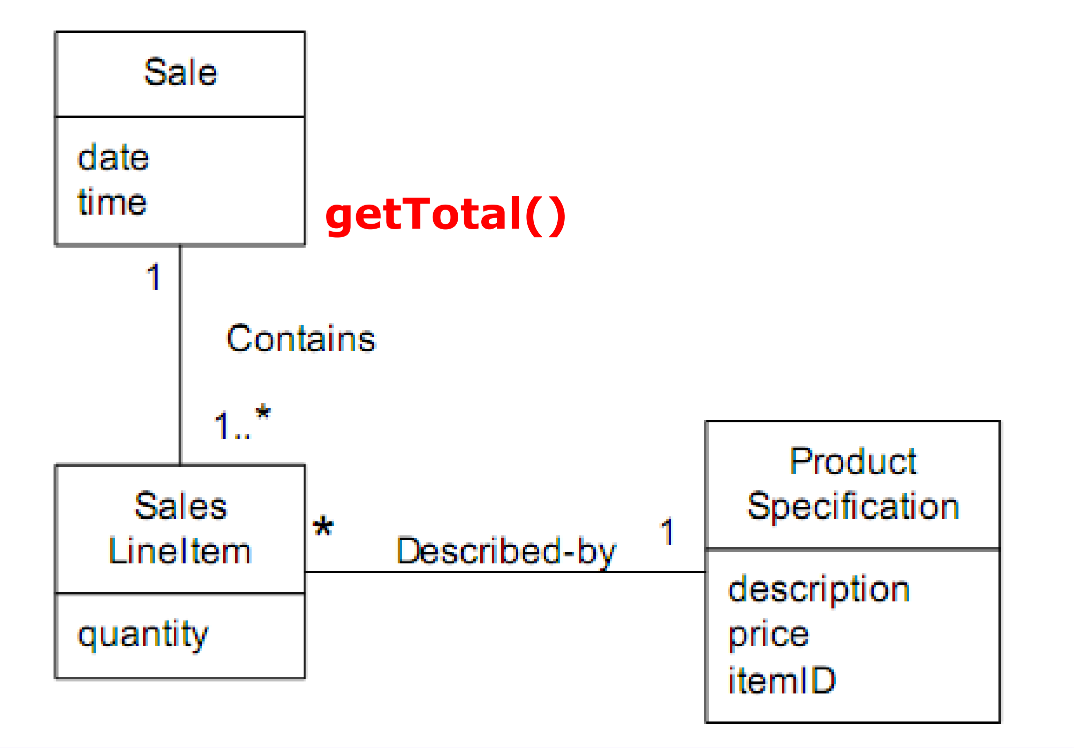


Information Expert

- 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!
- Typically follows common intuition
- Design Classes (Software Classes) instead of Conceptual Classes
  - If Design Classes do not yet exist, look in Domain Model for fitting abstractions (-> low representational gap)

Information Expert

- What information is needed to determine the grand total?
  - Line items and the sum of their subtotals
- *Sale* is the information expert for this responsibility.

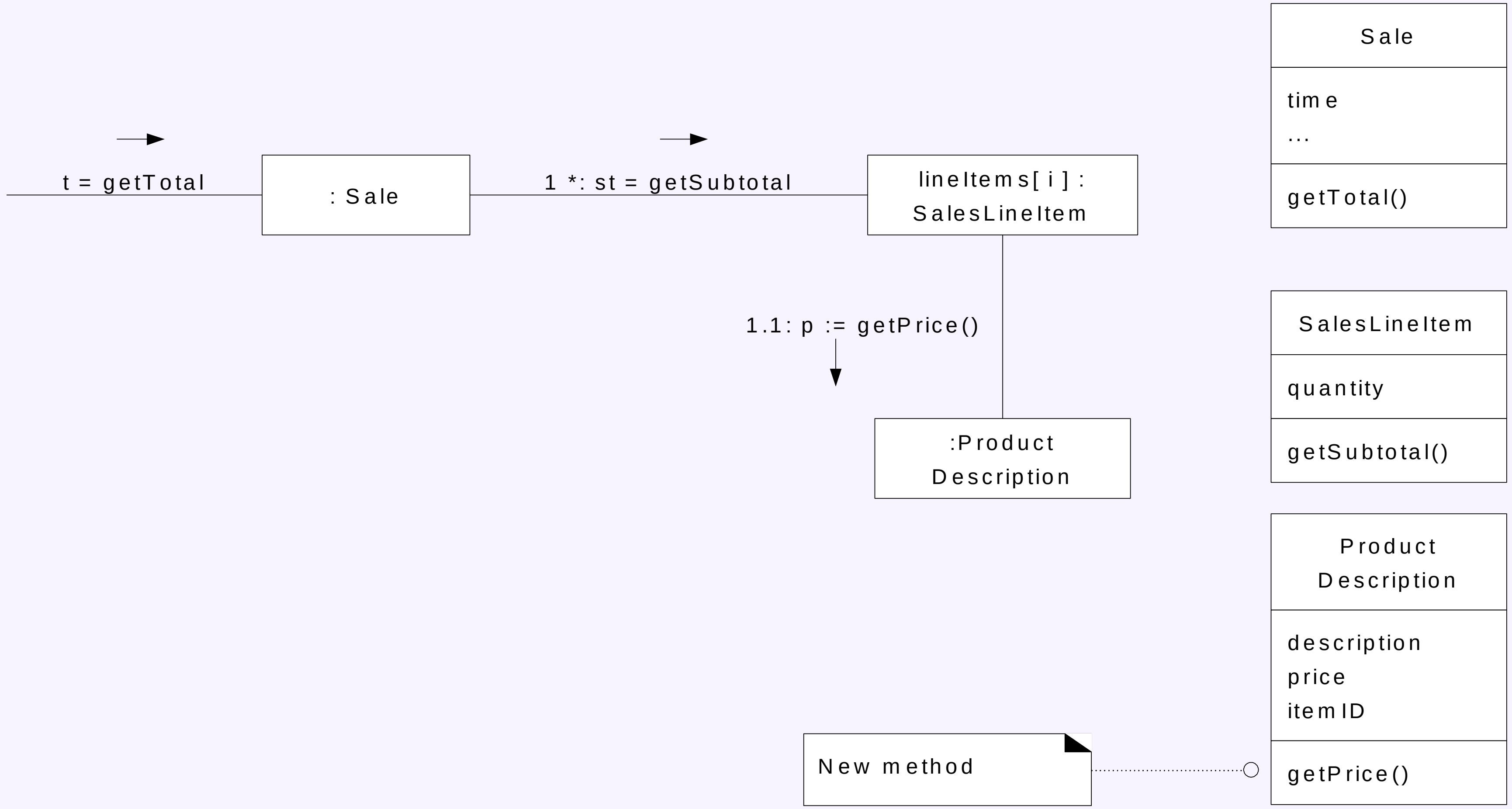


Information Expert

- 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



Information Expert -> "Do It Myself Strategy"

- 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

Reflection on Design Goals

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection
- Direct Mapping / Low Representational Gap
- Few Interfaces
- Small Interfaces
- Explicit Interfaces
- Information Hiding

Creator Principle: Problem

- Who creates Nodes in a Graph?
- Who creates instances of SalesItem?
- Who creates Rabbit-Actors in a Game?
- Who creates Tiles in a Monopoly game?
  - AI? Player? Main class? Board? Meeple (Dog)?

