Design of Software Systems (Ontwerp van SoftwareSystemen)

2 Basic OO Design

Roel Wuyts
2015-2016

Acknowledgments

Tom Holvoet (KUL, Leuven, BE) for the GRASP pattern slides

Oscar Nierstrasz (University of Bern, CH) for the Tic Tac Toe example

The whole course in one slide ?

Basic OO Design Principles:

- **Minimize Coupling**
- **Increase Cohesion**
- **Distribute Responsibilities**

Basic OO Design Principles

No matter whether you use forward engineering or re-engineering,
These basic OO Design Principles hold:

- Minimize Coupling
- Increase Cohesion
- Distribute Responsibilities

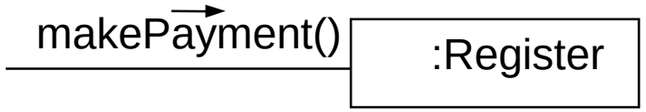
You should always strife to use and balance these principles

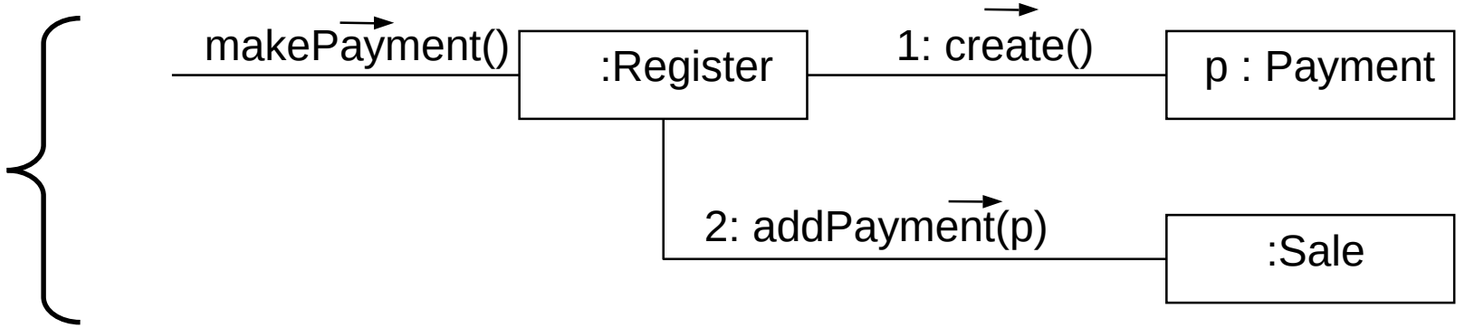
- they are fairly language- and technology independent
- processes, methodologies, patterns, idioms, ... all try to help to apply these principles in practice
  - It's still your job to determine the best balance

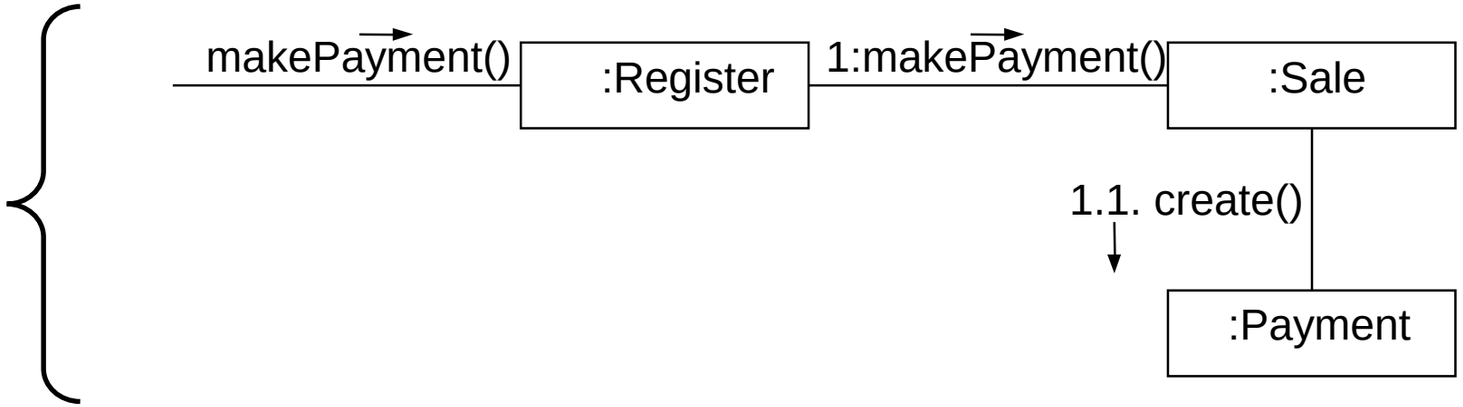
4. Low Coupling Pattern

Pattern	Low Coupling
Problem	How to stimulate low independance, reduce impact of change and increase reuse?
Solution	Assign responsibilities such that your design exhibits low coupling. Use this principle to evaluate and compare alternatives.

Low Coupling Pattern





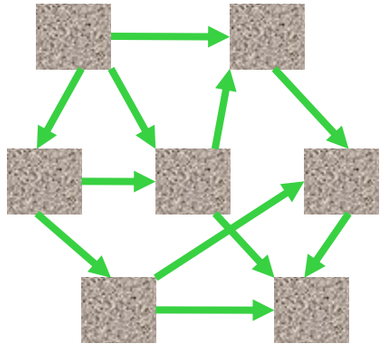


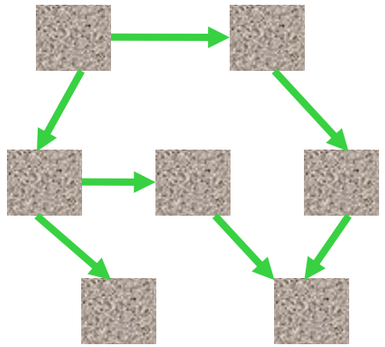
- Which design is better?
- Coupling to stable libraries/classes?
- Key principle for evaluating choices

Low Coupling Pattern

Coupling is a measure that shows how much a class is dependent on other classes

- X depends on Y (~ X does not compile without Y):
  - X has attribute of type Y
  - X uses a service of Y
  - X has method referencing Y (param, local variable)
  - X inherits from Y (direct or indirect)
  - X implements interface Y
- "evaluative" pattern:
  - use it to evaluate alternatives
  - try to reduce coupling





Low Coupling Pattern

Advantages of low coupling:

- reduce impact of changes (isolation)
- increase understandibility (more self-contained)
- enhance reuse (independance)

Is not an absolute criterium

- Coupling is always there
- Therefore you will need to make trade-offs !

Inheritance is strong coupling !!

Low Coupling Pattern: remarks

Aim for low coupling with all design decisions

Cannot be decoupled from other patterns

Learn to draw the line (experience)

- do not pursue low coupling in the extreme
  - Bloated and complex active objects doing all the work
  - lots of passive objects that act as simple data repositories
- OO Systems are built from connected collaborating objects

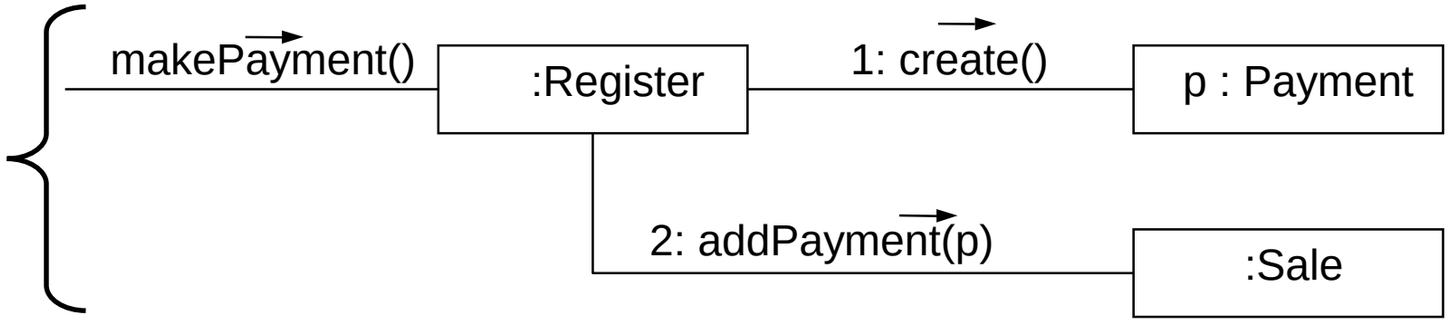
Coupling with standardized libraries is NOT a problem

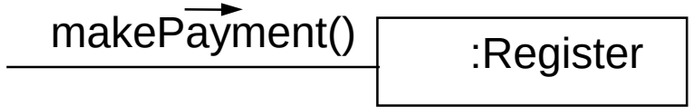
Coupling with unstable elements IS a problem

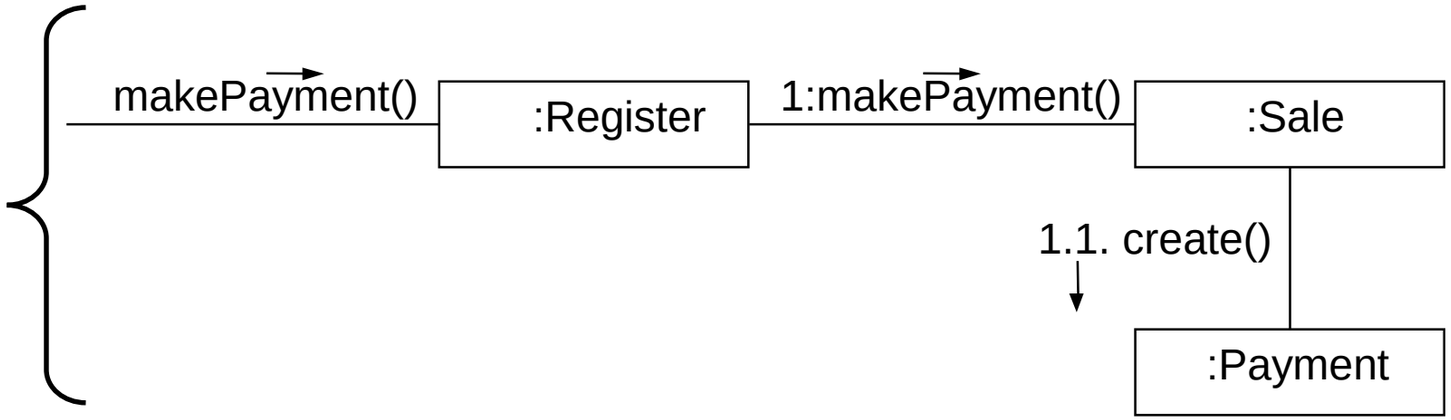
5. High Cohesion Pattern

Pattern	High Cohesion
Problem	How to retain focus, understandability and control of objects, while obtaining low coupling?
Solution	Assign responsibilities such that the cohesion of an object remains high. Use this principle to evaluate and compare alternatives.

High Cohesion Pattern







- Cohesion: Object should have strongly related operations or responsibilities
- Reduce fragmentation of responsibilities (complete set of responsibility)
- To be considered in context => register cannot be responsible for all register-related tasks

High Cohesion Pattern

Cohesion is a measure that shows how strong responsibilities of a class are coupled.

