Ontwerp van SoftwareSystemen

2 Basic OO Design

Roel Wuyts

OSS 2013-2014

Basic OO Design Principles

- No matter whether you use forward engineering or re-engineering: 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
- Let's have a look at concrete code bases for examples...

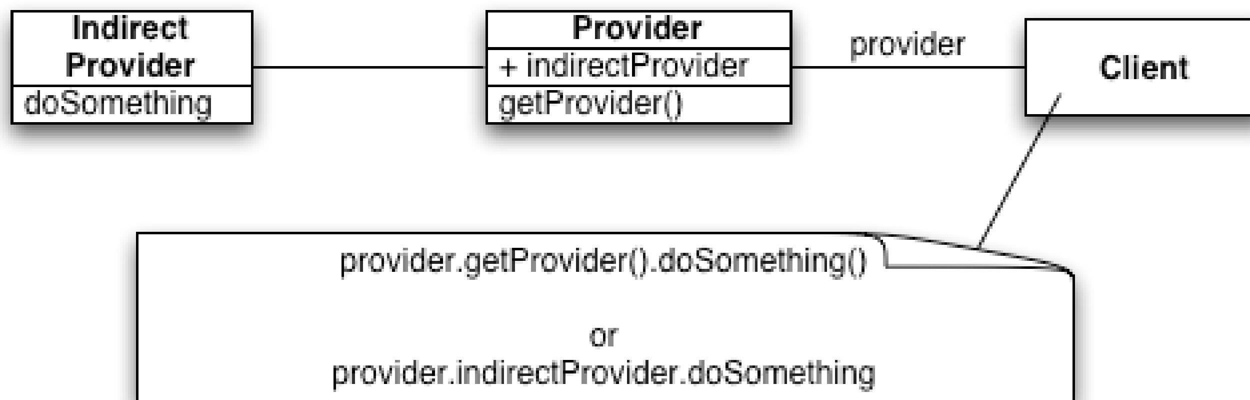
© imec restricted 2007

To remember...

- You write code once,
- but it is read many times more.
  - By you
    - 2 years later...
  - By other people
- So your code better be readable!

