OODP- OOA/OOD– Session 3

Object Oriented

Design

Oded Lachish

Department of Computer Science

and Information Systems

Birkbeck College, University of London

Email: oded@dcs.bbk.ac.uk

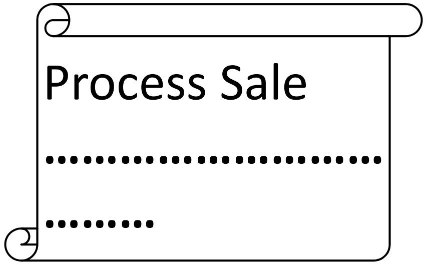
Web Page: http://www.dcs.bbk.ac.uk/~oded

Previously on OOAD

- In the beginning there was CHAOS
- Waterfall
- Agile
  - Iterative, Incremental, Interactive
    - Unified Process (UP)
      - Inception
      - Elaboration
      - …

Previously on OOAD

UP Use-case Model

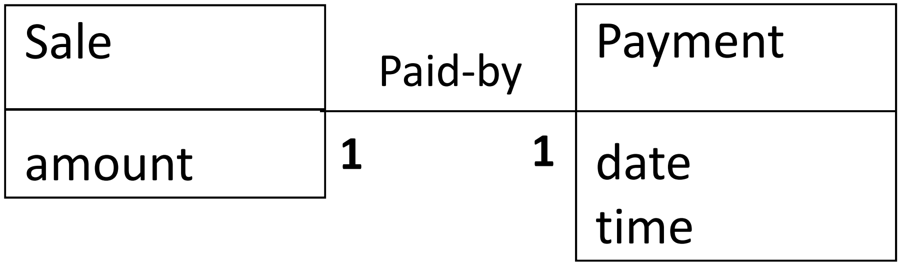

Process Sale

……………………….

………

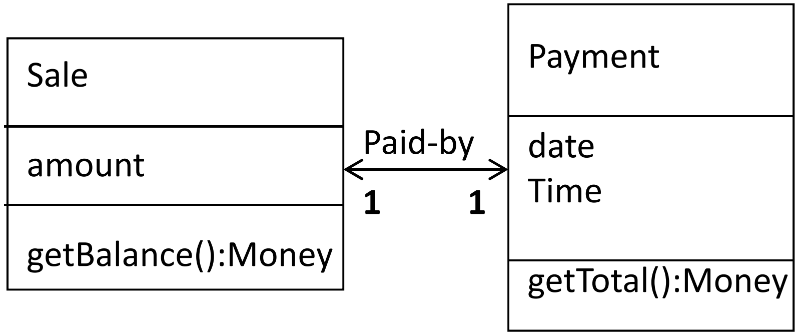
System sequence diagram

UP Domain Model

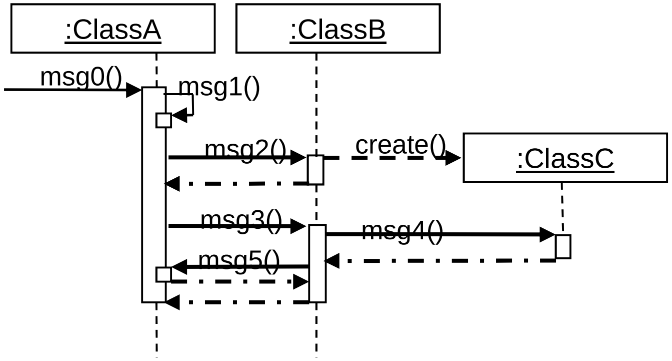


UP Design Model

Design Class Diagrams



Interaction Diagrams



Some remarks on previously

- Testing
- Instantiation Use Case
- Contracts etc.
- Coding

Starting Point, Getting Personal

- The use cases describe different tasks that must be performed in their context. Who is responsible for performing these tasks?

- The objects are responsible!

- Objects have responsibilities (grow up objects)

Responsibilities

Objects have responsibilities - obligations of an object in terms of its behaviour:

- Doing
  - Running a computation, creating an object
  - Initiating action in another object
  - Controlling and coordinating activities in other objects
- Knowing
  - Private encapsulated data
  - Related objects
  - What can it do

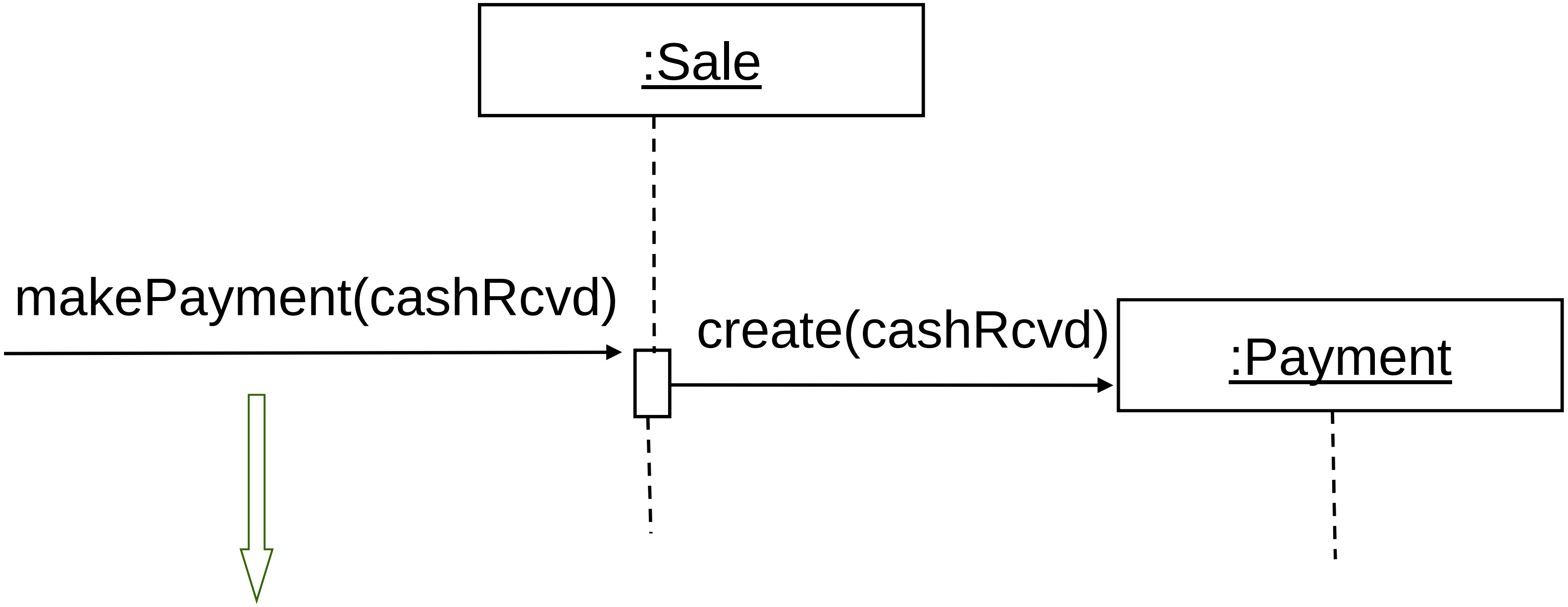
Responsibilities and methods

- In general, responsibilities are implemented using methods.
- Methods either act alone or **collaborate** with other object methods.