Is an “evaluative” pattern:

- use it to evaluate alternatives
- aim for maximum cohesion
  - (well-bounded behavior)

Cohesion ↘

- number of methods ↗ (bloated classes)
- understandability ↘
- reuse ↘
- maintainability ↘

High Cohesion Pattern: remarks

Aim for high cohesion in each design decision

degree of collaboration

- Very low cohesion: a class has different responsibilities in widely varying functional domains
  - class RDB-RPC-Interface: handles Remote Procedure Calls as well as access to relational databases
- Low cohesion: a class has exclusive responsibility for a complex task in one functional domain.
  - class RDBInterface: completely responsible for accessing relational databases
  - methods are coupled, but lots and very complex methods

…

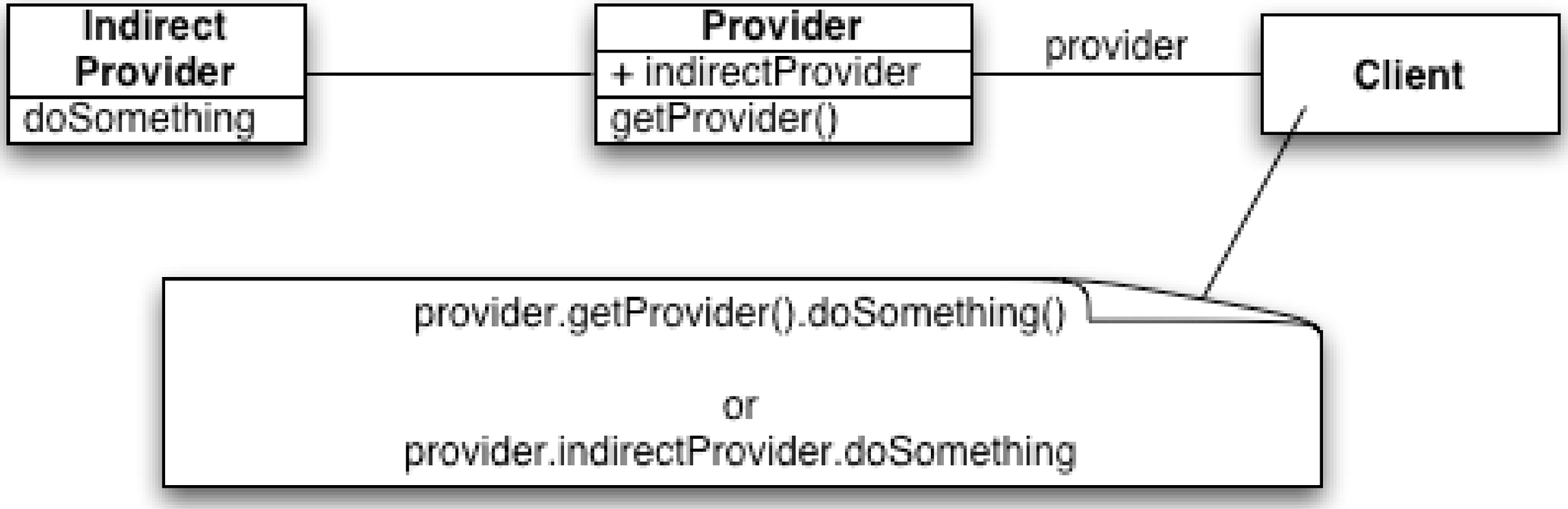
High Cohesion Pattern: remarks

Aim for high cohesion in each design decision

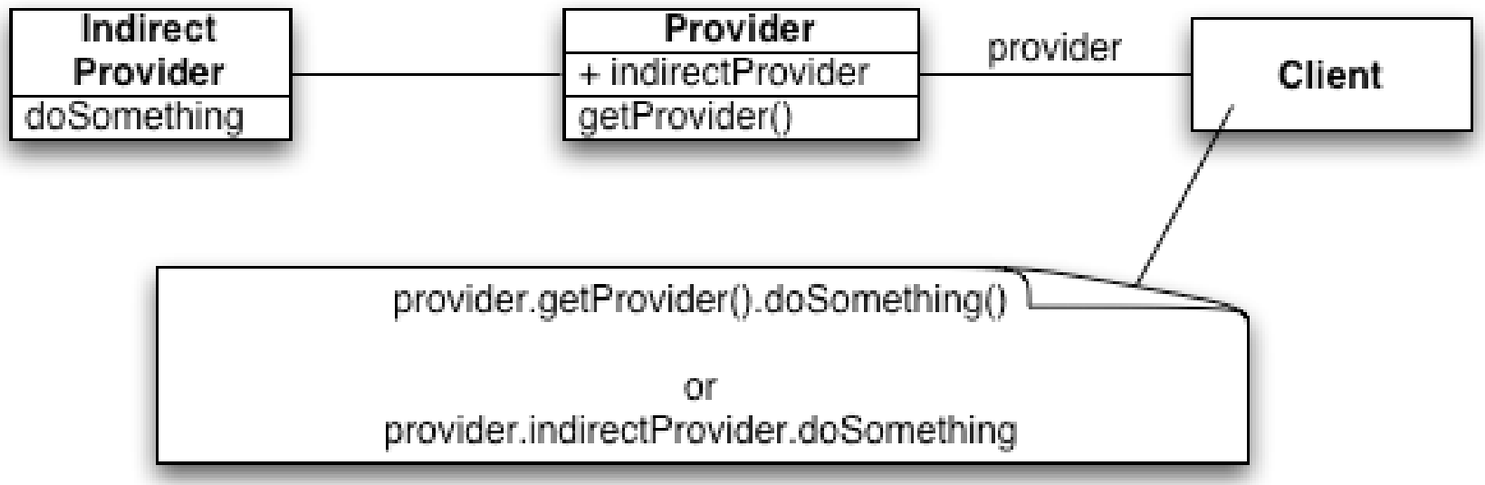
degree of collaboration (ctd)

- Average cohesion: a class has exclusive ‘lightweight’ responsibilities from several functional domains. The domains are logically connected to the class concept, but not which each other
  - a class Company that is responsible to manage employees of a company as well as the financials
  - occurs often in ‘global system’ classes !!
- High cohesion: a class has limited responsibilities in one functional domain, collaborating with other classes to fulfill tasks.
  - Class RDBInterface: partially responsible for interacting with relational databases

Example 1



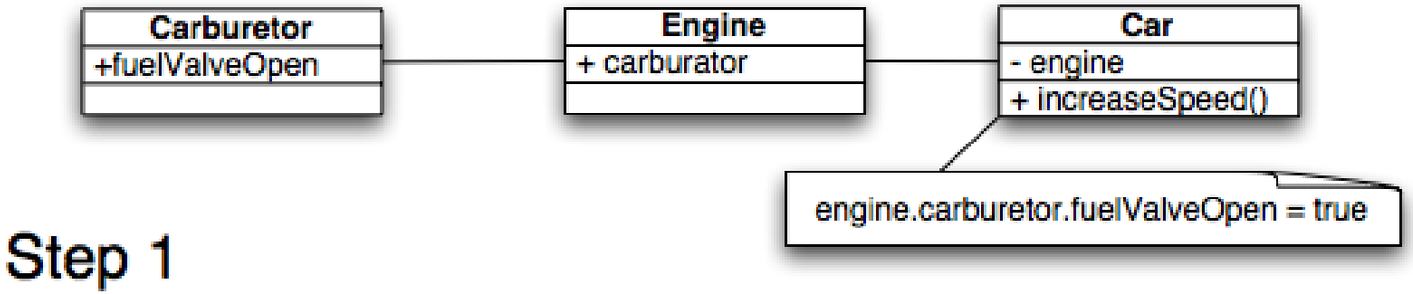
Why is this bad ?



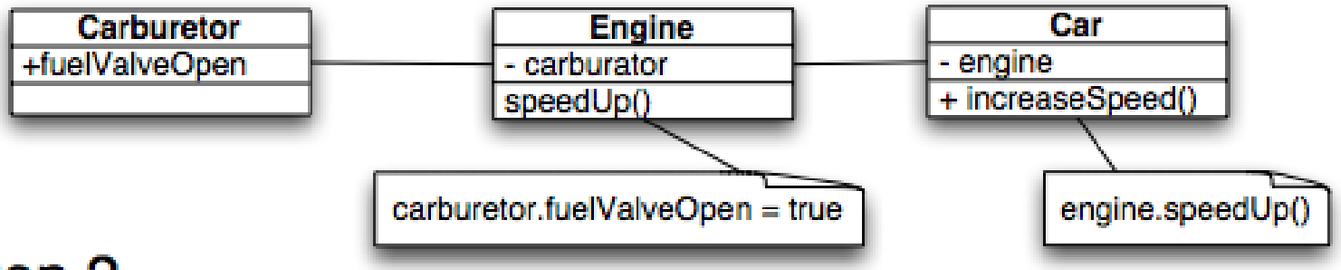
Client knows how Provider is implemented

- has to know that it uses an IndirectProvider
  - uses the interface of Provider as well as of IndirectProvider
- Client and IndirectProvider are strongly coupled !
  - Client has to use them together
  - Changing either Provider or IndirectProvider impacts Client

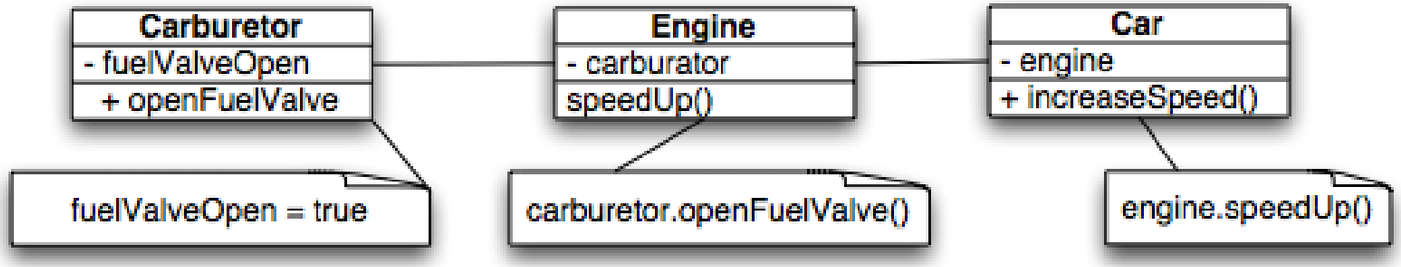
Reducing the Coupling



Step 1



Step 2



Reducing Coupling impacts the design

The interfaces of the classes become more clear

- a method ‘speedUp()’ makes perfect sense: cohesion

Allows for more opportunity for reuse

- A subclass of Engine, “ElectricalEngine”, might not need a Carburetor at all
  - This is transparent for Car

“Law of Demeter” / Don’t talk to strangers

Each unit should only talk to its friends;
don't talk to strangers

or, more mechanically:

You are only allowed to send messages to:

- yourself (self/this, super)
- an argument passed to you
- an object you create

Lieberherr, Karl. J. and Holland, I., Assuring good style for object-oriented programs, IEEE Software, September 1989, pp 38-48

Example 2

procedural approach:
passing an int and using switch to decide which behavior to execute based on that int

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
  {
  switch ( aCommand )
    {
        case EAknSoftkeyExit:
        case EAknSoftkeyBack:
        case EEikCmdExit:
            { Exit();  break; }

        // Play command is selected
        case EVideoCmdAppPlay:
            { DoPlayL(); break; }

         // Stop command is selected
        case EVideoCmdAppStop:
            { DoStopL(); break; }

        // Pause command is selected
        case EVideoCmdAppPause:
            { DoPauseL(); break; }

        // DocPlay command is selected
        case EVideoCmdAppDocPlay:
            { DoDocPlayL(); break; }
        ......
```

Nokia S60 mobile video player 3gpp source code
http://www.codeforge.com/article/192637

Why is this bad ?

Case (switch) statements in OO code are a sign of a bad design

- lack of polymorphism: procedural way to implement a choice between alternatives
- hardcodes choices in switches, typically scattered in several places
  - when the system evolves these places have to be updated, but are easy to miss

See also: Replace Conditional with Polymorphism
(http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism)

Solution: Replace case by Polymorphism

OO approach:
passing an object and using polymorphism to select behavior to execute