Creator: Problem

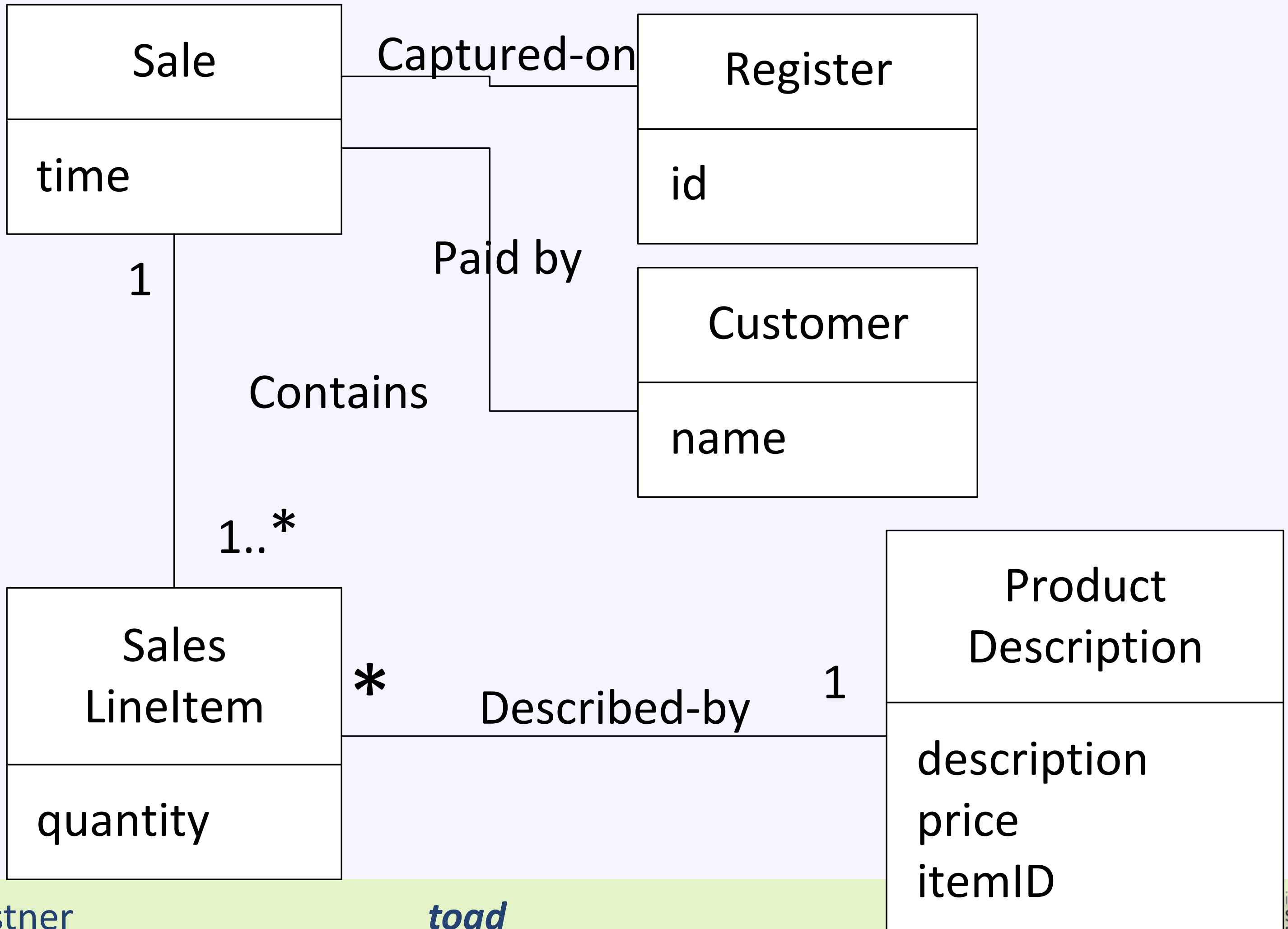
- Who creates Tiles in a Monopoly game?
  - Typical Answer: The board
  - Container creates things contained

Creator Principle

- 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
- Key idea: Creator needs to keep reference anyway and will frequently use the created object