Object collaboration often results in dependencies.

Responsibilities and interaction diagrams

- How do we assign responsibilities (implemented as methods) to objects? Interaction diagrams show choices in assigning responsibilities.



It implies Sale objects "have" a responsibility to create Payments. This responsibility is invoked with a makePayment message and handled with a corresponding makePayment method.

Where are we now?

- We have UML classes that can be converted to software classes and attributes
- We have Interaction Diagrams that can be converted to methods
- We even got personal with the objects

What don’t we have?

- The means to make this
  - Easier to Modify
  - Easier for Maintenance
  - Easier for Verification

Hand waving

Easier to Modify, Easier for Maintenance, Easier for Verification.

How?

- Low interdependency – modification impact should be as local as possible
- “Well Structured” – similar functionality concentrated in the same place
- Modularity

Conclussion

- A critical task in the Design Model is to

  assign responsibilities to objects.

- How do we deal with this?

  Experience

  (other peoples experience, especially if you don’t have enough of your own)

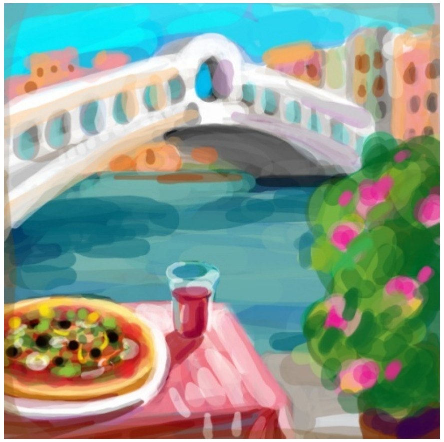
Patterns

Experience

- Start from other peoples experience
- How is experience passed on in other engineering areas.
- Civil engineering
- Specifically: building bridges

Bridges

- A bridge is a structure which is used for traversing a chasm.
- In its basic form it consists of a beam constructed from a rigid material.
- The two ends of the beam are fixed at opposite edges of the chasm



- The bridge will fulfil its function if the rigidity of the beam can support the loads that go over it.
- Heavier loads may tax the rigidity of the bridge.
- The rigidity depends on both

the length and the material

that the beam is made of.

Modifying the design

If the bridge might fail:

- heaviness of the load
- size of span
- material of construction

Then modify bridge design

- increase the rigidity
- decrease the span

Increase the rigidity/redistribute the material

Civil engineering patterns

- These are ALL the patterns of bridge designs.
- Civil engineers only build bridges following one of the patterns shown.
- The idea of patterns comes from architects who also follow a fixed number of designs.
- Why should software design be different from design in other engineering disciplines?

What is a pattern?

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Patterns

- Patterns guide object-oriented developers in the creation of software.
- They codify existing tried-and-true knowledge and principles; the more widely they are used, the better.
- They are expressed in a standard format:
  - **Pattern Name:** …
  - **Problem It Solves:** …
  - **Solution:** …

Why are patterns useful?

- Patterns are very simple to use.
- If one learns how to apply them properly, they will generate designs that are easy to understand, maintain and reuse.
- By referring to pattern names, designers can communicate easily and explain the reasons behind their design choices.
- Patterns provide a formal framework for generating good designs => avoid hand-waving.

GRASP

Basic Patterns

GRASP -

GRASP – General Responsibility Assignment Software Patterns

(some may principles and not patterns)

- A learning aid for understanding object design.
- Putting basic design ideas into words.

Pattern Example: The Information Expert

- **Pattern Name:** Information Expert
- **Problem It Solves:** What is a general principle of assigning responsibility to objects?
- **Solution:** Assign a responsibility to the information expert- the class that has the information necessary to fulfil the responsibility.

The GRASP patterns

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

What to look for in GRASP patterns

- Low Coupling –

  reducing the impact of change

- High Cohesion –

  similar functionality in the same place

- Low representational gap (LRP) –

  minimal conceptual gaps

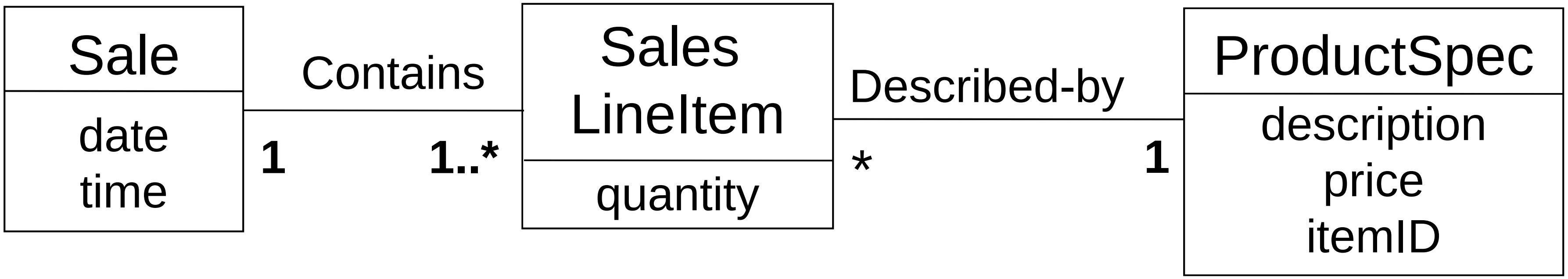
The Creator pattern

- **Pattern Name:** Creator
- **Problem It Solves:** Who should be responsible for creating a new instance of some class?
- **Solution:** Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - B aggregates A objects
  - B contains A objects
  - B records instances of A objects
  - B closely uses A objects
  - B has the initializing data that will be passed to A when it is created

B is a *creator* of A objects. If more than one option applies, prefer a class B which aggregates or contains class A.

Creator example

- Which class should be responsible for creating a new SalesLineItem?



Sale

date
time

Contains

1

1..*

Sales LineItem

quantity

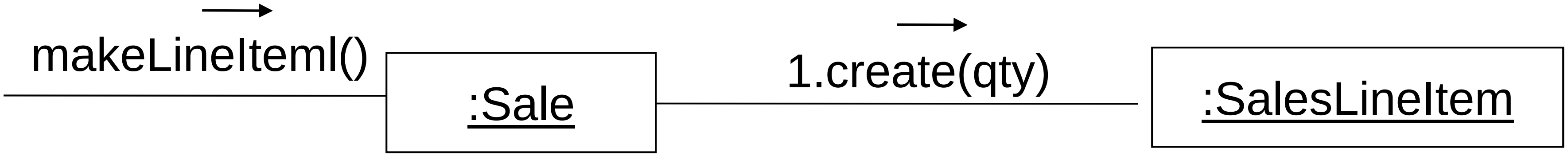
Described-by

*

1

ProductSpec

description
price
itemID



makeLineIteml()

:Sale

1.create(qty)

:SalesLineItem

Discussion about the Creator

- Goal: find the creator of objects of a class
- Guideline: look for relationships like *aggregates, contains, records* etc. The enclosing aggregator, container or recorder is a good candidate for creating the object aggregated, contained or recorded.
- The creator may be chosen as the class that knows initializing data for the creation operation. Which other pattern does this remind you of?
- How does the Creator pattern achieve low coupling?