```
void CVideoAppUi::HandleCommandL(Command aCommand)
  {
        aCommand.execute();
       }
```

Create a Command class hierarchy, consisting of a (probably) abstract class AbstractCommand, and subclasses for every command supported. Implement execute on each of these classes:

- virtual void AbstractCommand::execute() = 0;
- virtual void PlayCommand::execute() { ... do play command ...};
- virtual void StopCommand::execute() { ... do stop command ...};
- virtual void PauseCommand::execute() { ... do pause command ...};
- virtual void DocPlayCommand::execute() { ... do docplay command ...};
- virtual void FileInfoCommand::execute() { ... do file info command ...};

Added advantage

These case statements occur wherever the command integer is used in the original implementation

- you will quickly assemble a whole set of useful methods for these commands
- Moreover, commands are then full-featured classes so they can share code, be extended easily without impacting the client, ...
- They can also be used when adding more advanced functionalities such as undo etc.

Have you noticed that the methods are shorter ?

Open question: can you think of disadvantages ?

Stepping Back

Showed concrete examples (and solutions) of breaches of basic OO design principles visible in code

- Fixing them improved the design!

Question: how can we avoid this ?

- be cautious ;-)
- get help by applying:
  - Design principles and methodologies
    - eg.: Responsibility Driven Design
  - GRASP patterns, Design Patterns
  - Idioms and Programming Practices
- use your head!

Responsibility Driven Design

http://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf

Metaphor – can compare to people

- Objects have responsibilities
- Objects collaborate

In RDD we ask questions like

- What are the responsibilities of this object ?
- Which roles does the object play ?
- Who does it collaborate with ?

Domain model

- classes do NOT have responsibilities!
- they merely represent concepts + relations
- design is about realizing the software ➔ someone has to do the work … who ??

Understanding Responsibilities

is key to good OO Design

RDD Process

Design = incremental journey of discovery and refinement

- build knowledge to take proper decisions
- start by looking for classes of key objects
  - can use the domain model for inspiration !
- then think about what actions must be accomplished, and who will accomplish them - how to accomplish them is for later !
  - Leads to responsibilities

Responsibilities

Two types of responsibilities

- Doing
  - Doing something itself (e.g. creating an object, doing a calculation)
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
- Knowing
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate

Object Collaboration

Objects collaborate: one object will request something from another object

To find collaborations answer the following questions:

- What other objects need this result or knowledge?
- Is this object capable of fulfilling this responsibility itself?
- If not, from what other objects can or should it acquire what it needs?

Cfr: Coupling and Cohesion !

Example: Tic Tac Toe

Requirements:

> "A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal."
>
> — Random House Dictionary

We should design a program that implements the rules of Tic Tac Toe.

Setting Scope

Questions:

- Should we support other games?
- Should there be a graphical UI?
- Should games run on a network? Through a browser?
- Can games be saved and restored?

A monolithic paper design is bound to be wrong!

Setting Scope

A monolithic paper design is bound to be wrong?

Let's follow an *iterative development* strategy:

- limit initial scope to the minimal requirements that are interesting
- grow the system by adding features and test cases
- let the design emerge by refactoring roles and responsibilities

How much functionality should you deliver in the first version of a system?

- Select the minimal requirements that provide value to the client.

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Tic Tac Toe Objects

Some objects can be identified from the requirements:

Objects	***Responsibilities***
Game	Maintain game rules
Player	Make moves Mediate user interaction
Compartment	Record marks
Figure (State)	Maintain game state

Entities with clear responsibilities are more likely to end up as objects in our design.

Tic Tac Toe Objects …

Others can be eliminated:

Non-Objects	***Justification***
Crosses, ciphers	Same as Marks
Marks	Value of Compartment
Vertical lines	Display of State
Horizontal lines	ditto
Winner	State of Player
Row	View of State
Diagonal	ditto

✎ How can you tell when you have the "right" set of objects?

✓ *Each object has a clear and natural set of responsibilities.*

Missing Objects

Now we check if there are unassigned responsibilities:

- Who starts the Game?
- Who is responsible for displaying the Game state?
- How do Players know when the Game is over?

Let us introduce a Driver that supervises the Game.

? How can you tell if there are objects missing in your design?

- When there are responsibilities left unassigned.

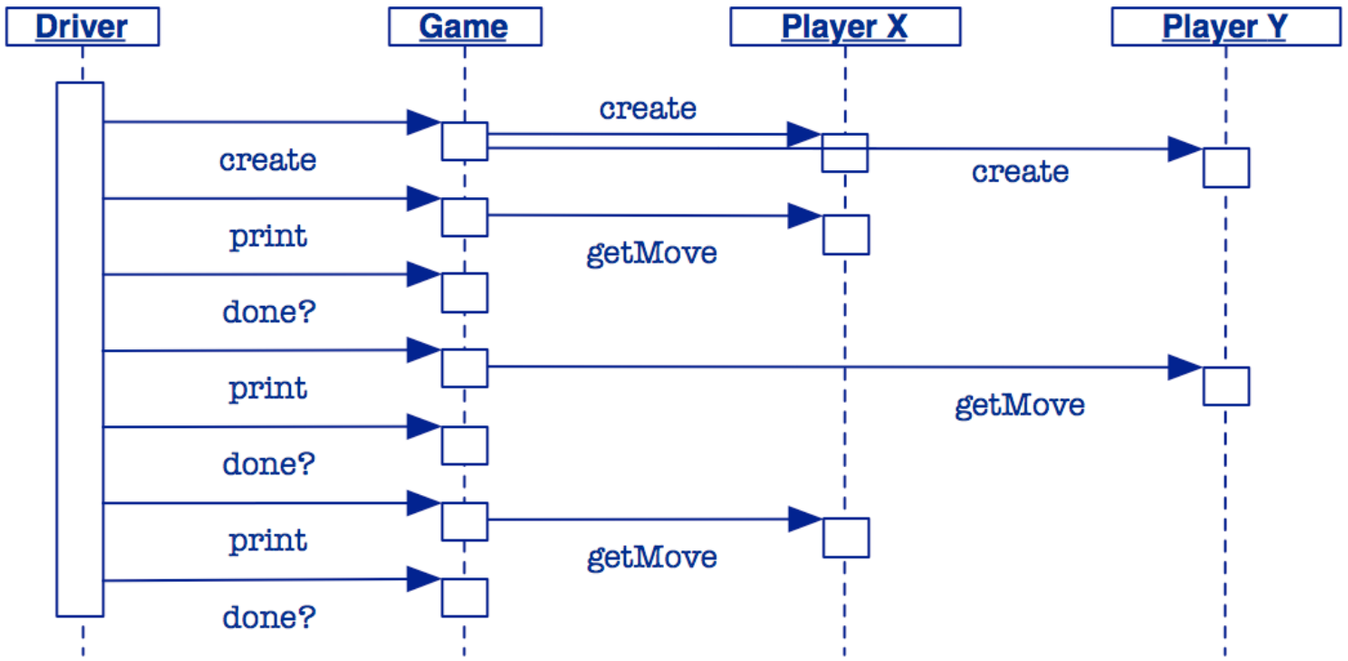
Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Scenarios

A scenario describes a typical sequence of interactions:



Are there other equally valid scenarios for this problem?

Version 0 — skeleton

Our first version does very little!

```
class GameDriver {
   static public void main(String args[]) {
      TicTacToe game = new TicTacToe();
      do { System.out.print(game); }
      while(game.notOver());
   }
public class TicTacToe {
   public boolean notOver() { return false; }
   public String toString() { return("TicTacToe\n");}
}
```

? How do you iteratively “grow” a program?

– Always have a running version of your program.

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Version 1 — game state

We will use chess notation to access the game state

- Columns ‘a’ through ‘c’
- Rows ‘1’ through ‘3’

? How do we decide on the right interface?

- First write some tests!

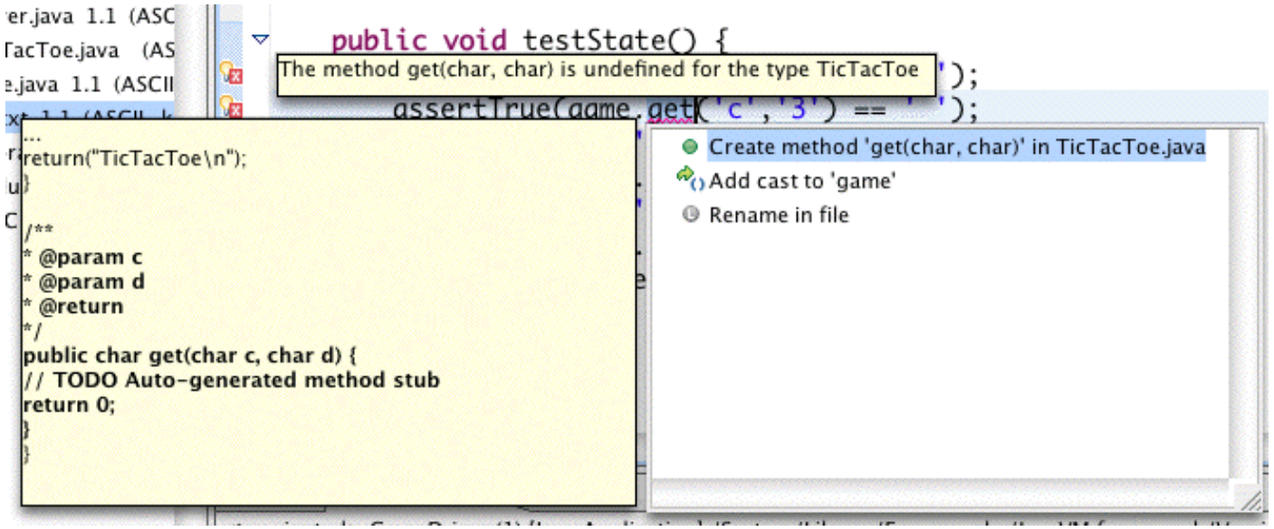
Test-first development

```
public class TicTacToeTest {
    private TicTacToe game;

    @Before public void setUp() {
        super.setUp();
        game = new TicTacToe();
    }

    @Test public void testState() {
        assertTrue(game.get('a','1') == ' ');
        assertTrue(game.get('c','3') == ' ');
        game.set('c','3','X');
        assertTrue(game.get('c','3') == 'X');
        game.set('c','3',' ');
        assertTrue(game.get('c','3') == ' ');
        assertFalse(game.inRange('d','4'));
    }
}
```

Generating methods



Test-first programming can drive the development of the class interface …

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Representing game state

```
public class TicTacToe {
   private char[][] gameState;
   public TicTacToe() {
      gameState = new char[3][3];
      for (char col='a'; col <='c'; col++)
         for (char row='1'; row<='3'; row++)
            this.set(col,row,' ');
   }
...
```

Checking pre-conditions

set() and get() translate from chess notation to array indices.

```
public void set(char col, char row, char mark) {
   assert(inRange(col, row));  // NB: precondition
   gameState[col-'a'][row-'1'] = mark;
}
public char get(char col, char row) {
   assert(inRange(col, row));
   return gameState[col-'a'][row-'1'];
}
public boolean inRange(char col, char row) {
   return (('a'<=col) && (col<='c')
      && ('1'<=row) && (row<='3'));
}
```

Printing the State

By re-implementing TicTacToe.toString(), we can view the state of the game:

```
3     |     |
   ---+---+---
2     |     |
   ---+---+---
1     |     |
    a    b    c
```

? How do you make an object printable?

- Override Object.toString()

TicTacToe.toString()

Use a StringBuilder (not a String) to build up the representation:

```
public String toString() {
   StringBuffer rep = new StringBuilder();
   for (char row='3'; row>='1'; row--) {
      rep.append(row);
      rep.append("   ");
      for (char col='a'; col <='c'; col++) { ... }
      ...
   }
   rep.append("   a   b   c\n");
   return(rep.toString());
}
```

PS: newer version of Java have improved on using String directly

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Version 2 — adding game logic

We will:

Add test scenarios

Add Player class

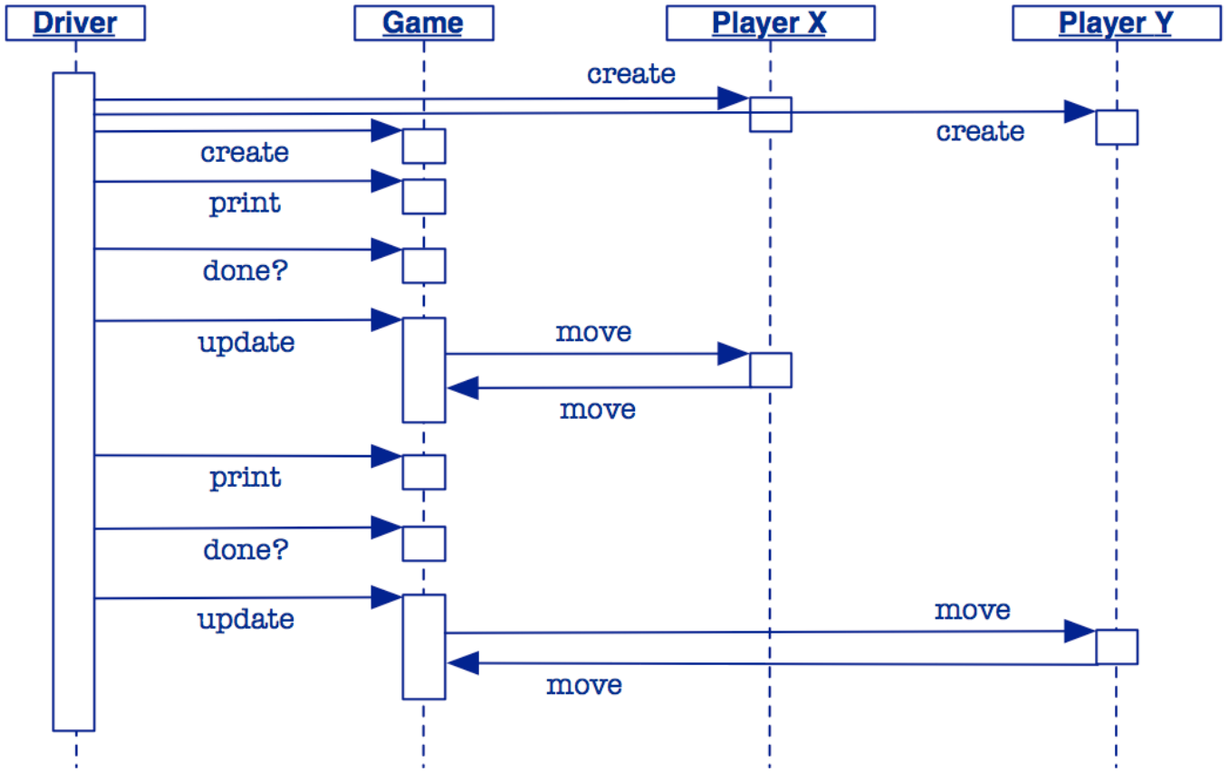
Add methods to make moves, test for winning

Refining the interactions

We will want both real and test Players, *so the Driver should create them.*

Updating the Game and printing it *should be separate operations.*

The Game should ask the Player to make a move, and *then the Player will attempt to do so.*



Testing scenarios

Our test scenarios will play and test scripted games