Creator : Example

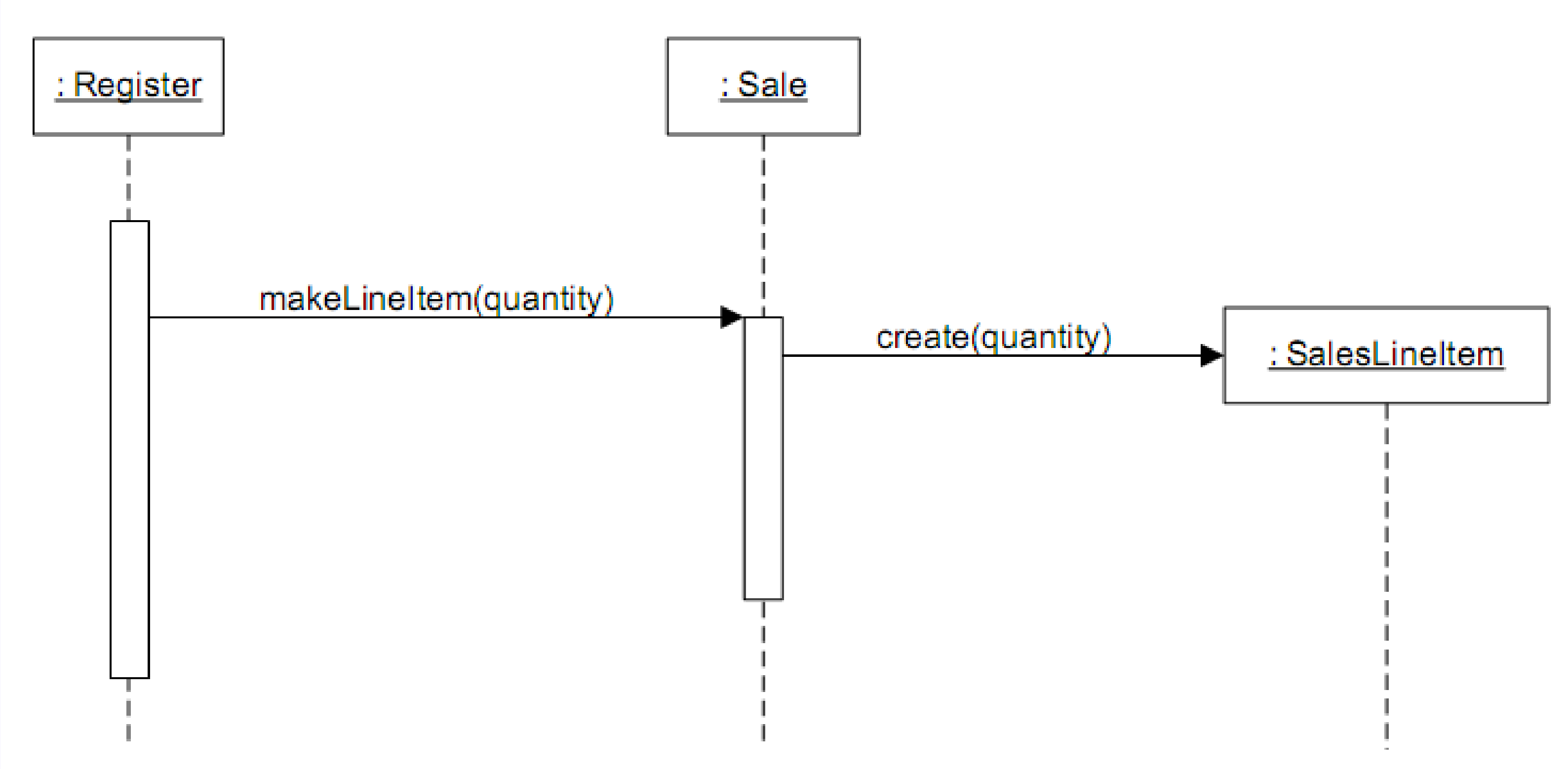
- Who is responsible for creating SalesLineItem objects?



Creator : Example

- Creator pattern suggests Sale.

- Collaboration diagram is



Creator: Discussion

- 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
- 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”)

Reflection on Design Goals

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection
- Direct Mapping / Low Representational Gap
- Few Interfaces
- Small Interfaces
- Explicit Interfaces
- Information Hiding

Low Coupling Principle

Problem:

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

Solution:

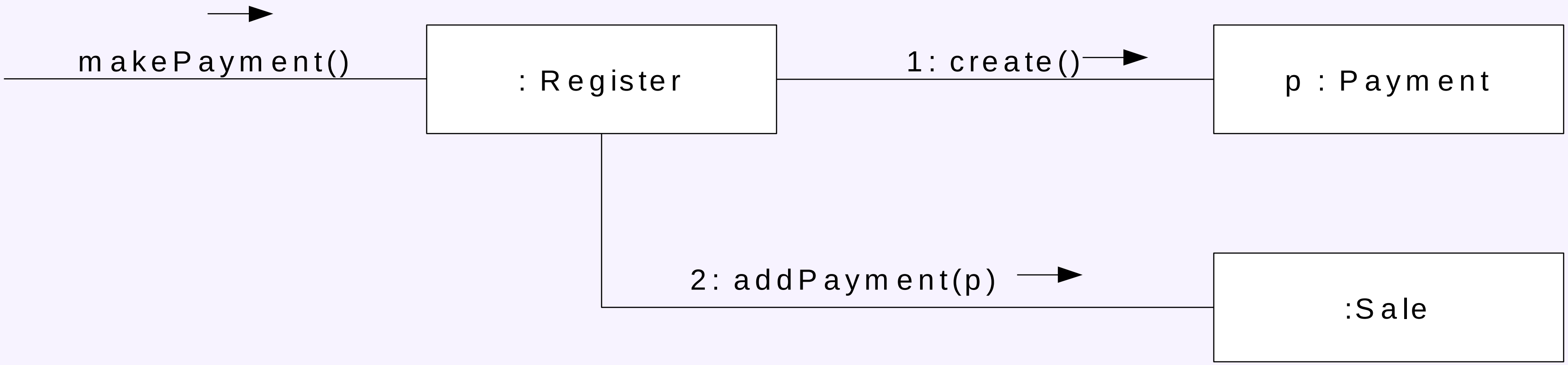
Assign a responsibility so that coupling remains low.