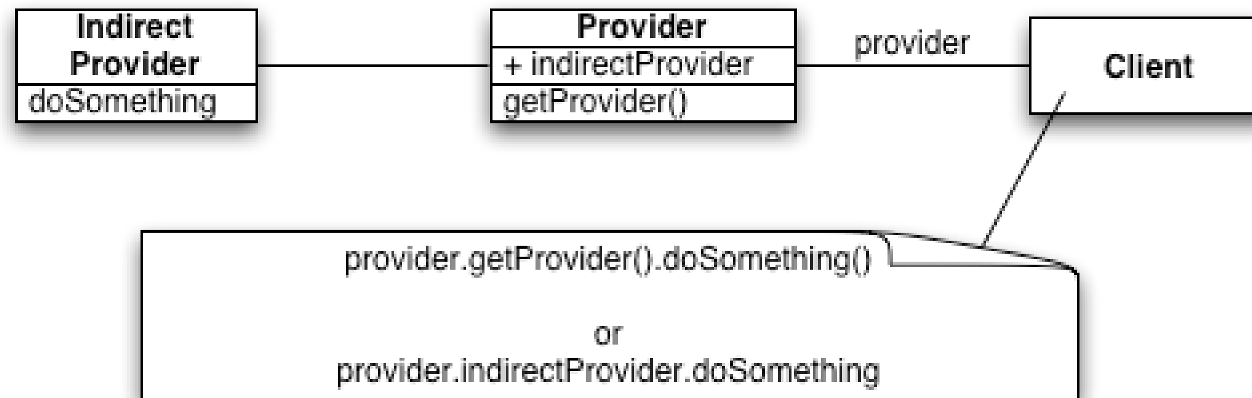
© imec restricted 2007

Example 1



© imec restricted 2007

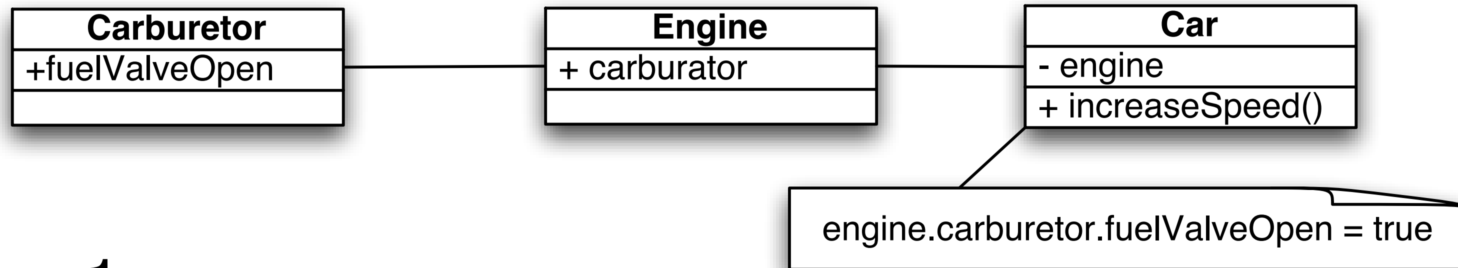
Why is this bad ?



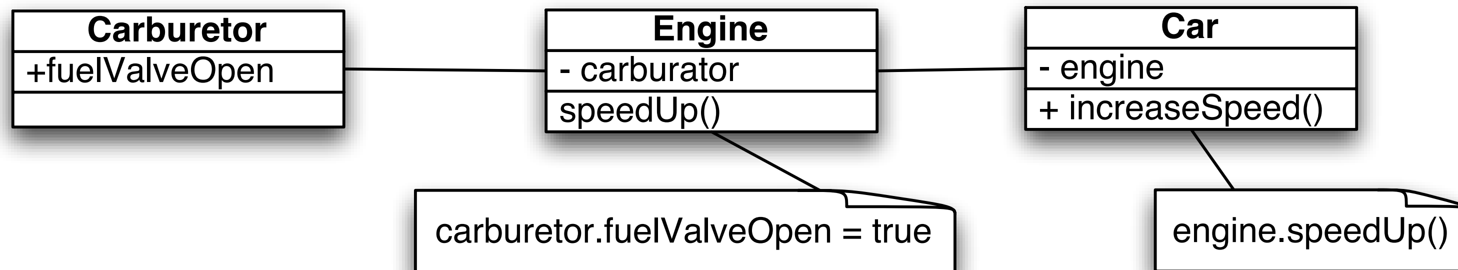
- Client knows how Provider is implemented
  - has to know that it uses an IndirectProvider
    - uses the interface of Provider as well as of IndirectProvider
  - Client and IndirectProvider are strongly *coupled* !
    - Client has to use them together
    - Changing either Provider or IndirectProvider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

© imec restricted 2007

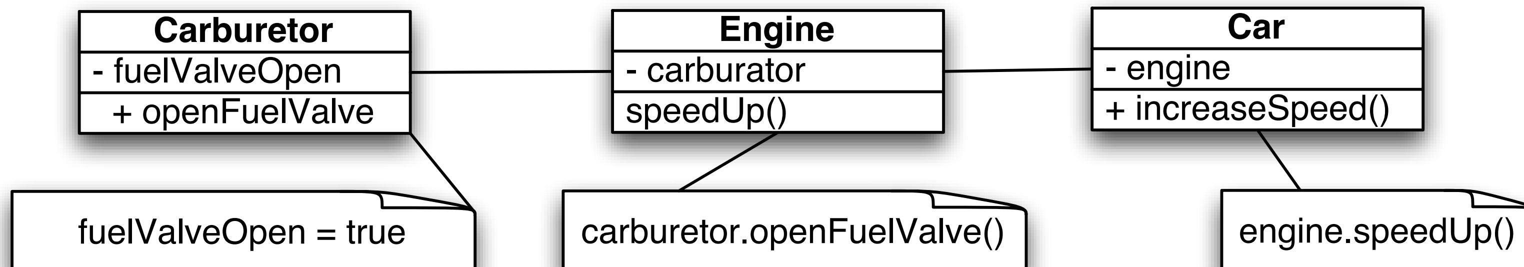
Reducing the Coupling



Step 1



Step 2



© imec restricted 2007

Reducing Coupling impacts the design

- The interfaces of the classes become more clear
  - a method ‘speedUp()’ makes perfect sense
- Allows for more opportunity for reuse
  - A subclass of Engine, “ElectricalEngine”, might not need a Carburetor at all
    - This is transparent for Car

© imec restricted 2007

"Law of Demeter"

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

or, more formally:

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

© imec restricted 2007

Example 2

```
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; }

        // File info command is selected
         case EVideoCmdAppDocFileInfo:
           { DoGetFileInfoL(); break; }
......
```

[Nokia S60 mobile video player 3gpp source code](http://www.codeforge.com/article/192637)
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 scatteredin several places
    - when the system evolves these places have to be updated, but are easy to miss

© imec restricted 2007

Solution: Replace case by Polymorphism

```
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 ...};
```


© imec restricted 2007

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 ?