```
@Test public void testXWinDiagonal() {
    checkGame("a1\nb2\nc3\n", "b1\nc1\n", "X", 4);
}
// more tests …

public void checkGame(String Xmoves, String Omoves,
        String winner, int squaresLeft) {
    Player X = new Player('X', Xmoves);// a scripted player
    Player O = new Player('O', Omoves);
    TicTacToe game = new TicTacToe(X, O);
    GameDriver.playGame(game);
    assertTrue(game.winner().name().equals(winner));
    assertTrue(game.squaresLeft() == squaresLeft);
}
```

Running the test cases

```
3     |    |
  ---+---+---
2     |    |
  ---+---+---
1     |    |
   a    b    c
Player X moves: X at a1
3     |    |
  ---+---+---
2     |    |
  ---+---+---
1  X  |    |
   a    b    c
...
```

```
Player O moves: O at c1
3     |    |
  ---+---+---
2     | X  |
  ---+---+---
1  X  | O  | O
   a    b    c
Player X moves: X at c3
3     |    | X
  ---+---+---
2     | X  |
  ---+---+---
1  X  | O  | O
   a    b    c
game over!
```

The Player

We use different constructors to make real or test Players:

```
public class Player {
    private final char mark;
    private final BufferedReader in;
```

A real player reads from the standard input stream:

```
    public Player(char mark) {
        this(mark, new BufferedReader(
                new InputStreamReader(System.in)
        ));
    }
```

This constructor just calls another one ...

...

Player constructors ...

But a Player can be constructed that reads its moves from any input buffer:

```
protected Player(char initMark, BufferedReader initIn) {
        mark = initMark;
        in = initIn;
}
```

This constructor is not intended to be called directly.

...

Player constructors ...

A test Player gets its input from a String buffer:

```
public Player(char mark, String moves) {
    this(mark, new BufferedReader(
            new StringReader(moves)
    ));
}
```

The default constructor returns a dummy Player representing “nobody”

```
public Player() { this(' '); }
```

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Tic Tac Toe Contracts

Explicit invariants:

- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

Implicit invariants:

- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

Contracts:

- the current player may make a move, if the invariants are respected

Encoding the contract

We must introduce state variables to implement the contracts

```
public class TicTacToe {
   static final int X = 0;                    // constants
   static final int O = 1;
   private char[][] gameState;
   private Player winner = new Player();      // = nobody
   private Player[] player;
   private int turn = X;                      // initial turn
   private int squaresLeft = 9;
...
```

Supporting test Players

The Game no longer instantiates the Players, but accepts them as constructor arguments:

```
public TicTacToe(Player playerX, Player playerO)
{   // ...
    player = new Player[2];
    player[X] = playerX;
    player[O] = playerO;
}
```

Invariants

These conditions may seem obvious, which is exactly why they should be checked ...

```
private boolean invariant() {
   return (turn == X || turn == O)
      && ( this.notOver()
         || this.winner() == player[X]
         || this.winner() == player[O])
      && (squaresLeft < 9          // else, initially:
         || turn == X && this.winner().isNobody());
}
```

Assertions and tests often tell us what methods should be implemented, and whether they should be public or private.

Delegating Responsibilities

When Driver updates the Game, the Game just asks the Player to make a move:

```
public void update() throws IOException {
   player[turn].move(this);
}
```

Note that the Driver may not do this directly!

…

Delegating Responsibilities ...

The Player, in turn, calls the Game's move() method:

```
public void move(char col, char row, char mark) {
    assert(notOver());
    assert(inRange(col, row));
    assert(get(col, row) == ' ');
    System.out.println(mark + " at " + col + row);
    this.set(col, row, mark);
    this.squaresLeft--;
    this.swapTurn();
    this.checkWinner();
    assert(invariant());
}
```

Small Methods

Introduce methods that make the intent of your code clear.

```
public boolean notOver() {
   return this.winner().isNobody()
                && this.squaresLeft() > 0;
}
private void swapTurn() {
   turn = (turn == X) ? O : X;
}
```

Well-named variables and methods typically eliminate the need for explanatory comments!

Use comments to explain <u>non obvious</u> design, algorithmic or implementation choices

Accessor Methods

Accessor methods protect clients from changes in implementation:

```
public Player winner() {
   return winner;
}
public int squaresLeft() {
   return this.squaresLeft;
}
```

? When should instance variables be public?

– Almost never! Declare public accessor methods instead.

getters and setters in Java

Accessors in Java are known as “getters” and “setters”.

- Accessors for a variable x should normally be called getx() and setx()

Frameworks such as EJB depend on this convention!

Code Smells — TicTacToe.checkWinner()

Duplicated code stinks!

How can we clean it up?

```
private void checkWinner()
   {
  char player;
  for (char row='3'; row>='1'; row--) {
    player = this.get('a',row);
    if (player == this.get('b',row)
      && player == this.get('c',row)) {
      this.setWinner(player);
      return;
    }
  }
```

```
  for (char col='a'; col <='c'; col++) {
    player = this.get(col,'1');
    if (player == this.get(col,'2')
      && player == this.get(col,'3')) {
      this.setWinner(player);
      return;
    }
  }
  player = this.get('b','2');
  if (player == this.get('a','1')
    && player == this.get('c','3')) {
      this.setWinner(player);
      return;
  }
  if (player == this.get('a','3')
    && player == this.get('c','1')) {
      this.setWinner(player);
      return;
  }
}
```

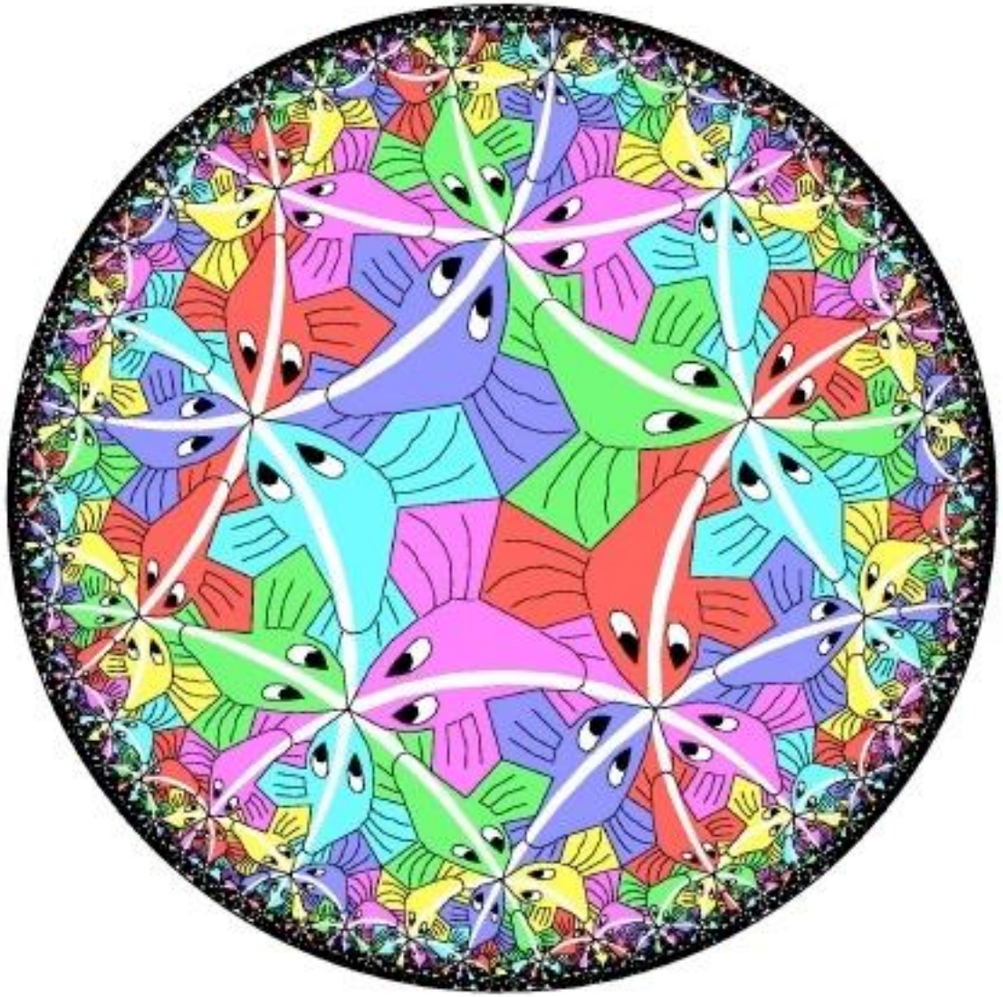
GameDriver

In order to run test games, we separated Player instantiation from Game playing:

```
public class GameDriver {
   public static void main(String args[]) {
      try {
         Player X = new Player('X');
         Player O = new Player('O');
         TicTacToe game = new TicTacToe(X, O);
         playGame(game);
      } catch (AssertionException err) {
         ...
      }
   }
```

How can we make test scenarios play silently?

Patterns



Bit of history…

Christoffer Alexander

- “The Timeless Way of Building”, Christoffer Alexander, Oxford University Press, 1979, ISBN 0195024028
- Structure of the book is magnificent (cfr writing your master dissertation…)
  - Christmass is close ;-)

More advanced than what computer science uses

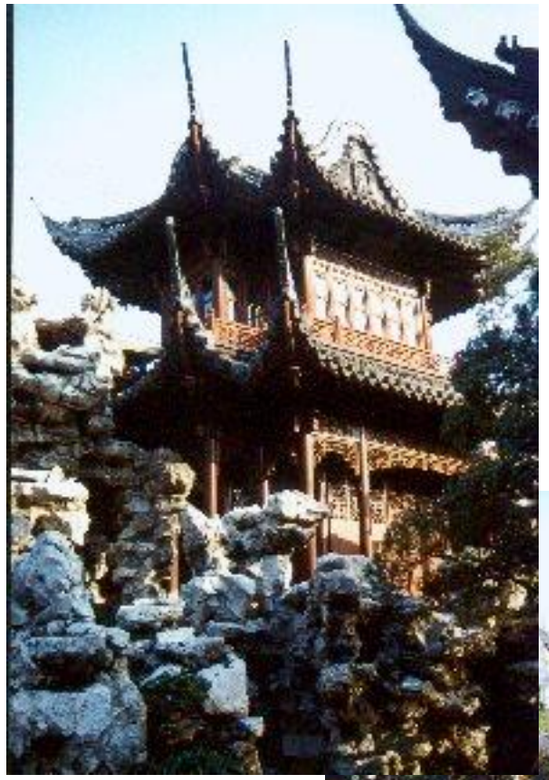
- only the simple parts got mainstream

Alexander's patterns

"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 doing it the same way twice"

- Alexander uses this as part of the solution to capture the "quality without a name"

Illustrating Recurring Patterns...



Essential Elements in a Pattern

Pattern name

- Increase of design vocabulary

Problem description

- When to apply it, in what context to use it

Solution description (generic !)

- The elements that make up the design, their relationships, responsibilities, and collaborations

Consequences

- Results and trade-offs of applying the pattern

GRASP Patterns

guiding principles to help us assign responsibilities

GRASP "Patterns" – guidelines

- Controller
- Creator
- Information Expert
- Low Coupling

Hs 17

- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Hs 25

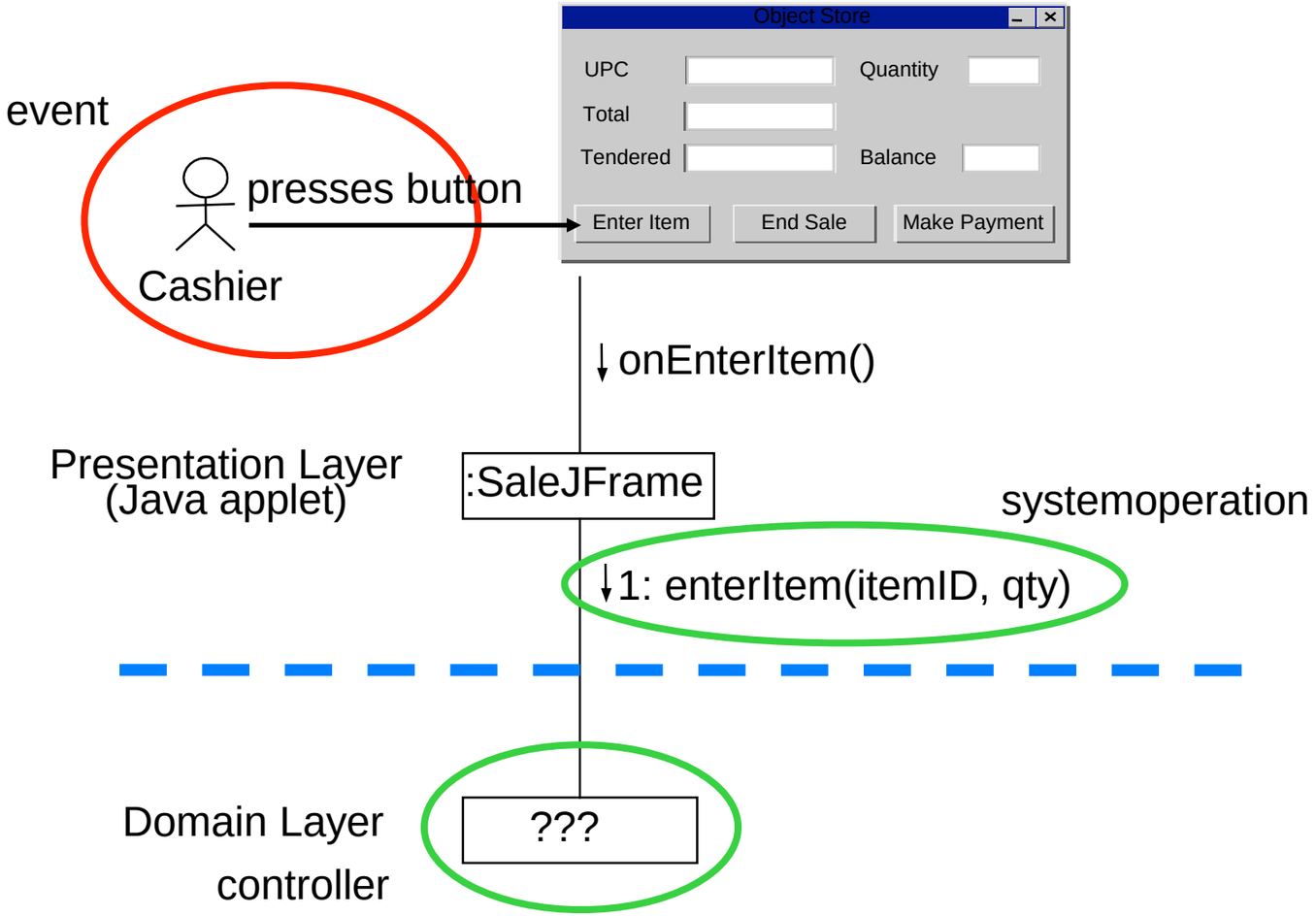
GRASP Patterns

We already saw High Coupling and Low Cohesion before

Let’s look at some more GRASP patterns now…

1. Controller Pattern

Who is responsible for handling Systemoperations ?



Controller Pattern

Pattern	Controller
Problem	Who is responsible for handling system events ?
Solution	Assign the responsibility to a class C representing one of the following choices: • C is a *facade controller*: it represents the overall system, a root object, the device that runs the software, or a major subsystem. • C is a *use case or session controller*: it represents an artificial objects (see *Pure Fabrication* pattern) that handles all events from a use case or session

System operations and System events

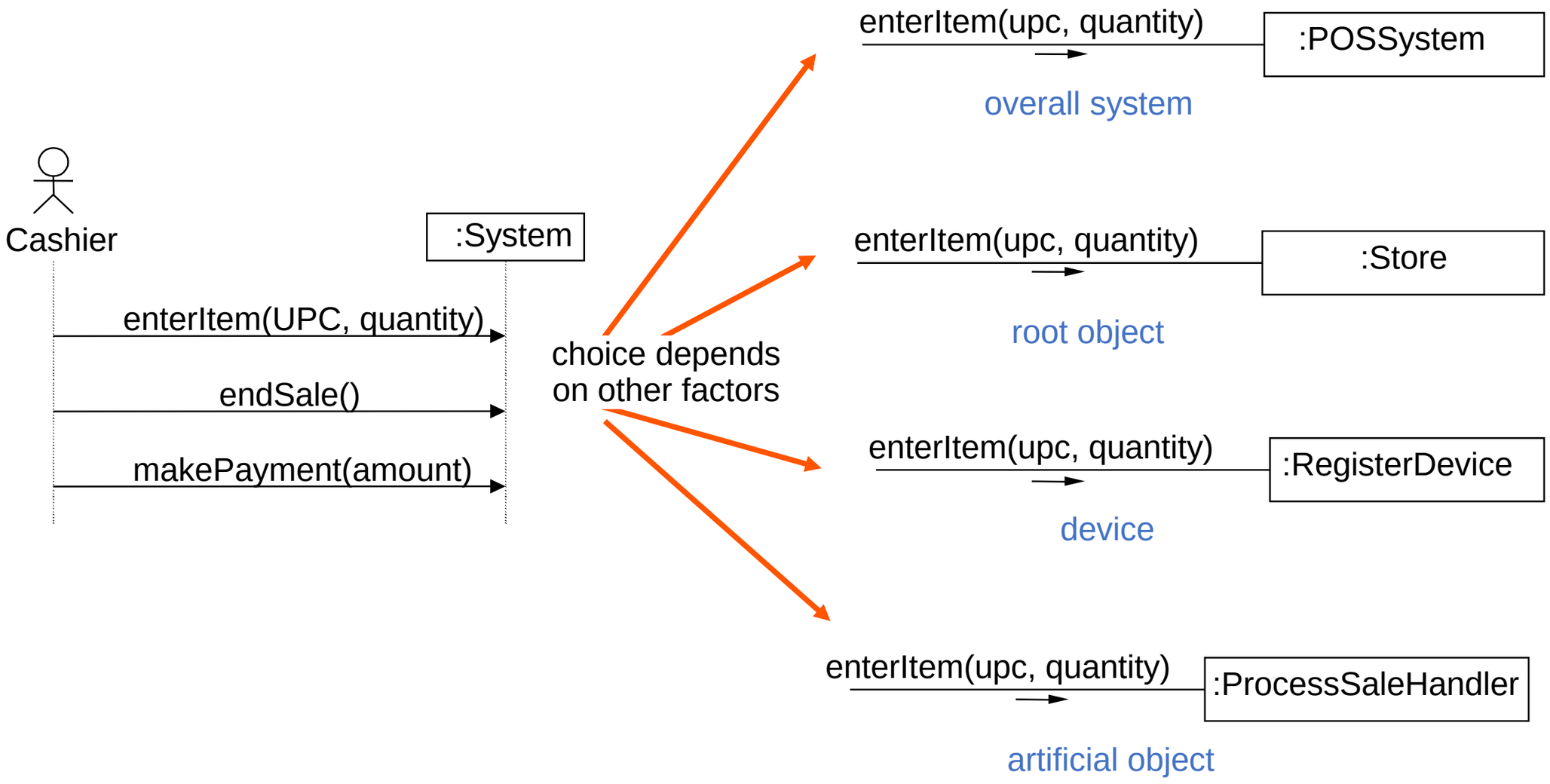
From analysis to design:

- Analysis: can group system operations in a conceptual “System” class
- Design: give responsibility for processing system operations to controller classes