Use this principle when evaluating alternatives

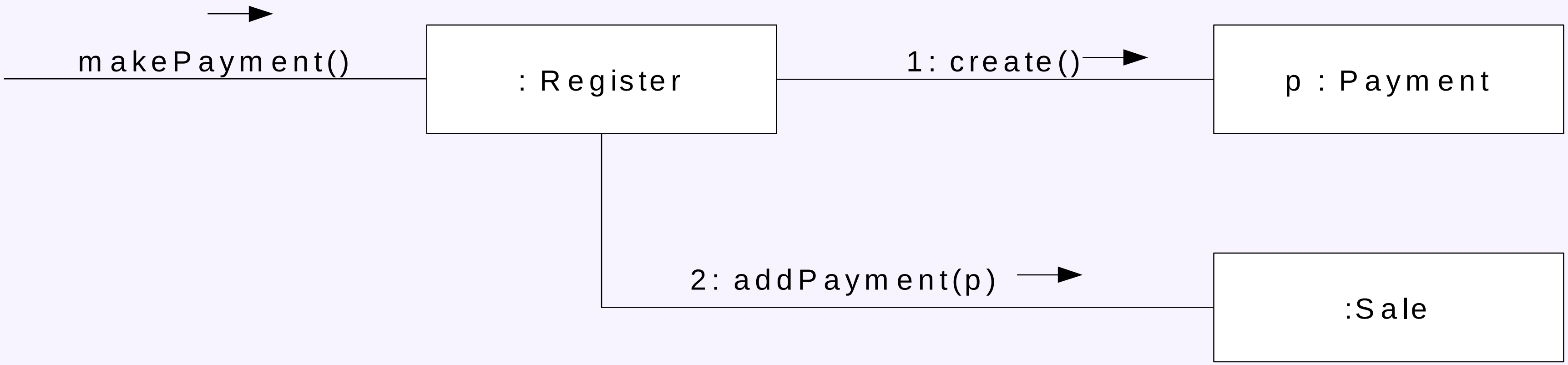
Example

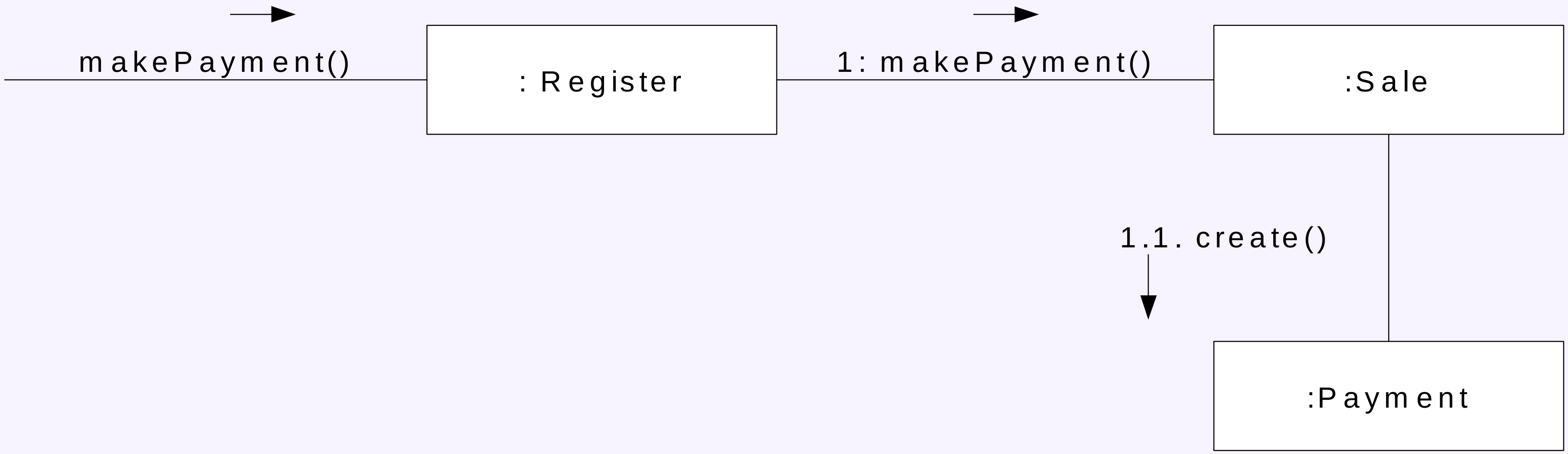
- Create a Payment and associate it with the Sale.

Example

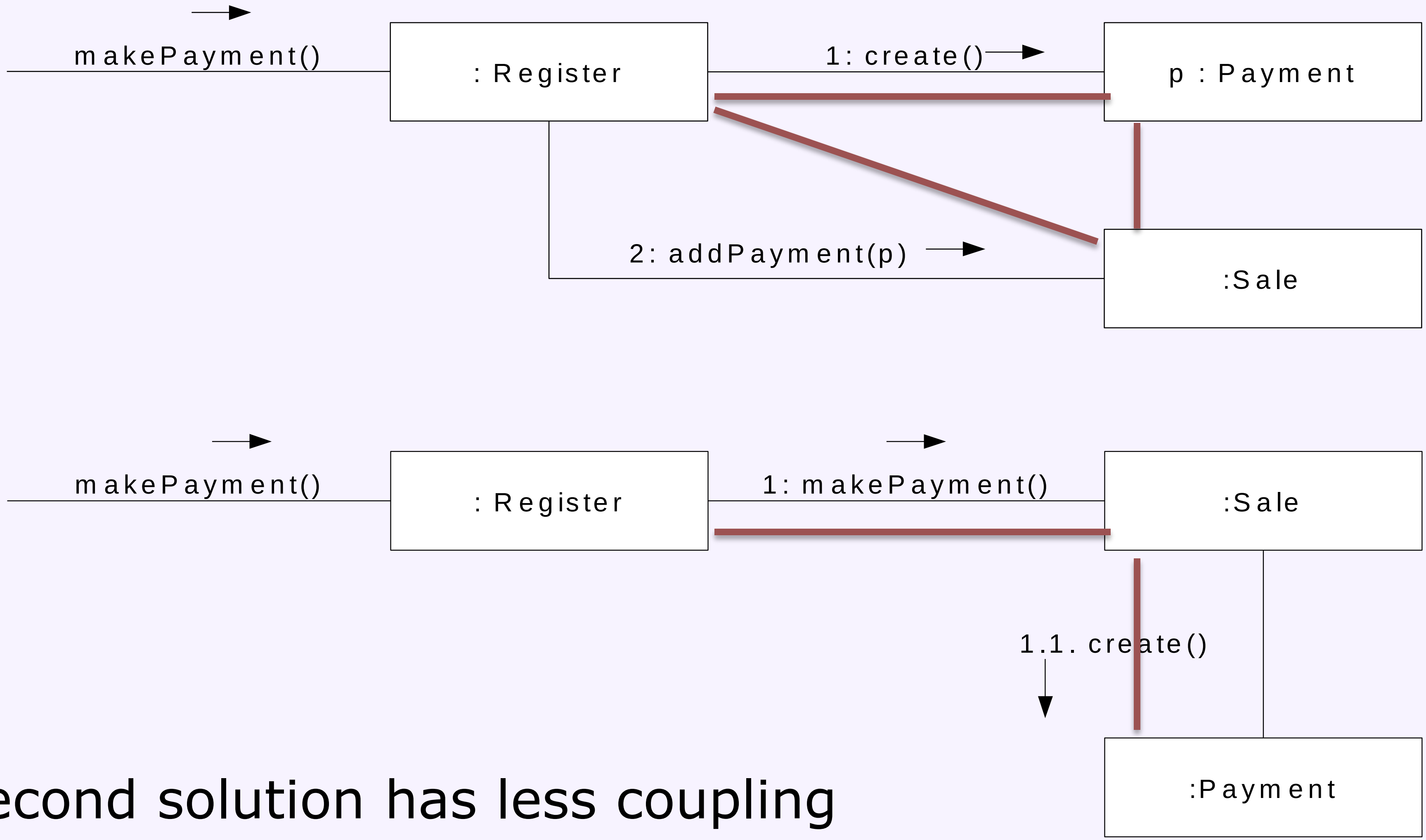

makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale

Example


makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale


makePayment()
: Register
1: makePayment()
:Sale
1.1. create()
:Payment

Coupling



Second solution has less coupling
Register does not know about Payment class

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

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
  - -> Prefer composition over inheritance
- 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

Reflection on Design Goals

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection
- Direct Mapping / Low Representational Gap
- Few Interfaces
- Small Interfaces
- Explicit Interfaces
- Information Hiding

High Cohesion Principle

Problem:

How to keep complexity manageable.