© imec restricted 2007

Example 3: Duplicated code

- Occurs a lot
  - Range of code duplication: roughly 10% to 25% !
  - 19% in X Window System
  - 68% of Java Buffer Library (JDK 1.4.1)

© imec restricted 2007

Problems with duplication

- Errors get spread
  - fixes do not...
- Evolution of code is not reflected everywhere
  - some places are forgotten and do not get updated
- Code bloat: code gets much bigger
  - since no sharing

Where can we find duplication?

- In the same class
  - several methods that repeat a number of instructions
- Between siblings
  - two classes that share a common superclass
  - methods in siblings can repeat a number of instructions
- Between unrelated classes
  - classes not in a hierarchy can still repeat the same sets of instructions

© imec restricted 2007

Removing Duplicated Code

- In the same class
  - Extract Method
- Between two sibling subclasses
  - Extract Method
  - Push identical methods up to common superclass
  - Form Template Method
- Between unrelated class
  - Create common superclass
  - Move to Component
  - Extract Component (e.g., Strategy)

© imec restricted 2007

Example 4: Guardian Code

- It happens regularly that the body of a method should only be executed when a certain condition is met
  - typically null checks for arguments, etc.
- Schematically the method typically looks like this:

```
MyMethod {
    if (guardian condition) {
        ...
    }
}
```


© imec restricted 2007

Guardian Code Example

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId == R_VIDEO_MENU )
            {
            // Check whether the database has been created or not
            if ( iEngine->GetEngineState() != EPPlaying )
                {
                // The video clip is not being played
                aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
                aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
                }

            // If there is no item in the list box, hide the play, docplay
            // and file info menu items
            if ( !iAppContainer->GetNumOfItemsInListBox() )
                    {
                    aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
                    aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
                    aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
                    }
        }
}
```

Switching it around...

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId != R_VIDEO_MENU ) {return };

      // Check whether the database has been created or not
      if ( iEngine->GetEngineState() != EPPlaying )
              {
              // The video clip is not being played
              aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
              aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
              }

      // If there is no item in the list box, hide the play, docplay
      // and file info menu items
      if ( !iAppContainer->GetNumOfItemsInListBox() )
            {
            aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
            aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
            aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
            }
     }
```

return when condition not met

© imec restricted 2007

Split using the comments

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId != R_VIDEO_MENU ) {return };

    // Check whether the database has been created or not
    if ( iEngine->GetEngineState() != EPPlaying )
            {
            // The video clip is not being played
            aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
            aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
            }

    // If there is no item in the list box, hide the play, docplay
    // and file info menu items
    if ( !iAppContainer->GetNumOfItemsInListBox() )
           {
           aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
           aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
           aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
           }
    }
```

dimButtonsWhenNotPlaying

dimButtonsWhenNoItem

Splitting results

```
void CVideoAppUi::DynInitMenuPaneL(TInt aResourceId,CEikMenuPane* aMenuPane)
{
   if ( aResourceId != R_VIDEO_MENU ) {return };
   dimButtonsWhenNotPlaying(aMenuPane);
   dimButtonsWhenNoItem(aMenuPane);
};

void CVideoAppUi::dimButtonsWhenNotPlaying(CEikMenuPane* aMenuPane)
{
   if ( iEngine->GetEngineState() != EPPlaying ) {
          aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
          aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
   }
}

void CVideoAppUi:: dimButtonsWhenNoItem(CEikMenuPane* aMenuPane)
{
  if ( !iAppContainer->GetNumOfItemsInListBox() ) {
          aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
          aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
          aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
  }
}
```


© imec restricted 2007

More things that could be done