Controller classes are not part of the User Interface

Model-View-Controller (MVC)

Who controls System events?



Controller Pattern: Guidelines

Limit the responsibility to “control and coordination”

- Controller = delegation pattern
- delegate real work to real objects
- Common mistake: fat controllers with too much behavior

Only support a limited number of events in Facade controllers

Controller Pattern: Use Case Controller Guidelines

Use Case (UC) controllers

- consider when too much coupling and not enough cohesion in other controllers (factor system events)
- Treat all UC events in the same controller class
- Allow control on the order of events
- Keep information on state of UC (statefull session)

Controller Pattern: Problems and Solutions

"Bloated" controllers

- symptoms
  - a single controller handling all system events
  - controller not delegating work
  - controller with many attributes, with system information, with duplicated information
- solutions
  - add Use Case controllers
  - design controllers that delegate tasks

Controller Pattern: Advantages

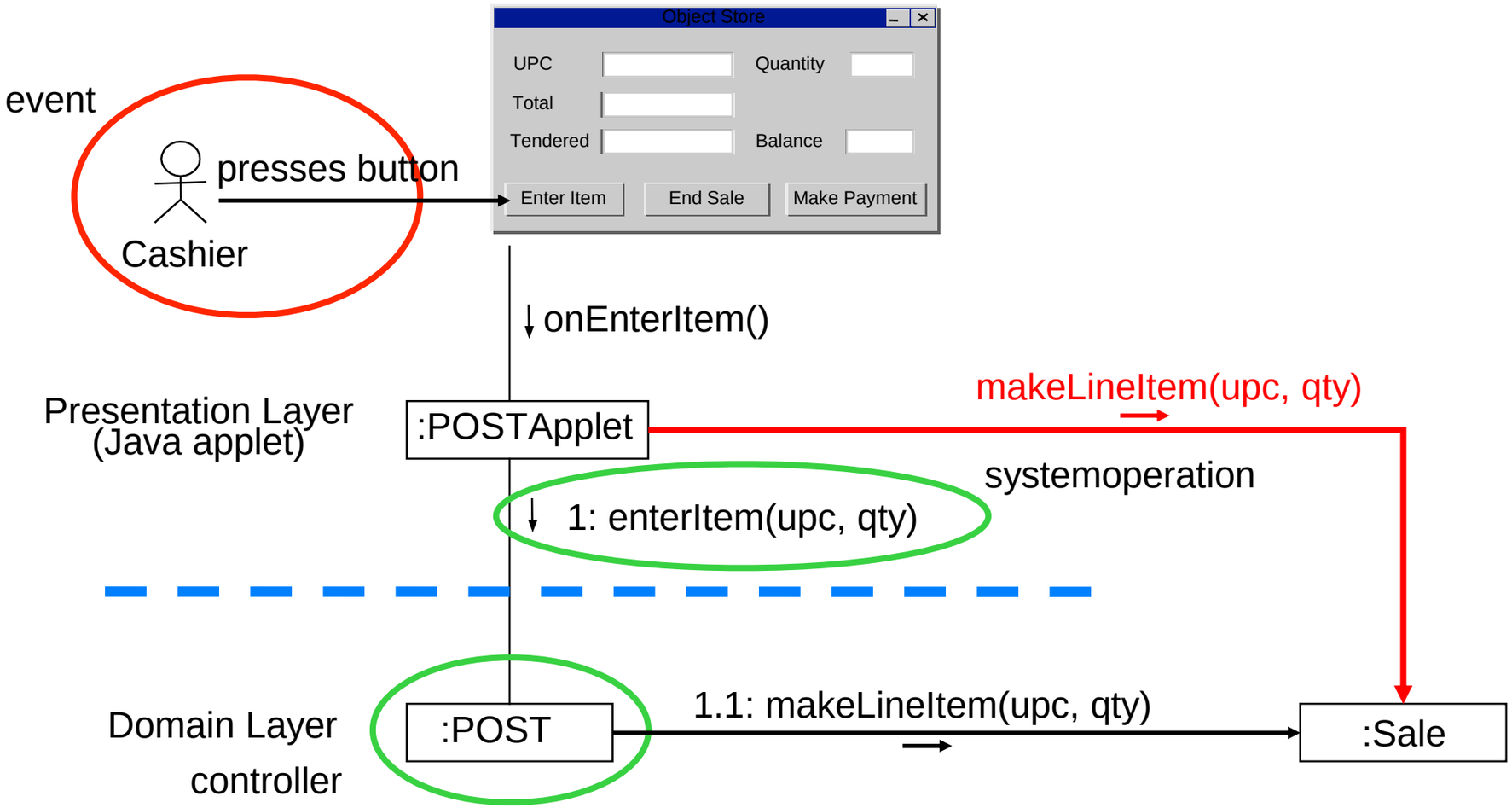
Increased potential for reuse

- domain-level processes handled by domain layer
- decouple GUI from domain level !
- Different GUI or different ways to access the domain level

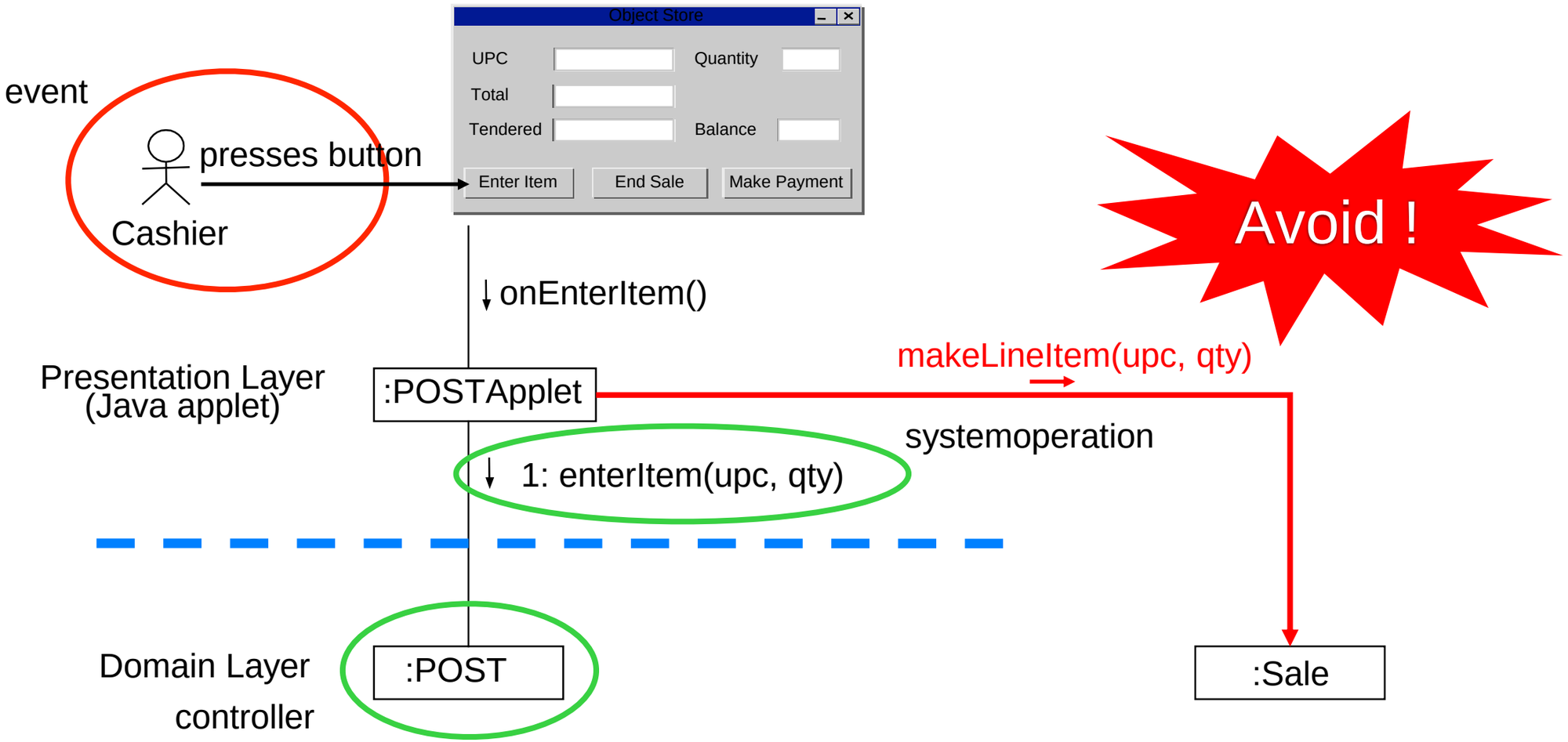
Reason about the state of the use case

- guarantee sequence of system operations

Example applying Controller Pattern



Example: **incorrect!**

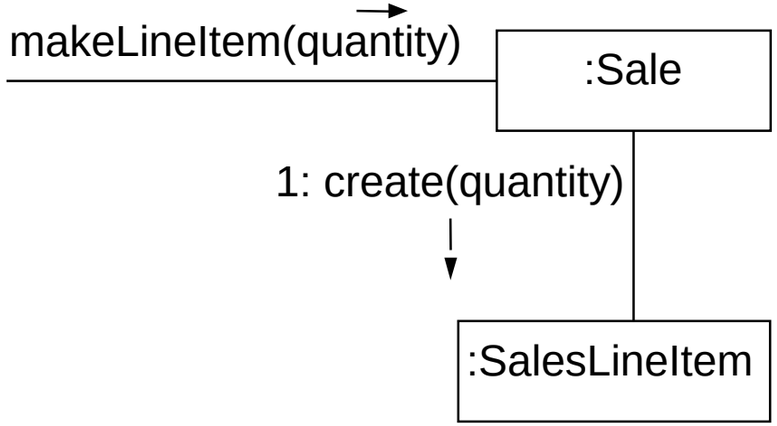


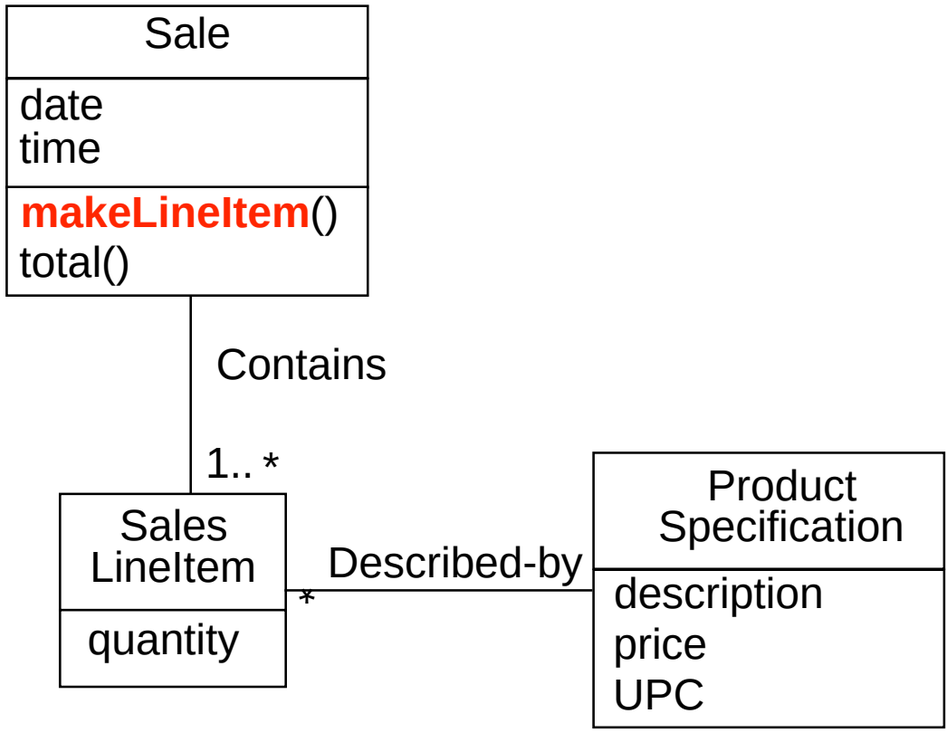
2. Creator Pattern

Pattern	Creator
Problem	Who is responsible for creating instances of classes ?
Solution	Assign a class B to create instances of a class A if: • B is a composite of A objects (*composition/aggregation*) • B contains A objects (*contains*) • B holds instances of A objects (*records*) • B closely collaborates with A objects • B has the information needed for creating A objects

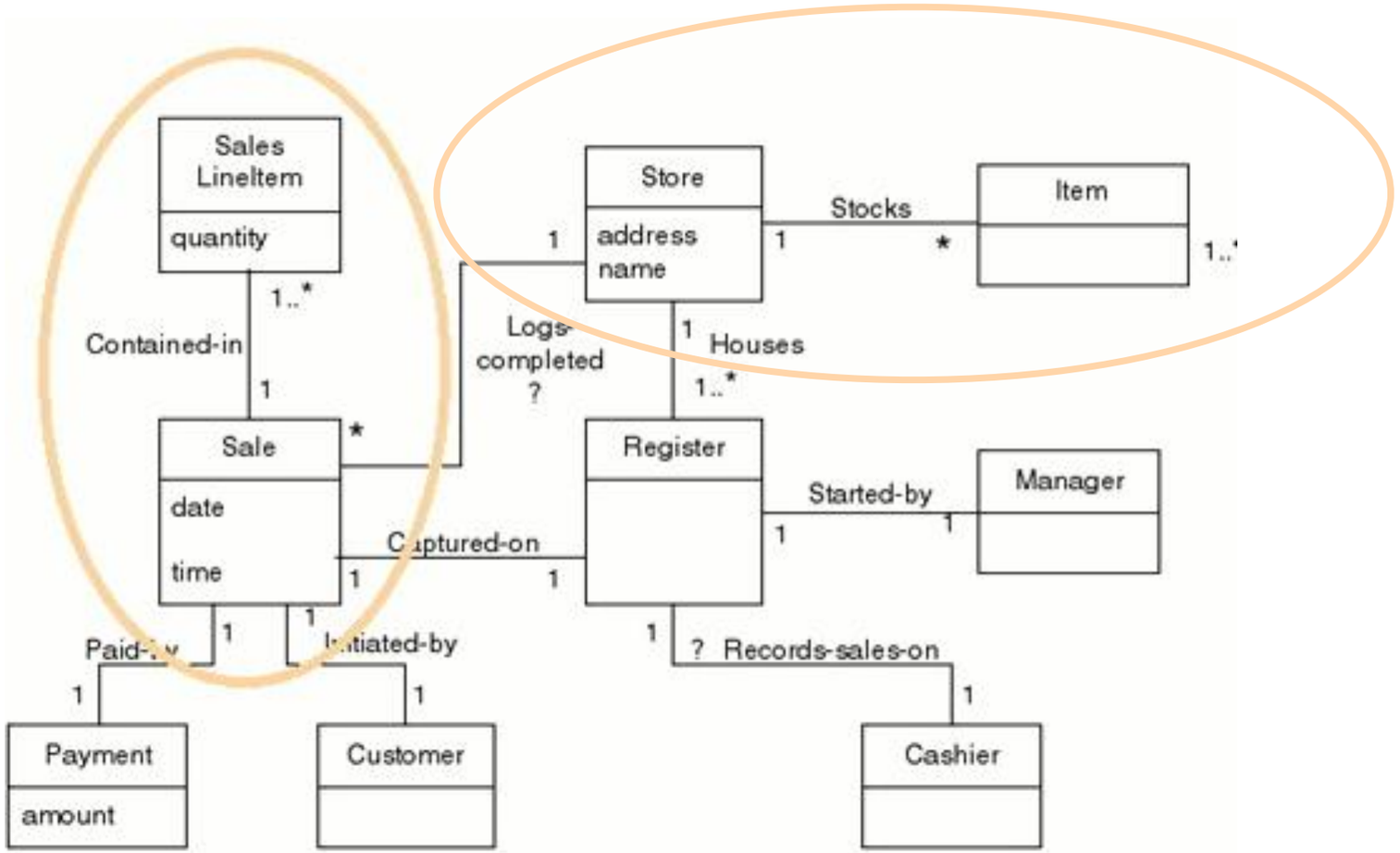
Creator Pattern: example

Creation of “SalesLineItem” instances





Creator Pattern: Inspiration from the Domain Model



3. Information Expert Pattern

A very basic principle of responsibility assignment

(cfr Responsibility Driven questions seen earlier)

Assign a responsibility to the object that has the information necessary to fulfill it -the information expert

- "That which has the information, does the work"
- Related to the principle of "low coupling"
  - ⇨ Localize work

Expert Pattern

Pattern	(Information) Expert