Contraindications

What if the creation requires heavy machinery?

(recycled instances, uncertainty about the class of the created object)

We may get

- Low Cohesion
- Strong coupling

Solution:

- Delegate to another class
- Use advanced patterns such as

  Concrete Factory, Abstract Factory

The Information Expert pattern

- **Pattern Name:** Information Expert
- **Problem It Solves:**

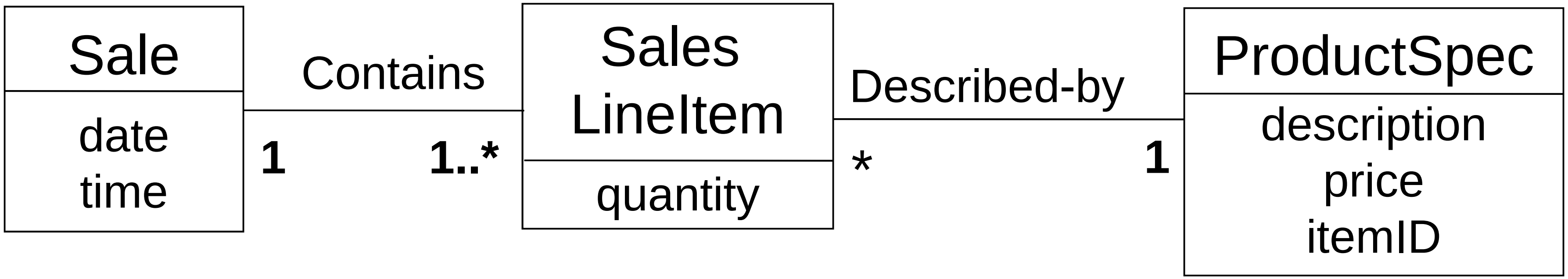
  What is a general principle of assigning responsibility to objects?

- **Solution:**

  Assign a responsibility to the information expert, i.e. the class that has the information necessary to fulfil the responsibility.

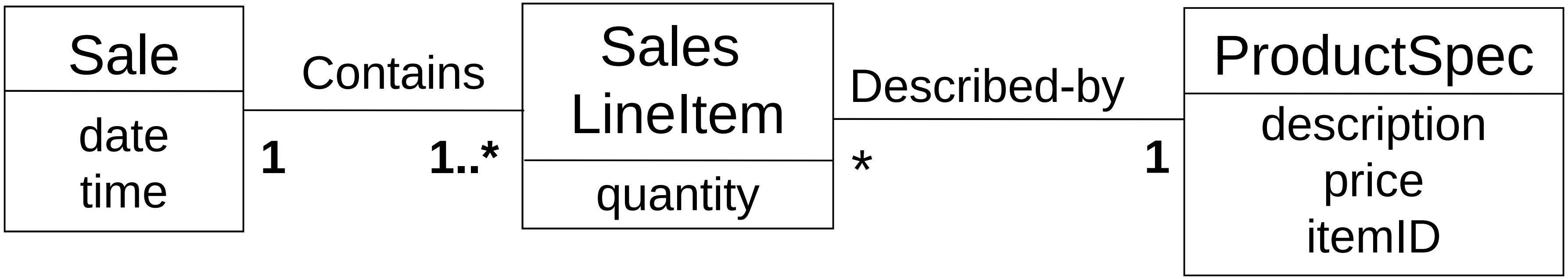
Information Expert example

- Which classes are responsible for evaluating the total of a sale, the subtotal of a SalesLineItem, the price of a product?



Information Expert example

Design Class	Knowing Responsibility
Sale	**Sale total**
SalesLineItem	**Logical Design Diagrams**
ProductDescription	**Product Price**



Discussion about the Information Expert

- The Information Expert is the most widely used guiding principle in object design.
- Information is spread across many classes => “partial” information experts that collaborate
- Software objects are “alive” (animated), they can do things based on the information they hold.
- How is Information Expert related to low coupling and high cohesion?

Contraindication

- What if Sale should be saved in a database
- Should this be a responsibility of Sale?

What is the Problem?

Solution?

The Low Coupling pattern

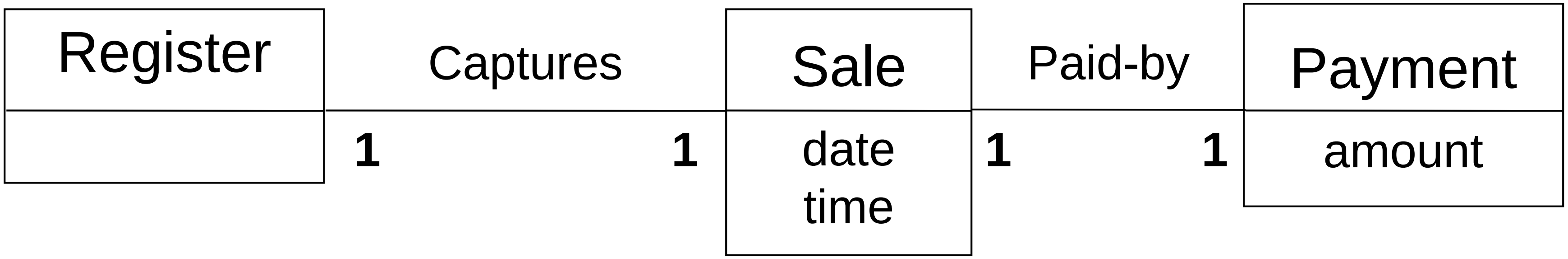
- **Pattern Name:** Low Coupling
- **Problem It Solves:**

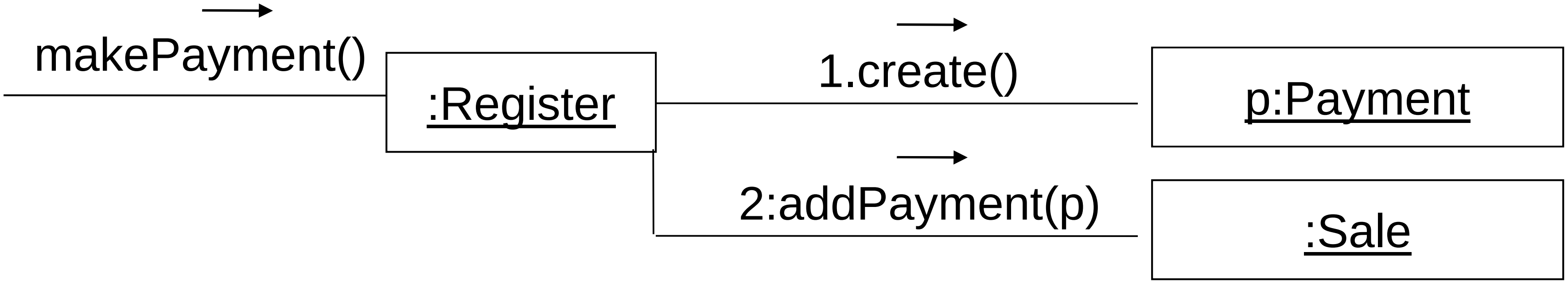
  How to support low dependency, low change impact, and increased reuse?
- **Solution:**

  Assign a responsibility so that coupling remains low.