```
void CVideoAppUi::DynInitMenuPaneL(TInt aResourceId,CEikMenuPane* aMenuPane)
{
   if ( aResourceId != R_VIDEO_MENU ) {return };
   dimButtonsWhenNotPlaying(aMenuPane);
   dimButtonsWhenNoItem(aMenuPane);
};

void CVideoAppUi::dimButtonsWhenNotPlaying(CEikMenuPane* aMenuPane)
{
   if ( iEngine->GetEngineState() != EPPlaying ) {
          aMenuPane->DimItem( EVideoCmdAppStop);
          aMenuPane->DimItem( EVideoCmdAppPause);
   }
}

void CVideoAppUi:: dimButtonsWhenNoItem(CEikMenuPane* aMenuPane)
{
  if ( !iAppContainer->GetNumOfItemsInListBox() ) {
          aMenuPane->DimItem( EVideoCmdAppDocFileInfo);
          aMenuPane->DimItem( EVideoCmdAppDocPlay);
          aMenuPane->DimItem( EVideoCmdAppPlay);
  }
}
```


© imec restricted 2007

And More...

```
void CVideoAppUi::DynInitMenuPaneL(CEikMenuPane* aMenuPane)
{
   dimButtonsWhenNotPlaying(aMenuPane);
   dimButtonsWhenNoItem(aMenuPane);
};

void CVideoAppUi::dimButtonsWhenNotPlaying(CEikMenuPane* aMenuPane)
{
   if ( iEngine->notPlaying() ) {
           aMenuPane->DimItem( EVideoCmdAppStop);
           aMenuPane->DimItem( EVideoCmdAppPause);
   }
}

void CVideoAppUi:: dimButtonsWhenNoItem(CEikMenuPane* aMenuPane)
{
  if ( iAppContainer->isListBoxEmpty() ) { return; };