Problem	What is the basic principle to assign responsibilities to objects ?
Solution	Assign responsibility to the class that has the information to fulfill it (the information expert)

Expert Pattern: remarks

Real-world analogy

- who predicts gains/losses in a company?
  - the person with access to the date (Chief Financial Officer)

Needed information to work out ‘responsibility’
=> spread over different objects

- “partial” experts that collaborate to obtain global information (interaction is required)

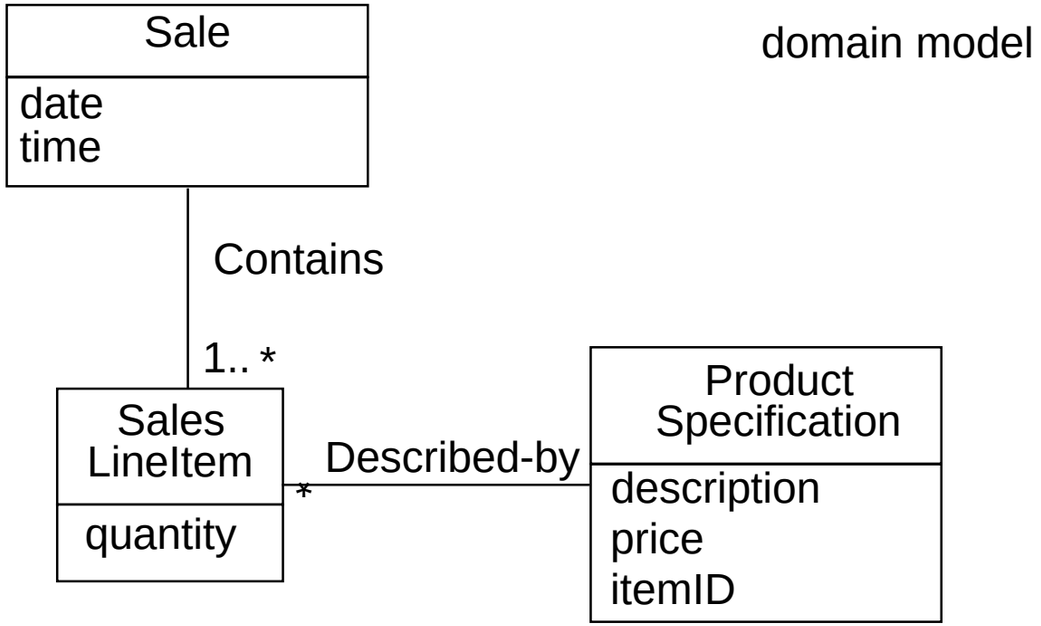
Not necessarily the best solution (e.g. database access)

- See low coupling & high cohesion

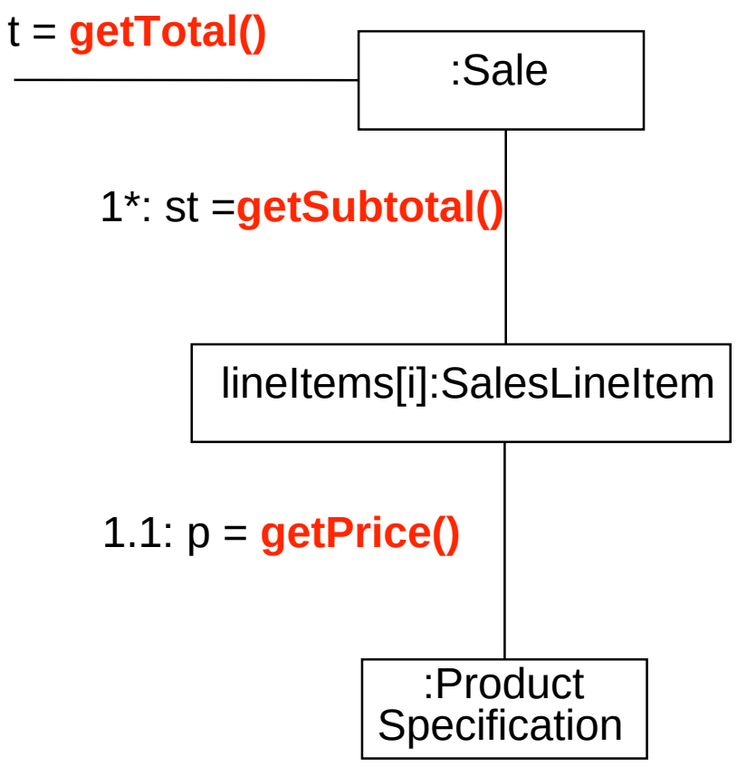
Expert Pattern: example 1

Example: Who is responsible for knowing the total of a “Sale”?

Who possesses the information?

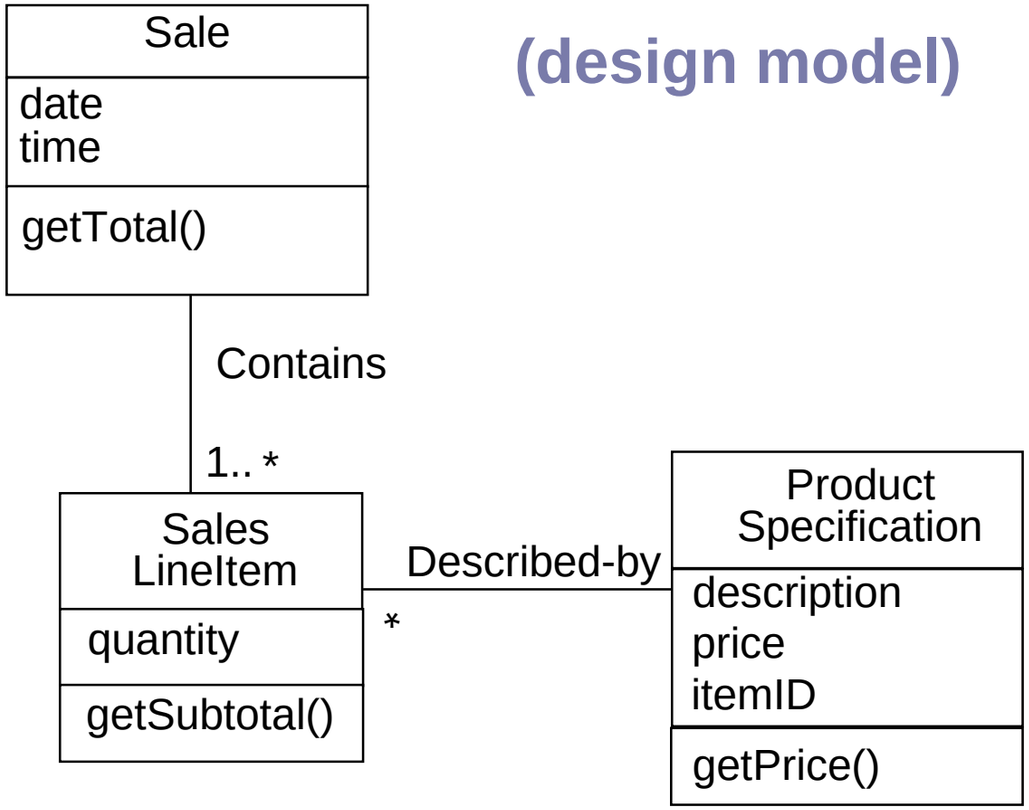


Expert Pattern: example 1



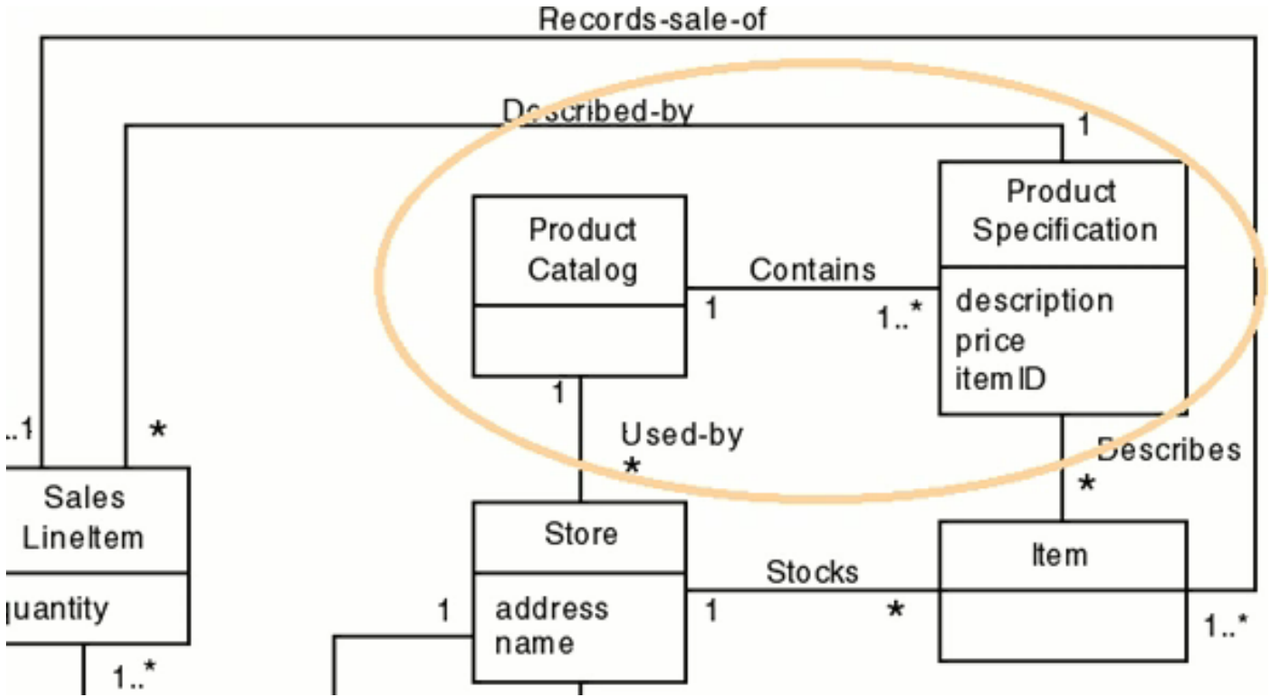
class diagram

(design model)

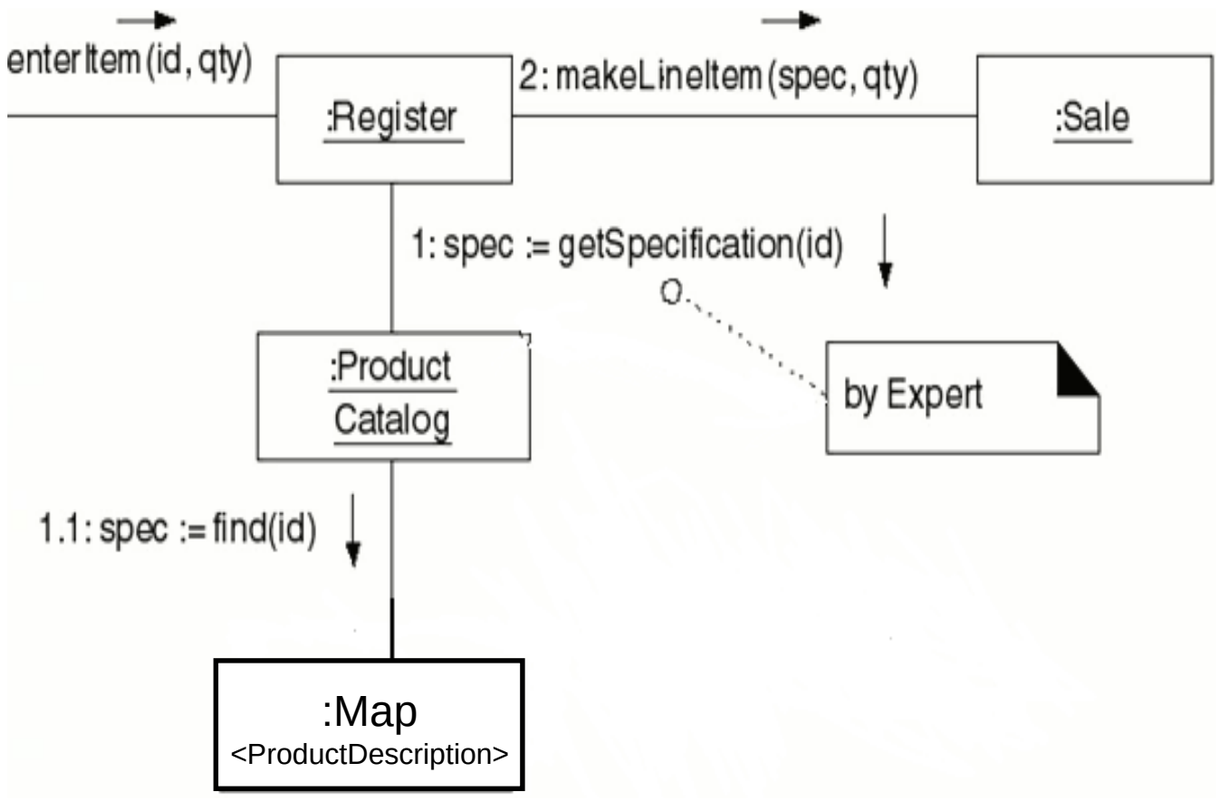


Expert Pattern: Example 2

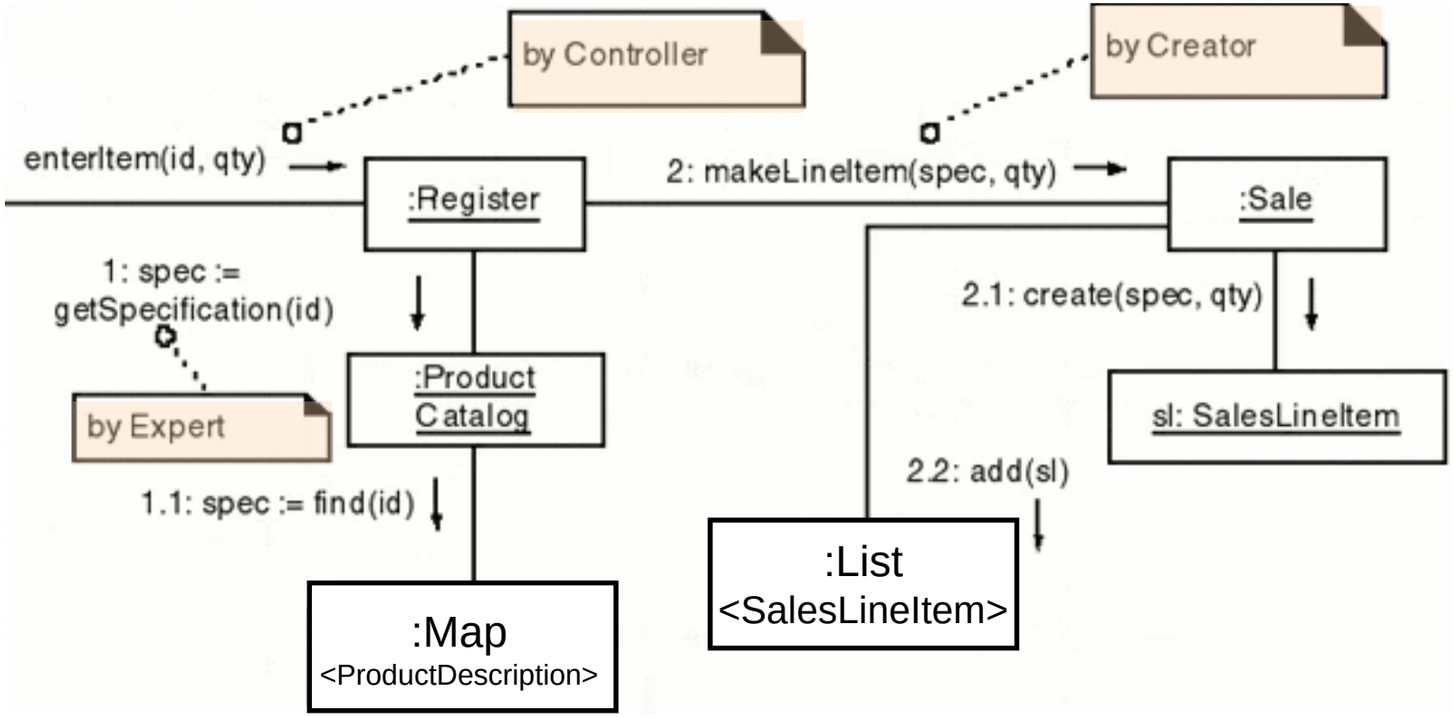
What object should be responsible for knowing ProductSpecifications, given a key?
Take inspiration from the domain model



Applying Information Expert



Design for “enterItem”: 3 patterns applied



GRASP Patterns

guiding principles to help us assign responsibilities

GRASP “Patterns” – guidelines

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

Hs 17

Hs 25

6. Polymorphism

Pattern	Polymorphism
Problem	How handle alternatives based on type? How to create pluggable software components?
Solution	When related alternatives or behaviours vary by type (class), assign responsibility for the behavior -using polymorphic operations- to the types for which the behavior varies.

Note: Not really a pattern but basic OO principle !

Example

procedural approach:
passing an int and using switch to decide which behavior to execute based on that int

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
  {
  switch ( aCommand )
    {
        case EAknSoftkeyExit:
        case EAknSoftkeyBack:
        case EEikCmdExit:
            { Exit();  break; }

        // Play command is selected
        case EVideoCmdAppPlay:
            { DoPlayL(); break; }

         // Stop command is selected
        case EVideoCmdAppStop:
            { DoStopL(); break; }

        // Pause command is selected
        case EVideoCmdAppPause:
            { DoPauseL(); break; }

        // DocPlay command is selected
        case EVideoCmdAppDocPlay:
            { DoDocPlayL(); break; }
        ......
```