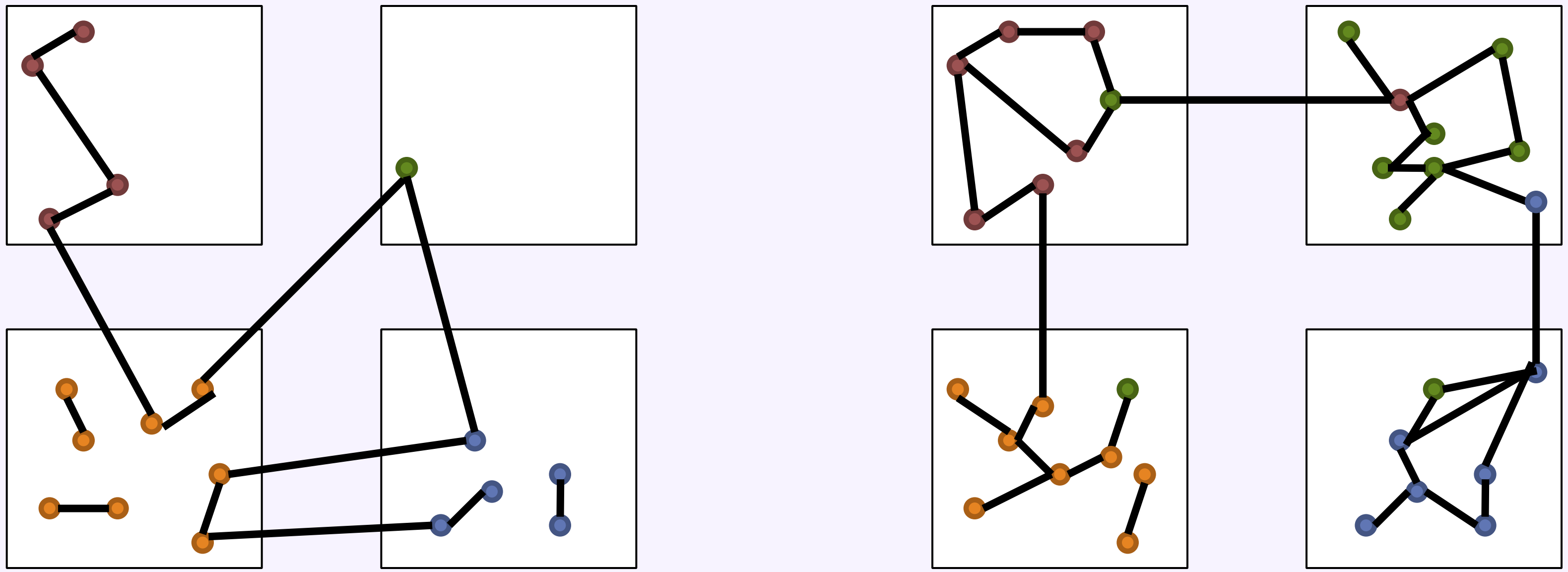
Solution:

Assign responsibilities so that cohesion remains high.

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

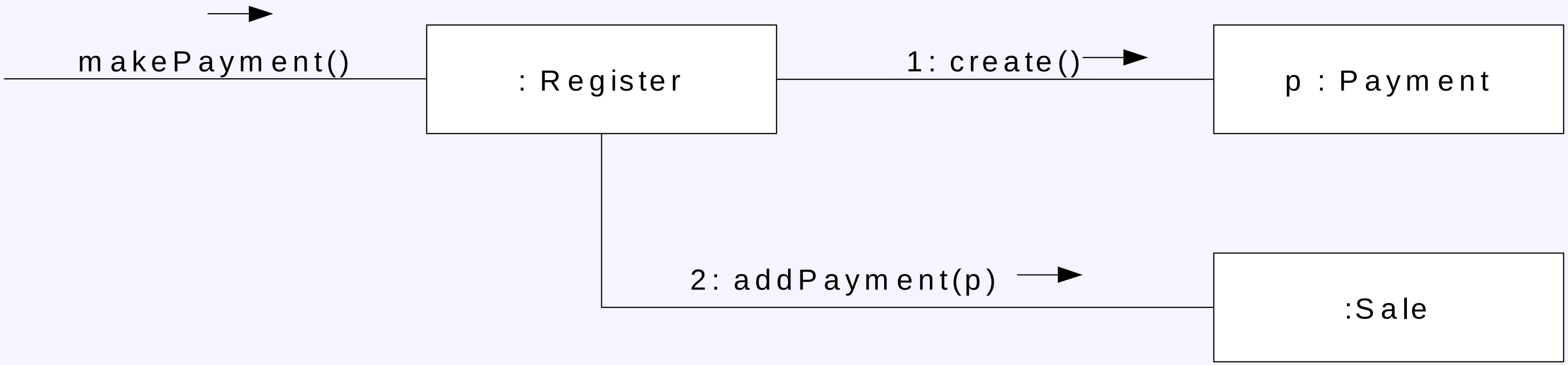
An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion

High cohesion



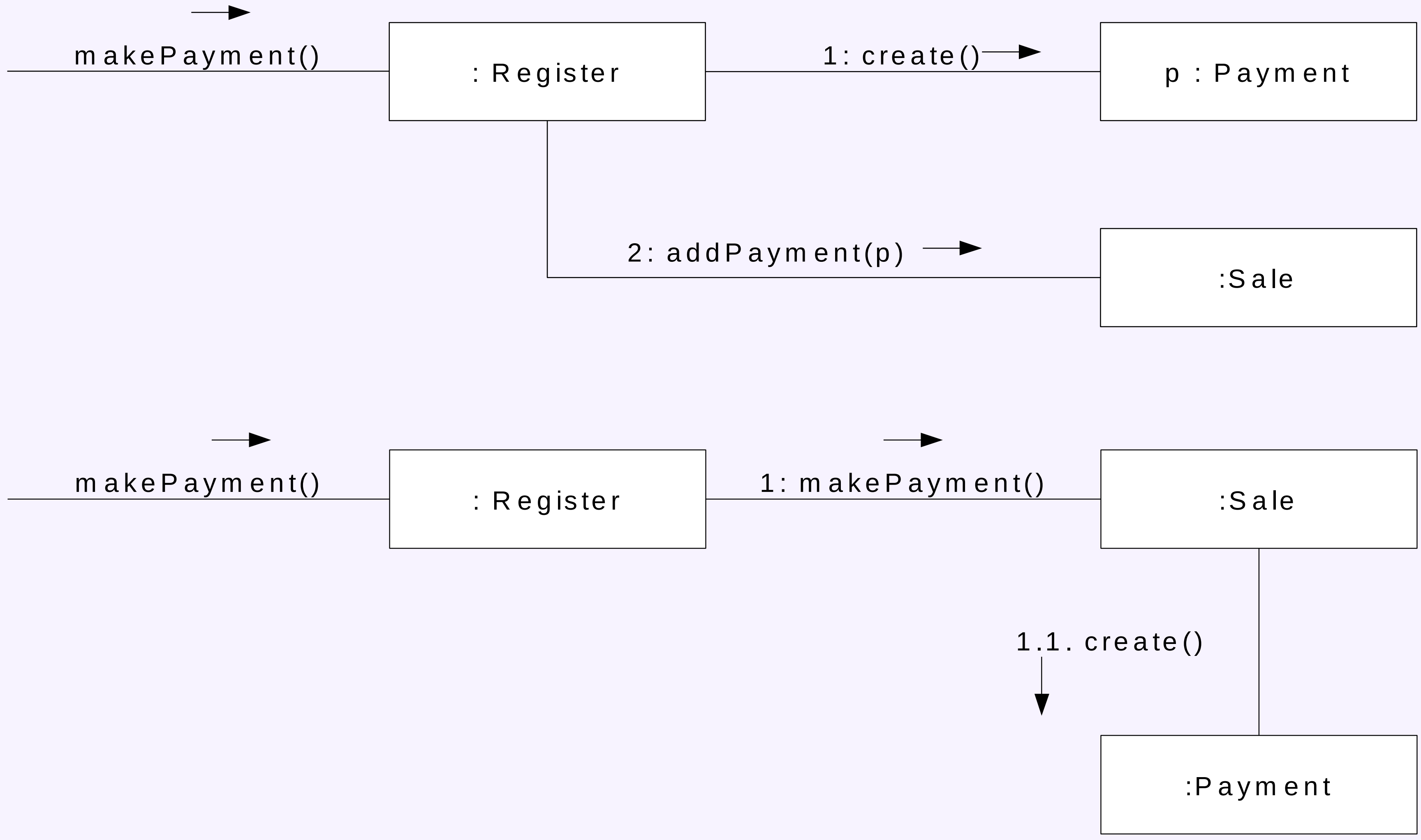
- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility

Example



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

Example


makePayment()
: Register
1: create()
p : Payment
2: addPayment(p)
:Sale
makePayment()
: Register
1: makePayment()
:Sale
1.1. create()
:Payment

Extra: isVisited

```
class Graph {
        Node[] nodes;
        boolean[] isVisited;
}
class Algorithm {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; …)
                        if (!g.isVisited[i]) {
                                …
                                g.isVisited[i] = true;
                        }
                }
                return v;
        }
}
```

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.

Controller Principle

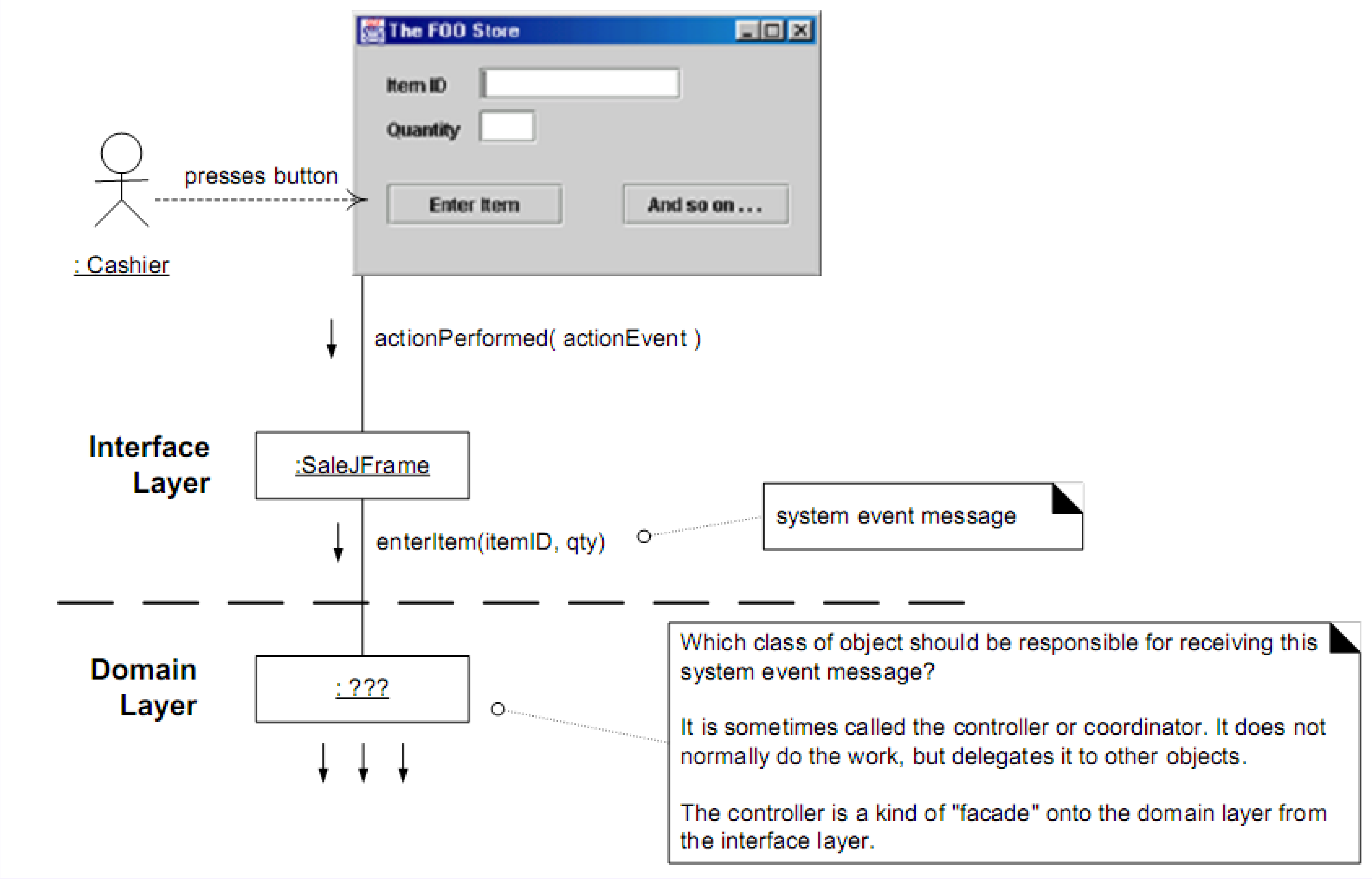
Problem:

Who should be responsible for handling an input system event?