Low coupling example

- Who should be responsible for creating a Payment and associating it with a Sale?

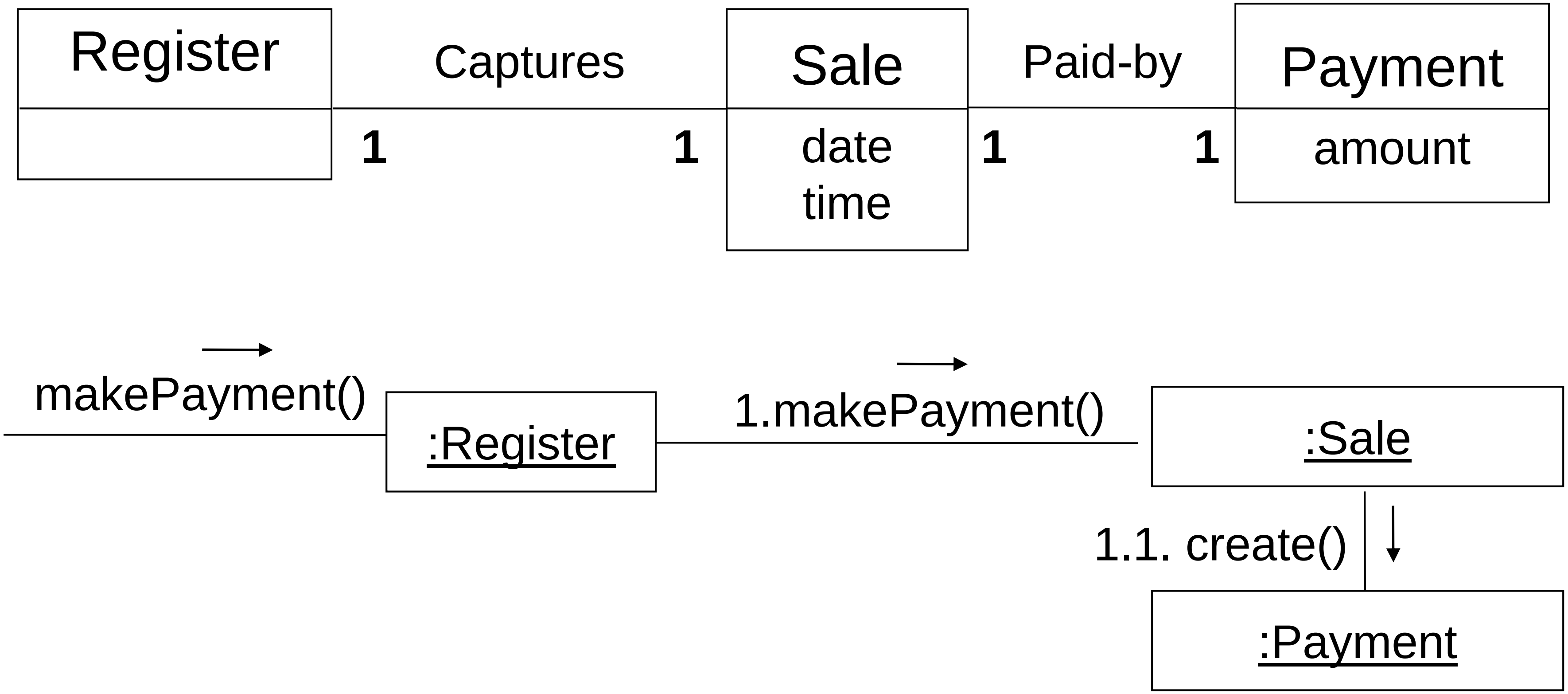




Is this the best possible design? How many objects are related to (coupled with) the Register object? Can we achieve lower coupling?

Low coupling example

- Who should be responsible for creating a Payment and associating it with a Sale?



Register is now coupled with objects of one class (Sale). If the class Payment is modified, only Sale will be affected. This design is preferred because it provides lower coupling between classes.

Discussion about Low Coupling

- Common forms of coupling from TypeX to TypeY include:
    - TypeX has an attribute that refers to a TypeY instance
    - TypeX calls on services of a TypeY object
    - TypeX has a method that references an instance of TypeY, or TypeY itself
    - TypeX is a direct or indirect subclass of TypeY
    - TypeY is an interface, and TypeX implements that inteface
- Low coupling supports the design of classes that are more independent => reduces impact of change

Discussion about Low Coupling

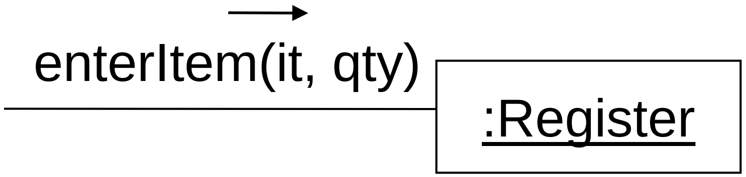
- Classes that are inherently generic and are reused in many places must have a particularly low degree of coupling
- Contraindication: It is safe to couple a class to stable classes (e.g. classes in Java libraries)
- Coupling becomes problematic when *unstable* elements (classes) are involved. What does it mean for a class to be unstable?
- Benefits of low coupling
  - Easy maintenance
  - Classes easy to understand
  - Software reuse

The Controller pattern

- **Pattern Name:** Controller
- **Problem It Solves:** 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 one of the following choices:
  - Represents the overall system, device, or subsystem (*facade controller)*
  - Represents a use case scenario within which the system event occurs, often named <UseCase>Handler, <UseCase> Coordinator, or <UseCase>Session *(use-case or session controller)*

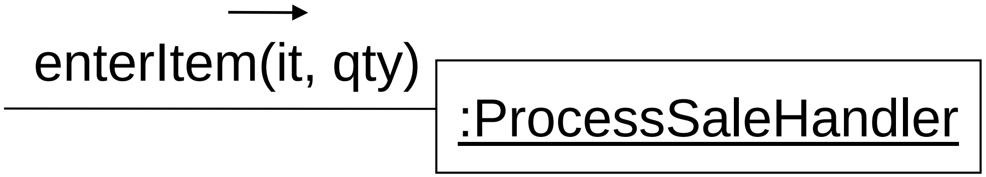
Controller example

- Who should be responsible of handling external system events (e.g. enterItem)? Which of the following interaction diagrams is preferred?



enterItem(it, qty)

:Register



enterItem(it, qty)

:ProcessSaleHandler

Discussion about the Controller

- The Controller pattern provides guidance about which object will handle external events.
- If all system events of a use case are handled in the same class, it is possible to identify out-of-sequence system events.
- Common defect in the design of controllers: too much responsibility
- A controller delegates work to other objects. It plays the role of a coordinator, but does not actually do the work.