  aMenuPane->DimItem( EVideoCmdAppDocFileInfo);
  aMenuPane->DimItem( EVideoCmdAppDocPlay);
  aMenuPane->DimItem( EVideoCmdAppPlay);
}
```

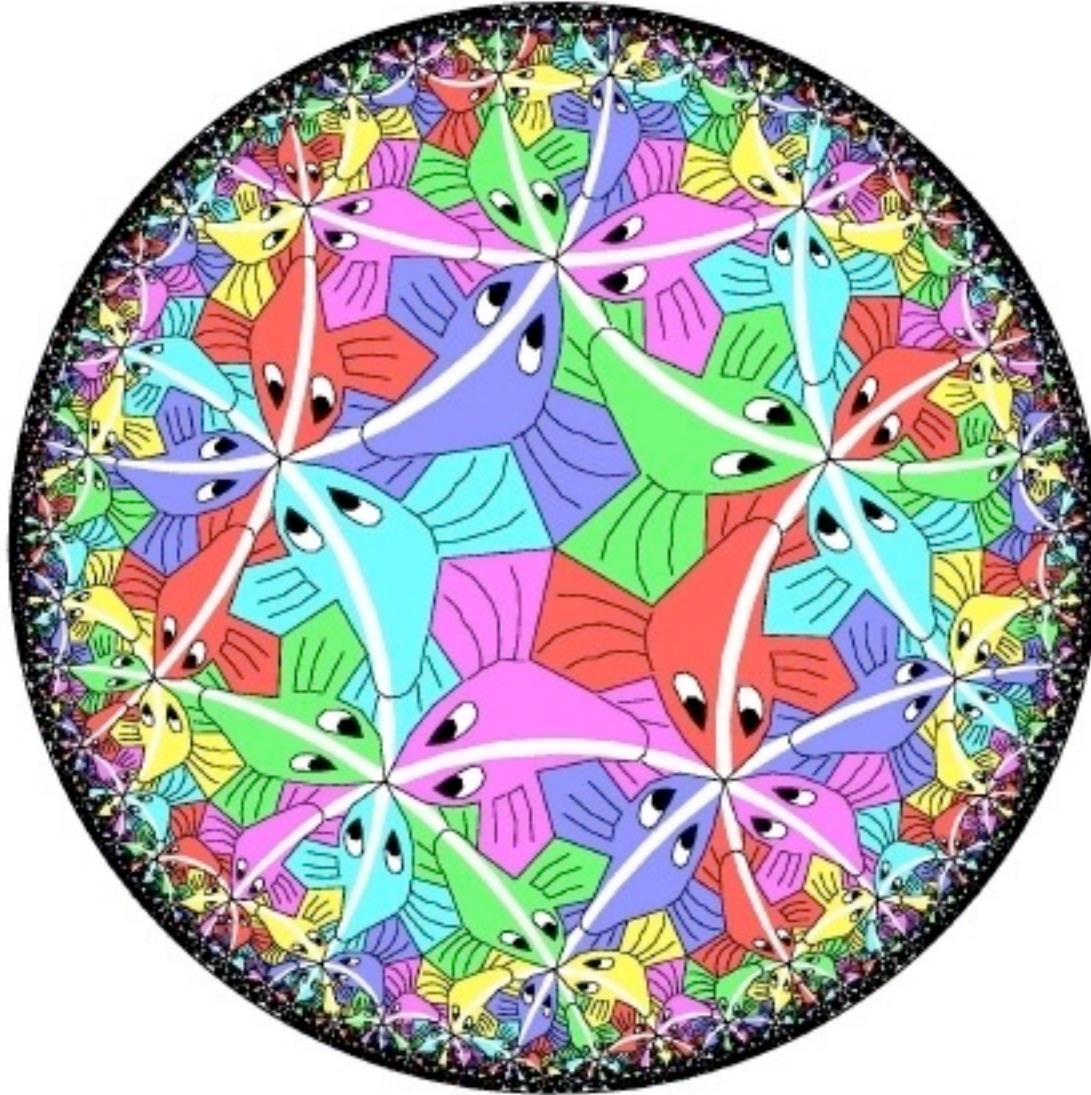

© imec restricted 2007

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

© imec restricted 2007

Patterns



© imec restricted 2007

Bit of history...

- Christoffer Alexander
  - "The Timeless Way of Building", Christoffer Alexander, Oxford University Press, 1979, ISBN 0195024028
  - Structure of the book is magnificent
    - 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"

© imec restricted 2007

Illustrating Recurring Patterns...

© imec restricted 2007

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

© imec restricted 2007

Responsibility Driven Design

- Metaphor – can compare to people
  - Objects have responsibilities
  - Objects collaborate
  - Similar to how we conceive of people
- 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

© imec restricted 2007

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

Wuyts Roel
© imec restricted 2007

Responsibilities and Methods

- Responsibilities are assigned to classes during object design, and are reflected in methods