Solution:

Assign the responsibility for receiving or handling a system event message to a class representing the overall system, device, or subsystem (facade controller) or a use case scenario within which the system event occurs (use case controller)

Controller: Example



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

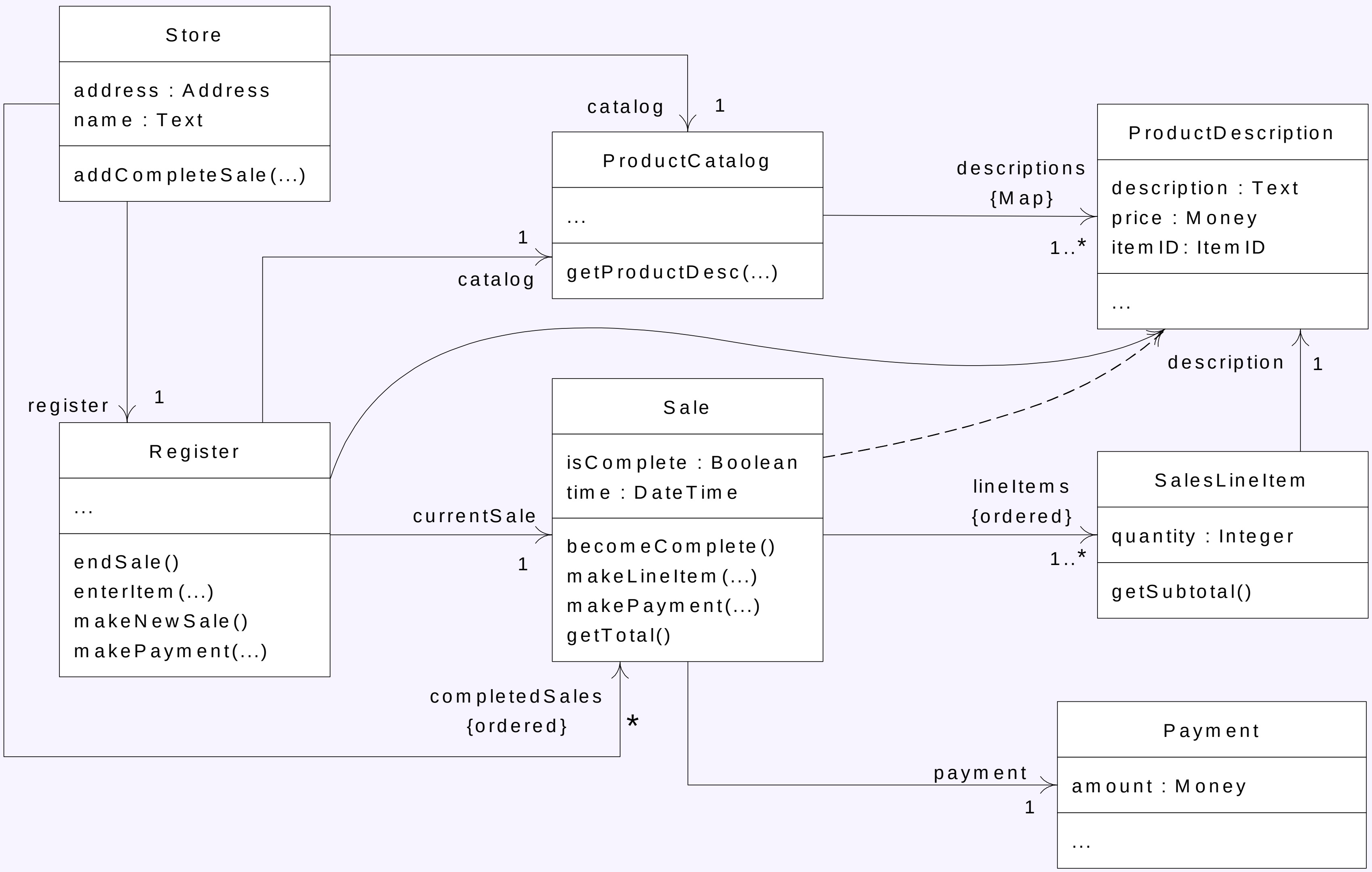
Controller: Discussion

- 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.

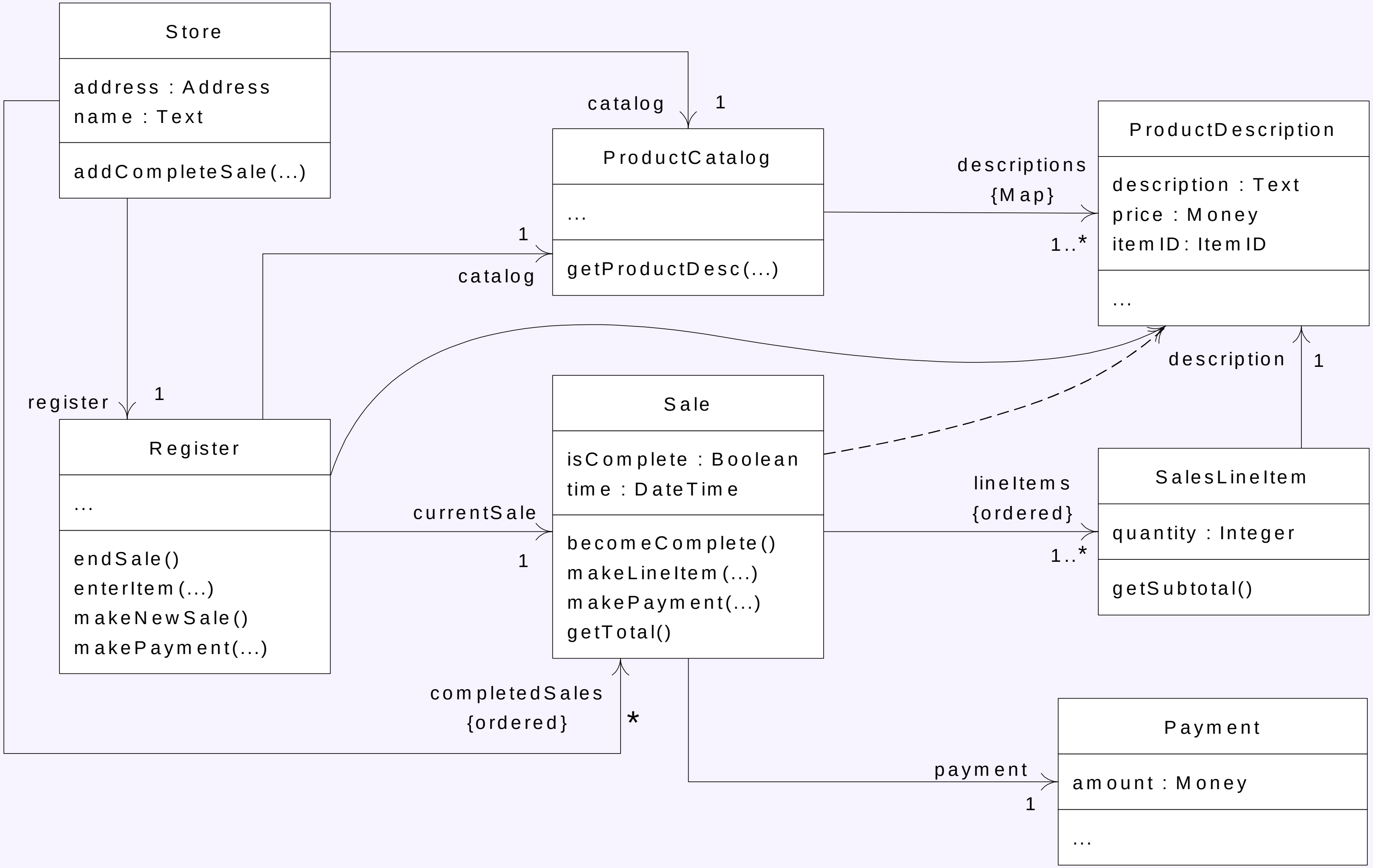
Controller: Discussion

- 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

Resulting Design Model (example, excerpt)