Discussion about the Controller

- System operations should be handled at the domain layer by controllers, not at the interface layer by GUI objects.

- Benefits:
  - Reuse of domain layer software, by plugging different interfaces
  - Control of out-of-sequence system operations

The High Cohesion pattern

- **Pattern Name:** High Cohesion
- **Problem It Solves:**

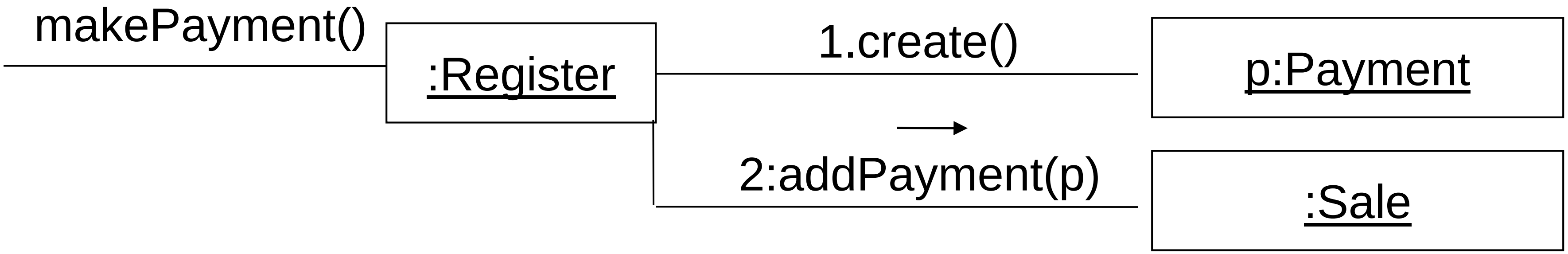
  How to keep complexity manageable? How to keep objects focused.

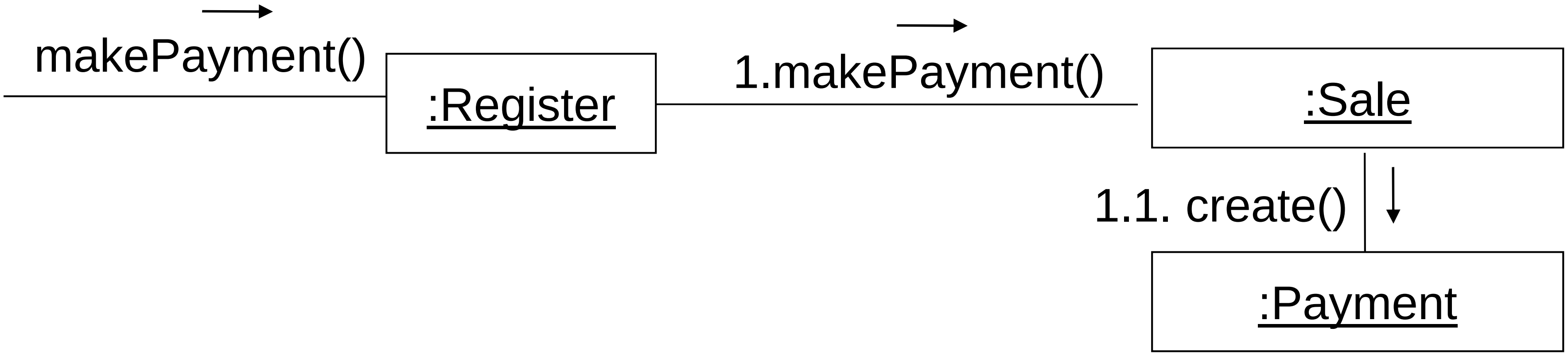
- **Solution:**

  Assign a responsibility so that cohesion remains high.

High cohesion example

- Who should be responsible for creating a Payment and associating it with a Sale? Comment on the level of cohesion of Register in the two diagrams. Is Register at risk of becoming incohesive in the future and why?





Discussion about High Cohesion

- A class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work.
- Benefits of high cohesion:
  - Easy maintenance
  - Classes easy to understand
  - Software reuse
- Coupling and cohesion viewed together => modular design

  Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules [Booch94]
- Contraindications: maintenance by one person, remote communication