  - We may declare the following:
    - "a Sale is responsible for creating SalesLineItems" (doing)
    - "a Sale is responsible for knowing its total" (knowing)
- Responsibilities related to "knowing"
  - often inferable from the Domain Model (because of the attributes and associations it illustrates)
  - ! **low representational gap** ! – **LRG**

Wuyts Roel
© imec restricted 2007

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

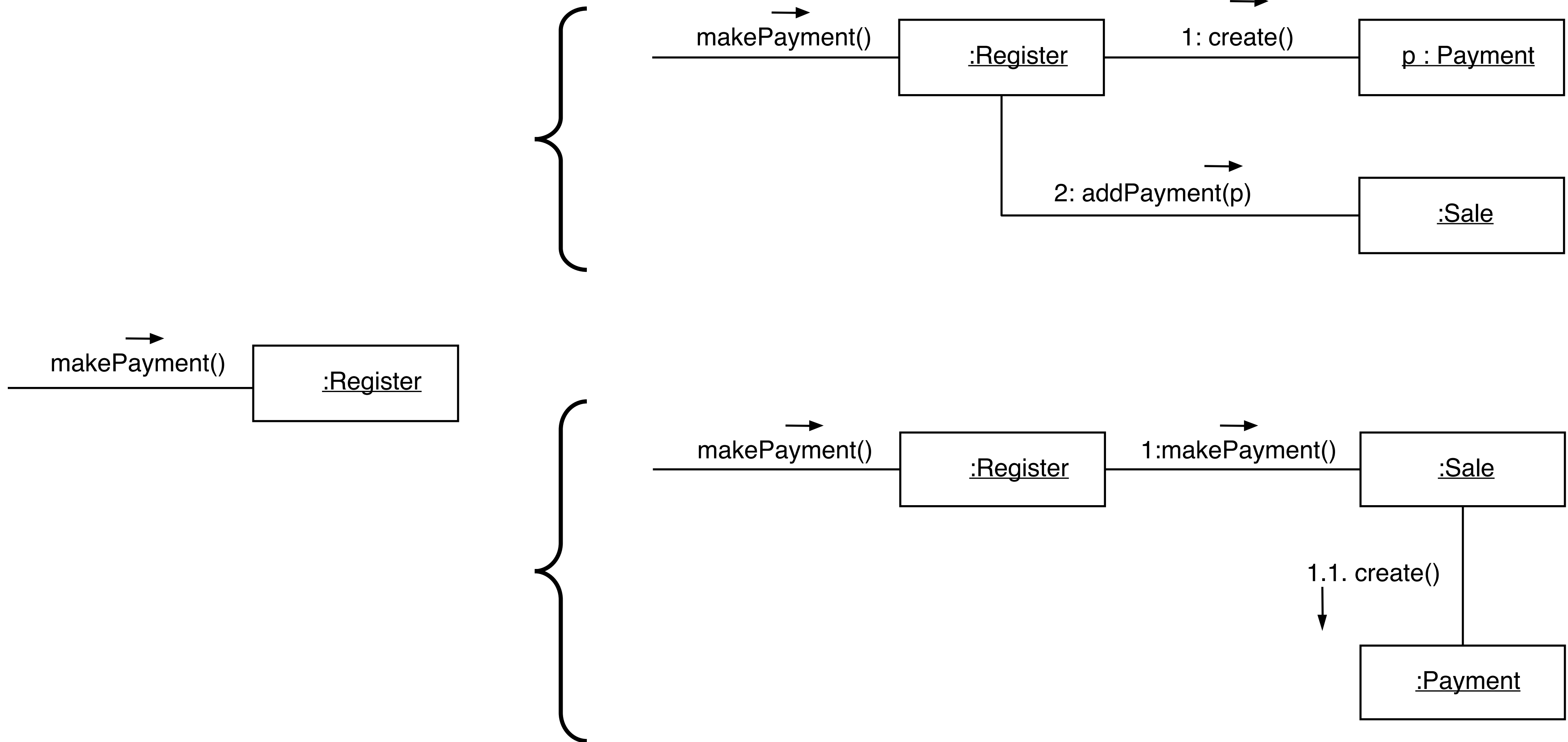
Wuyts Roel
© imec restricted 2007

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.

© imec restricted 2007

Low Coupling Patroon



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

© imec restricted 2007

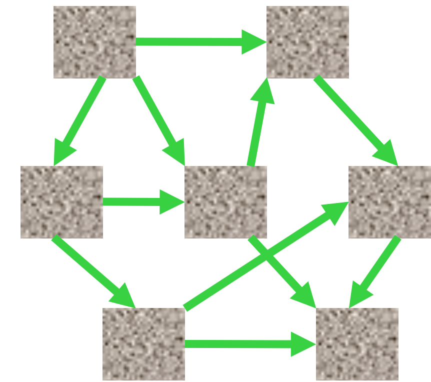
Low Coupling Patroon

- Coupling is a measure that shows how much a class is dependent on other classes
- X depends on 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
  - (X does not compile without Y)
- “evaluative” pattern:
  - use it to evaluate alternatives
  - try to reduce coupling

Wuyts Roel
© imec restricted 2007

Low Coupling Patroon

- Coupling is a measure that shows how much a class is dependent on other classes