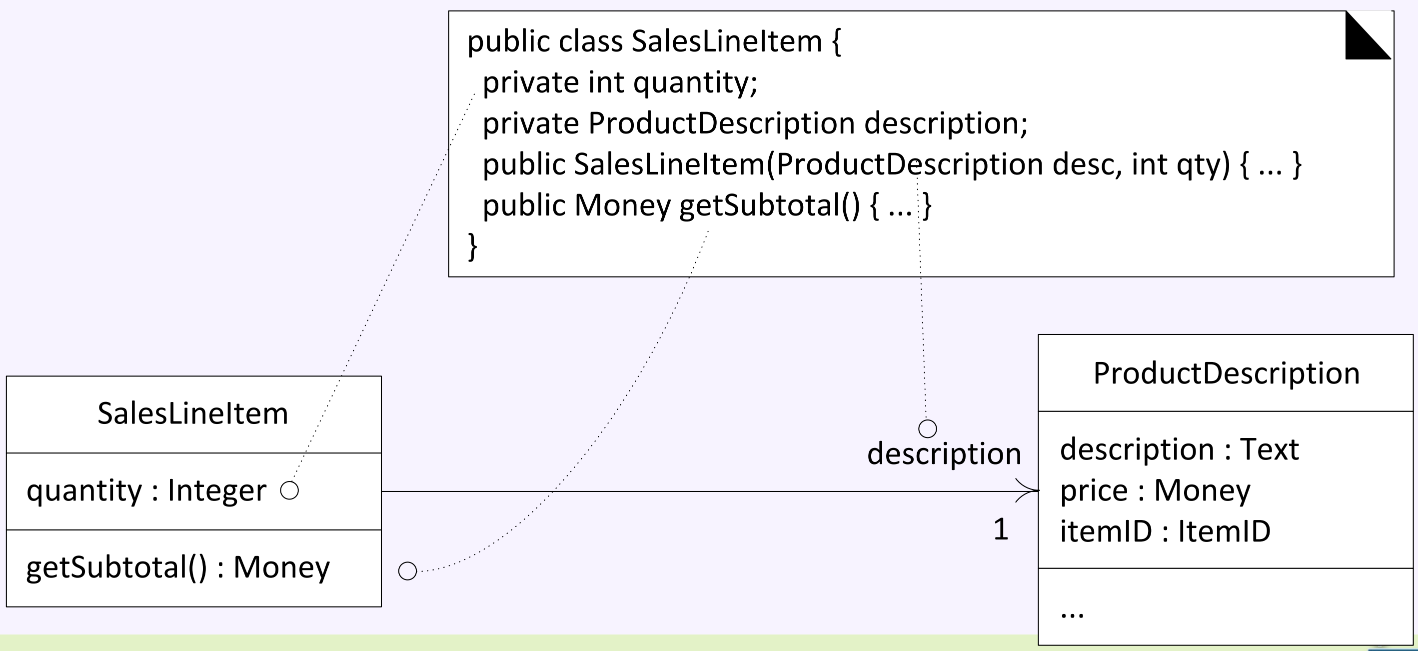


Resulting Design Model (example, excerpt)

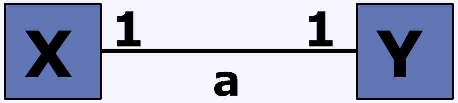


From Design to Implementation

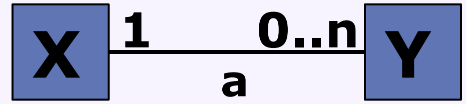
- Use Design Model as roadmap for implementation
- Decision making and creativity still required
  - Models typically incomplete at first
  - Models foster better understanding and help making better implementation decisions
- Start with class with least dependencies



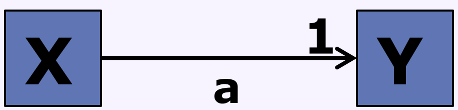
Implementing Associations



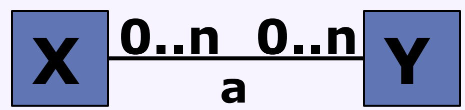

```
class X {
  Y a;
}
class Y {
  X a;
}
```




```
class X {
  List<Y> a;
}
class Y {
  X a;
}
```




```
class X {
  Y a;
}
class Y {}
```




```
class X {
  List<Y> a;
}
class Y {
 List<X> a;
}
```

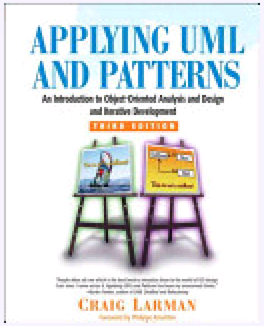
Summary: Phases and Terminology

- Conceptual Modeling / Object-Oriented Analysis
  - Create **Domain Model** / Conceptual Model
  - Analyzing the Domain, Vocabulary for further Design
  - Visualization of the concepts or mental models of a real-world domain
  - UML for sketching (conceptual perspective)
- Object-Oriented Design
  - **Design Model** / Object Model / Design Class Diagrams
  - Classes, Objects and their behavior and relationships
  - UML as a blueprint (specification perspective)
- Implementation
  - Mapping Designs to Code
  - Implementing classes and methods
  - (Code generation; UML as a programming language; implementation perspective)

Summary

- Design requires tradeoffs
- Conceptual modeling to understand domain
  - UML as visual language
- Design goals
  - Modularity, Low Represenational Gap, Few Interfaces, …
- GRASP Principles for first design considerations
  - Information Expert
  - Creator
  - Low Coupling, High Cohesion
  - Controller

Literature



- Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  - Chapter 9 introduces conceptual modeling
  - Chapter 16+17+22 introduce GRASP
- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997
  - Chapter 3 and 4 discuss Design Goals and Modularity