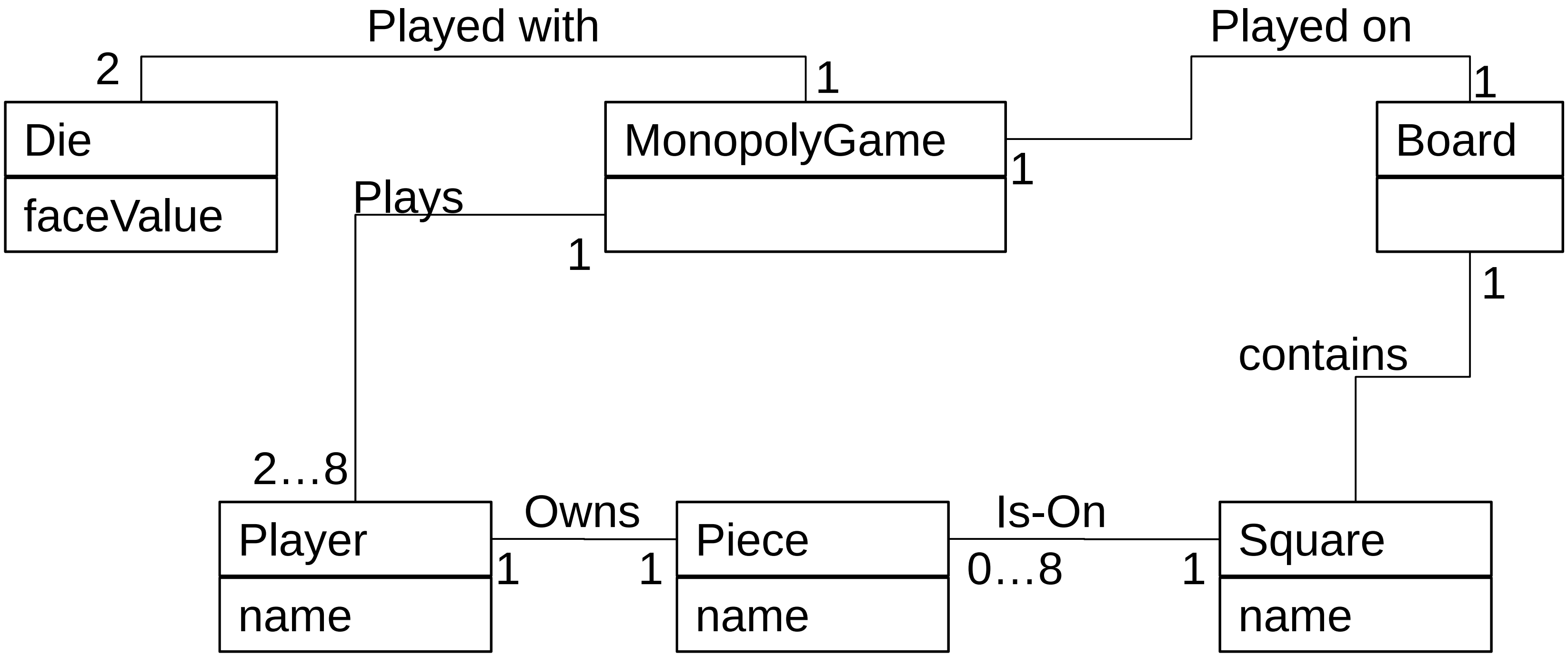
Use Case Realization

Monopoly game simulation

Monopoly Use-Cases

- The game starts, first player roles the dice, first player looks at value on dice, first player moves his piece value positions. Second…, i’th player…, first player,…, and this is all repeated n times.
- Instantiation

Monopoly Domain Model



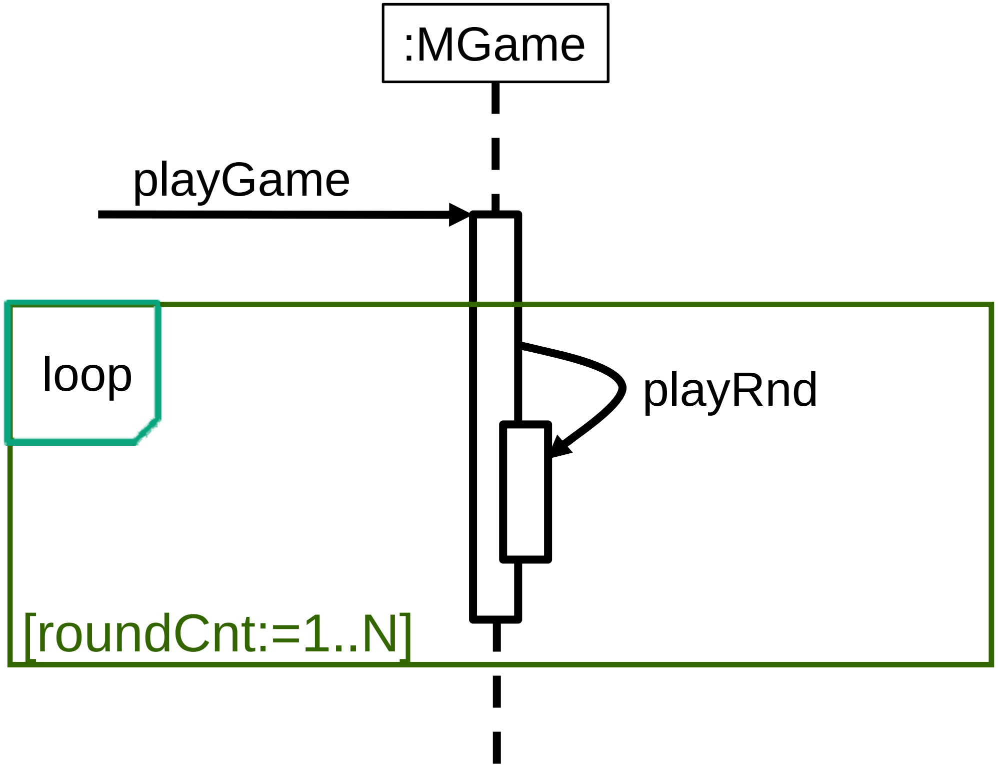
Monopoly: choosing a controller class

Two options

- MonopolyGame – one object to handle, simple controller pattern application
- MonopolyGameHandler – special object according to an advanced pattern

Monopoly Game Loop

- Who is responsible for Controlling the game loop? (Expert)

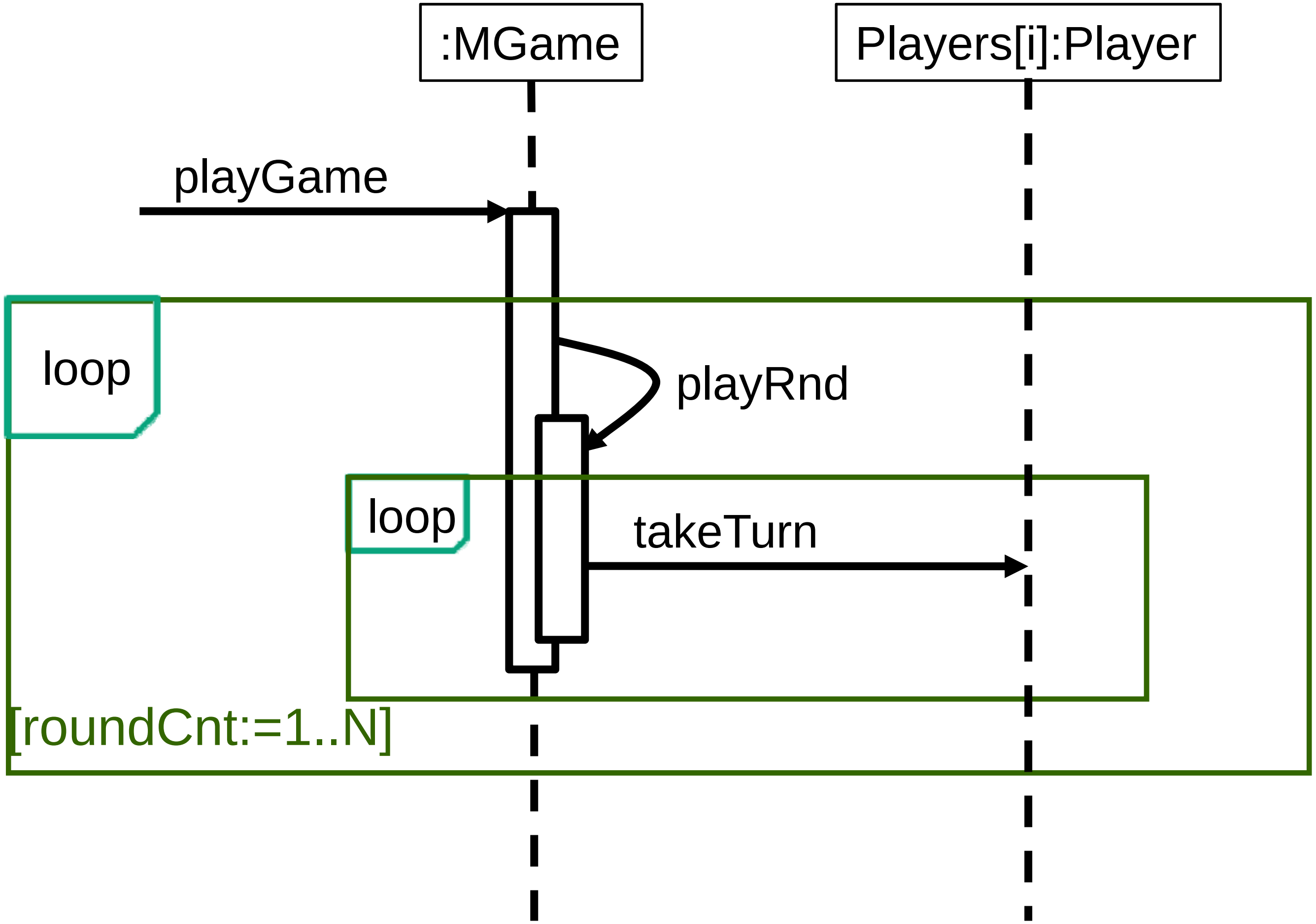


Monopoly Game Loop

- Who takes a turn? (Expert)
- Who has the information needed?