Nokia S60 mobile video player 3gpp source code
http://www.codeforge.com/article/192637

Solution: Replace case by Polymorphism

OO approach:
passing an object and using polymorphism to select behavior to execute

```
void CVideoAppUi::HandleCommandL(Command aCommand)
  {
        aCommand.execute();
       }
```

Create a Command class hierarchy, consisting of a (probably) abstract class AbstractCommand, and subclasses for every command supported. Implement execute on each of these classes:

- virtual void AbstractCommand::execute() = 0;
- virtual void PlayCommand::execute() { ... do play command ...};
- virtual void StopCommand::execute() { ... do stop command ...};
- virtual void PauseCommand::execute() { ... do pause command ...};
- virtual void DocPlayCommand::execute() { ... do docplay command ...};
- virtual void FileInfoCommand::execute() { ... do file info command ...};

7. Pure Fabrication Pattern

Pattern	Pure Fabrication
Problem	What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
Solution	Assign a cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept but is purely imaginary and fabricated to obtain a pure design with high cohesion and low coupling.

Pure Fabrication Pattern

Where no appropriate class is present: invent one

- Even if the class does not represent a problem domain concept
- “pure fabrication” = making something up: do when we’re desperate!

This is a compromise that often has to be made to preserve cohesion and low coupling

- Remember: the software is not designed to simulate the domain, but operate in it
- The software does not always have to be identical to the real world
  - Domain Model ≠ Design model

Pure Fabrication Example

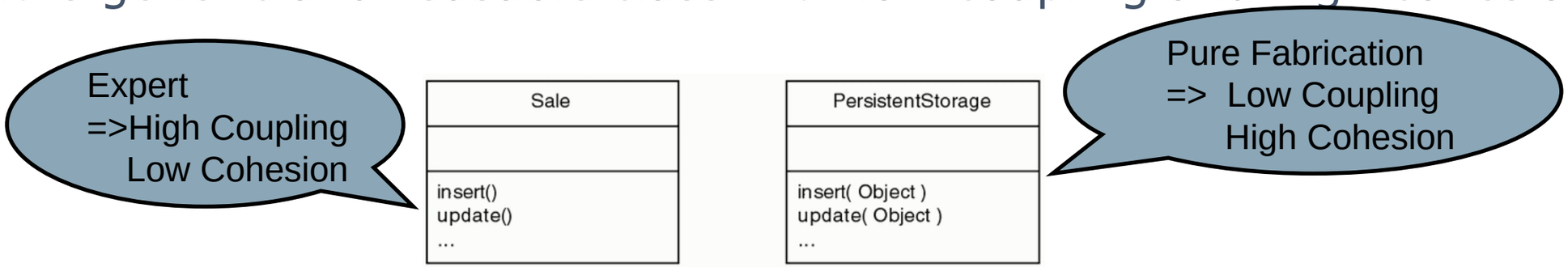
Suppose Sale instances need to be saved in a database

Option 1: assign this to the Sale class itself (Expert pattern)

- Implications of this solution:
  - auxiliary database-operations need to be added as well
  - coupling with particular database connection class
  - saving objects in a database is a general service

Option 2: create PersistentStorage class

- Result is generic and reusable class with low coupling and high cohesion



8. Indirection Pattern

Pattern	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. This intermediary creates an indirection between the other components.

Indirection Pattern

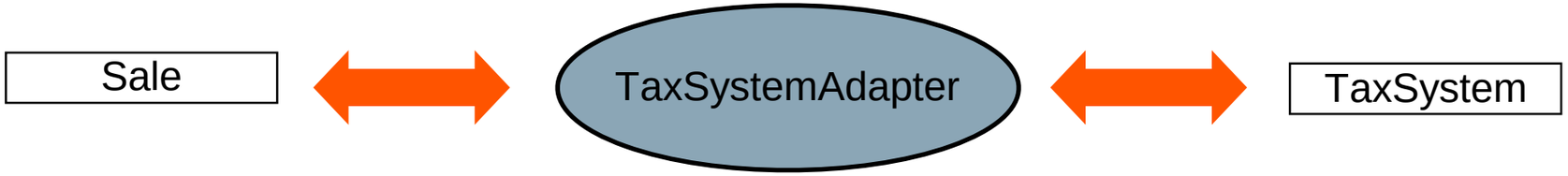
A common mechanism to reduce coupling

Assign responsibility to an intermediate object to decouple two components

- coupling between two classes of different subsystems can introduce maintenance problems

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

- A large number of design patterns are special cases of indirection (Adapter, Facade, Observer)



9. Protected Variations Pattern

Pattern	Protected Variations
Problem	How to design objects, subsystems, and systems so that the variations or instability of these elements does not have an undesirable impact on other elements ?
Solution	Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Protected Variations – example

Video game companies make money by creating a game engine

- many games use the same engine
- what if a game is to be ported to another console ???
  - a wrapper object will have to delegate 3D graphics drawing to different console-level commands
  - the wrapper is simpler to change than the entire game and all of its facets

Wrapping the component in a stable interface means that when variations occur, only the wrapper class need be changed

- In other words, changes remain localized
- The impact of changes is controlled

FUNDAMENTAL PRINCIPLE IN SW DESIGN

Protected Variations – Example

Open DataBase Connectivity (ODBC/JDBC)

- These are packages that allow applications to access databases in a DB-independent way
  - In spite of the fact that databases all use slightly different methods of communication
  - It is possible due to an implementation of Protected Variations
- Users write code to use a generic interface
  - An adapter converts generic method calls to DB and vice versa

Conclusion

Always try to apply and balance basic OO Design Principles

- Minimize Coupling
- Increase Cohesion
- Distribute Responsibilities

Use and learn from established sources of information

- Responsibility Driven Design
- GRASP patterns
  - Design Patterns: see later

References

Rebecca Wirfs-Brock, Alan McKean, Object Design — Roles, Responsibilities and Collaborations, Addison-Wesley, 2003.

http://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf

Craig Larman, Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.), Prentice Hall, 2005.

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.