Information Needed	Who has it?
Current location of player	**A player knows its piece and a piece knows it square (domain model)**
The die object (for rolling etc.)	**MGame**
All the squares	**Board**

Monopoly Player Takes a Turn



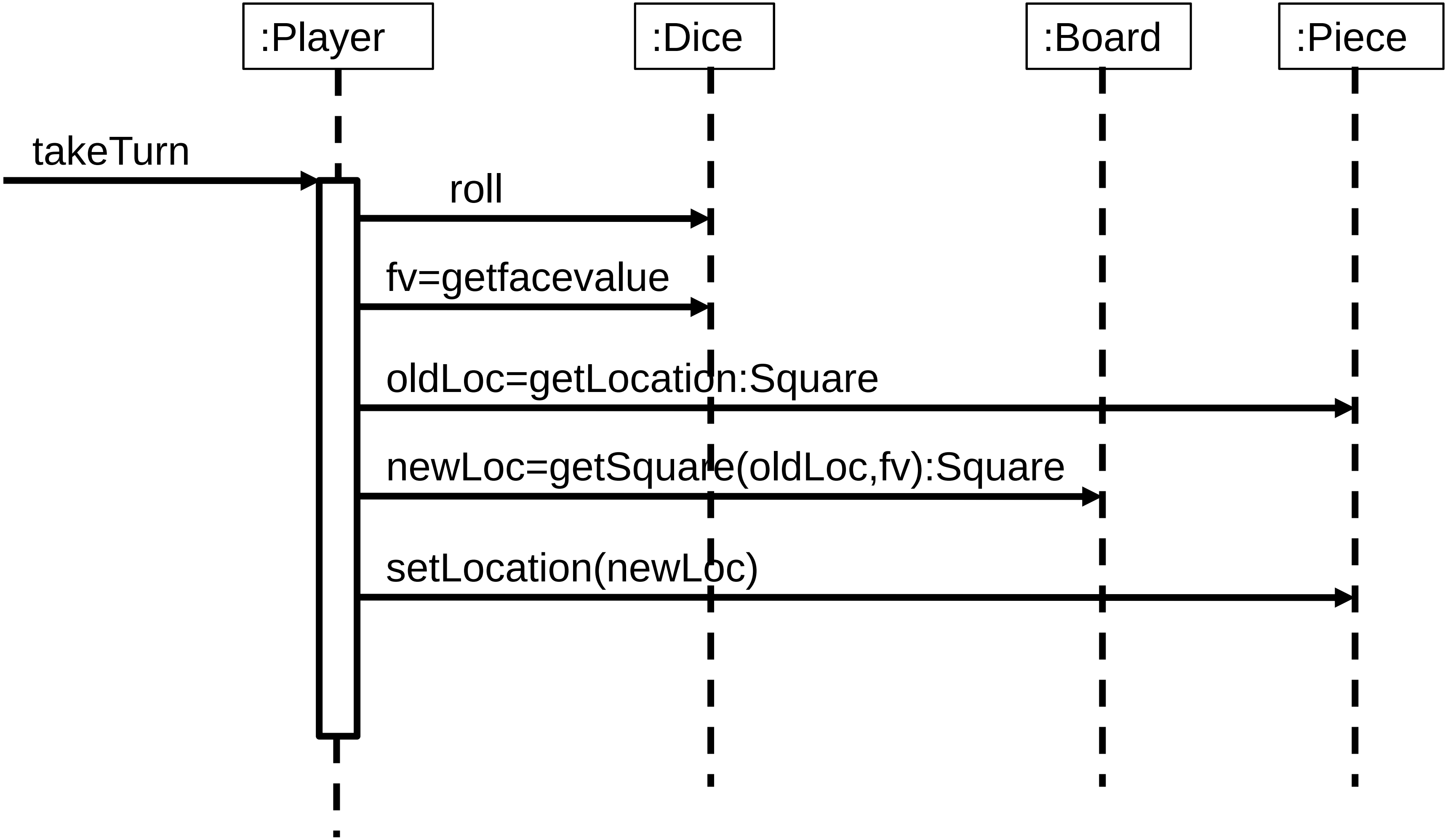
Monopoly Taking a Turn

1. Calculating a random number in the range 2…12
2. Calculating the new square location
3. Moving players piece

- Who computes the random number? die class
- Who computes new square position? Board, knows all its squares
- Who sets new location? piece class
- Who coordinates all of this?

Remember: these objects appeared in the domain model, but we are dealing with the design model

Monopoly Player Takes a Turn



Advanced Patterns

Classification (GOF)

- Creational Patterns
  - Abstract object instantiation
- Structural Patterns
  - Compose and organise objects into larger structures
- Behavioural Patterns
  - Algorithms, interactions and control flow between objects.

Creational Patterns

- *Singleton - for creating classes which must have only a* single instance (e.g. a printer spooler)
- *Factory method - used when a class can’t anticipate the* class of objects it must create but it wants its subclasses to specify the objects it creates
- *Abstract factory - provides an interface for creating* families of related objects without the need to specify their concrete classes
- *Prototype - creating objects from object instances*

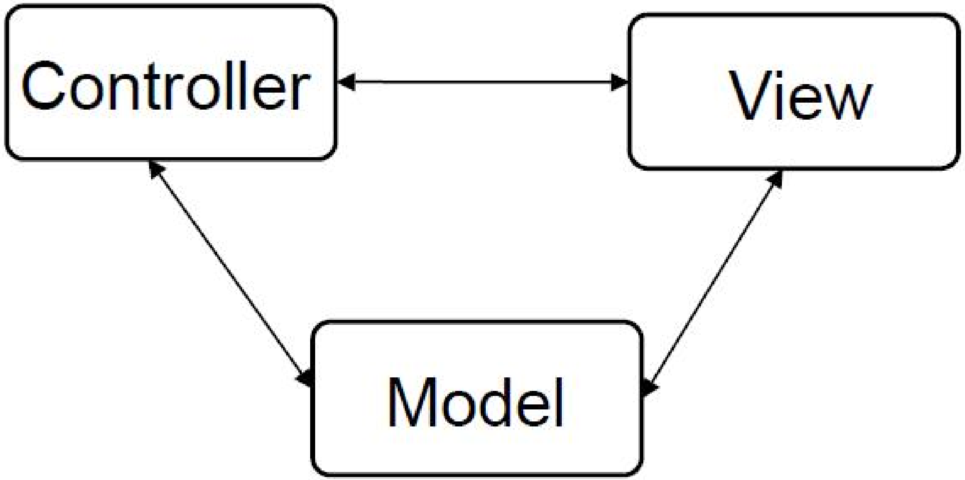
Example: Model-View-Controller (MVC)

Model-View-Controller group of classes used to build interfaces (originally in Smalltalk, Java Swing, EPOC uses MVC).

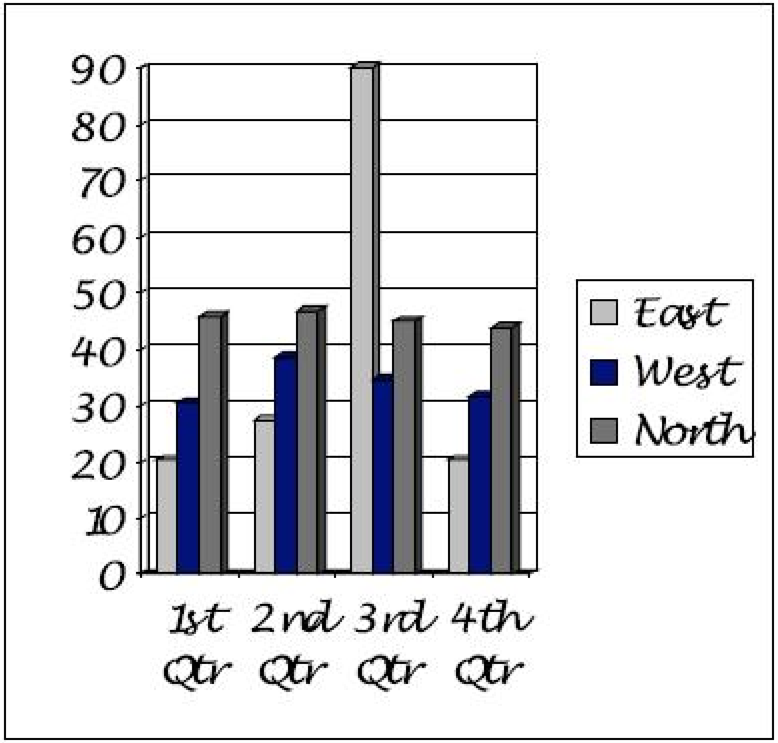
- Model - application object,
- View - presentation on screen,
- Controller - how user input controls the interface

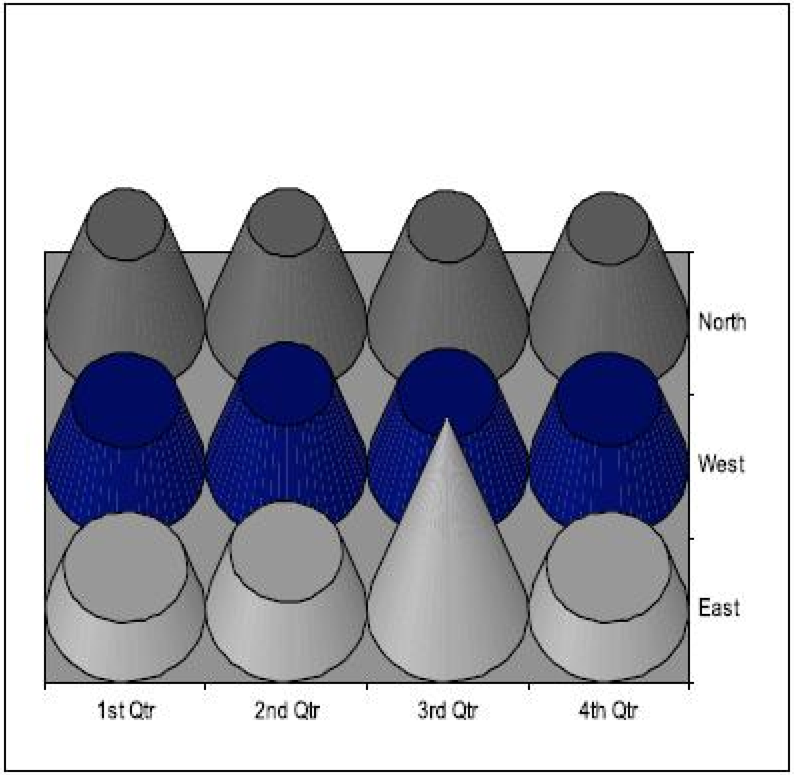
MVC

- Model must notify views of change.
- Views must keep themselves up to date.
- Several controllers e.g. command keys, pop-up menus can be used. Usually controllers organised in a class hierarchy -> sub-classing



Two views of the data



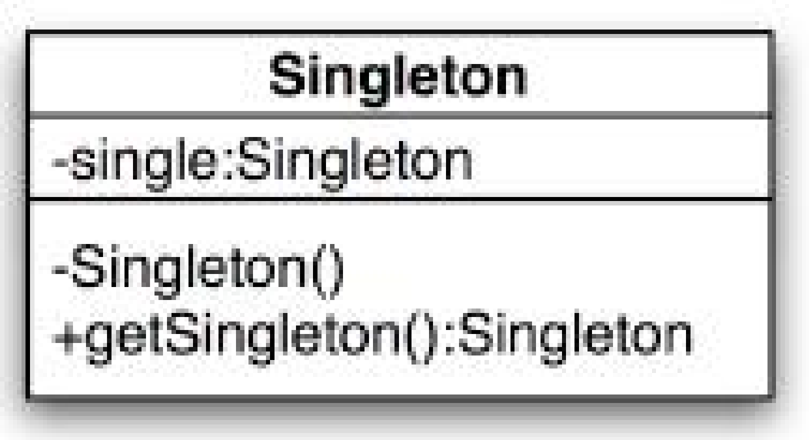


MVC Advantages

- MVC decouples the model management (data) from the representation (views) and the reaction to user input (controllers). Increased flexibility.
- Allows to have multiple (synchronized views) on the same data.
- Allows to associate different controllers with each of the views if needed. Change the way in which the interface reacts without changing the interface.
- Views can be easily composed.

Singleton

Structure:



Intent:

Ensure a class has one instance, and provide a global point of access to it.

Singleton - Example

```
Public class PrintSpooler {
     // a prototype for a spooler class,
     // such that only one instance can ever exist
   private static PrintSpooler _spooler;
   private PrintSpooler() { /* private constructor */ }
   public static synchronized getSpooler() {
     if (_spooler == null) _spooler = new PrintSpooler();
     return _spooler;
   }
}
```

```
class Main {   // example of use
   public static void main(String args[]) {
   PrintSpooler spl =PrintSpooler.getSpooler();
   spl.print ("Printing data");
   }
}
```