- X depends on 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
  - (X does not compile without Y)

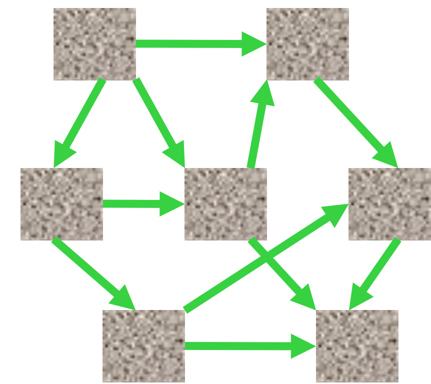


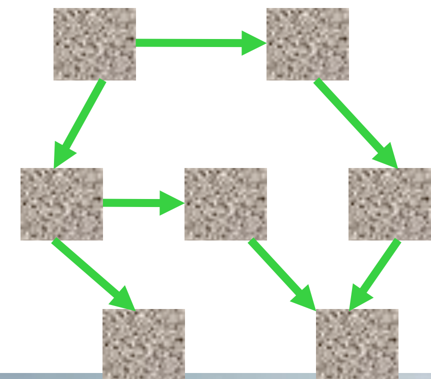
- “evaluative” pattern:
  - use it to evaluate alternatives
  - try to reduce coupling

© imec restricted 2007

Low Coupling Patroon

- Coupling is a measure that shows how much a class is dependent on other classes
- X depends on 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
  - (X does not compile without Y)
- "evaluative" pattern:
  - use it to evaluate alternatives
  - try to reduce coupling





© imec restricted 2007

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
- Inheritance is strong coupling !!

© imec restricted 2007

Low Coupling Patroon: 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

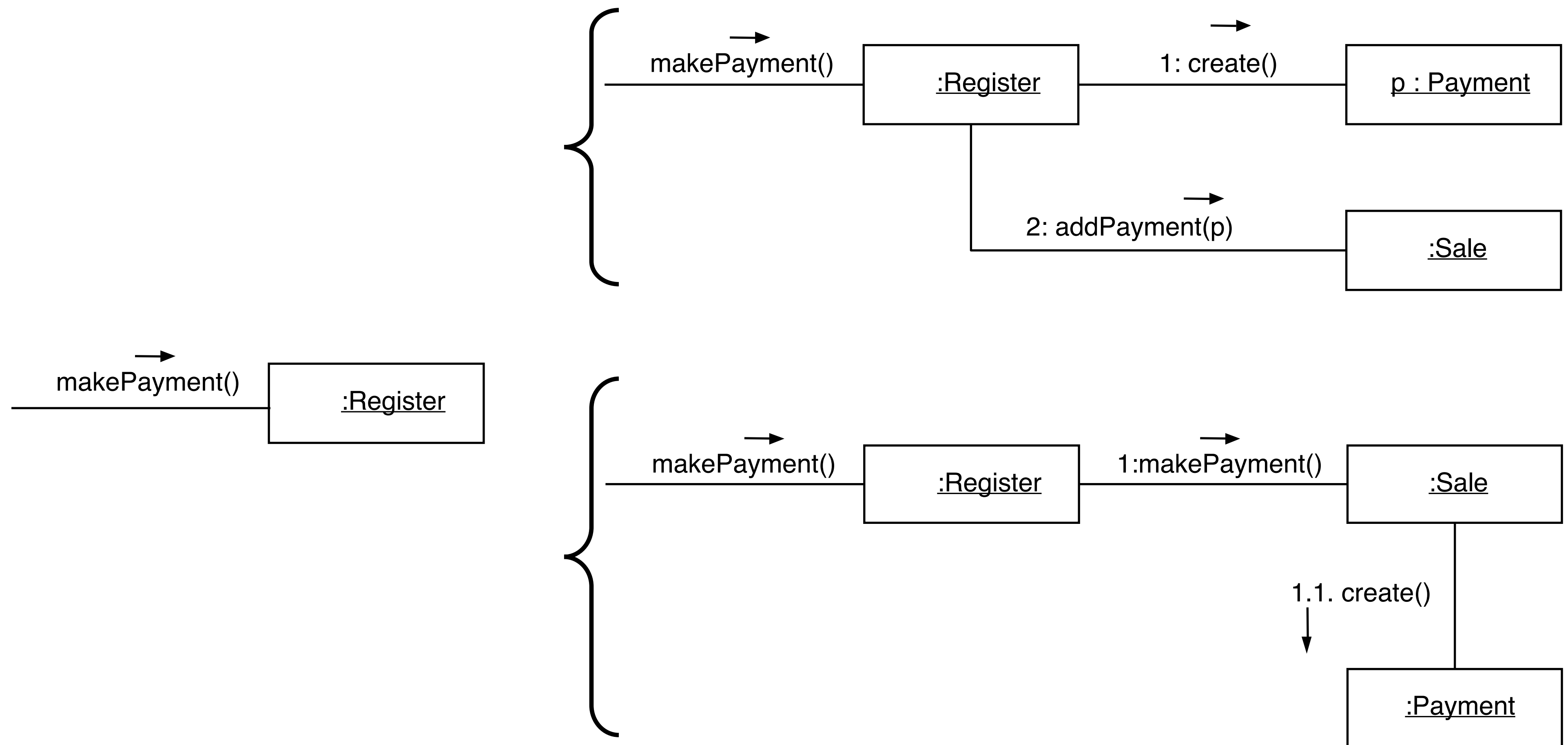
© imec restricted 2007

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.

© imec restricted 2007

High Cohesion Patroon



- 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 Patroon

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

Wuyts Roel
© imec restricted 2007

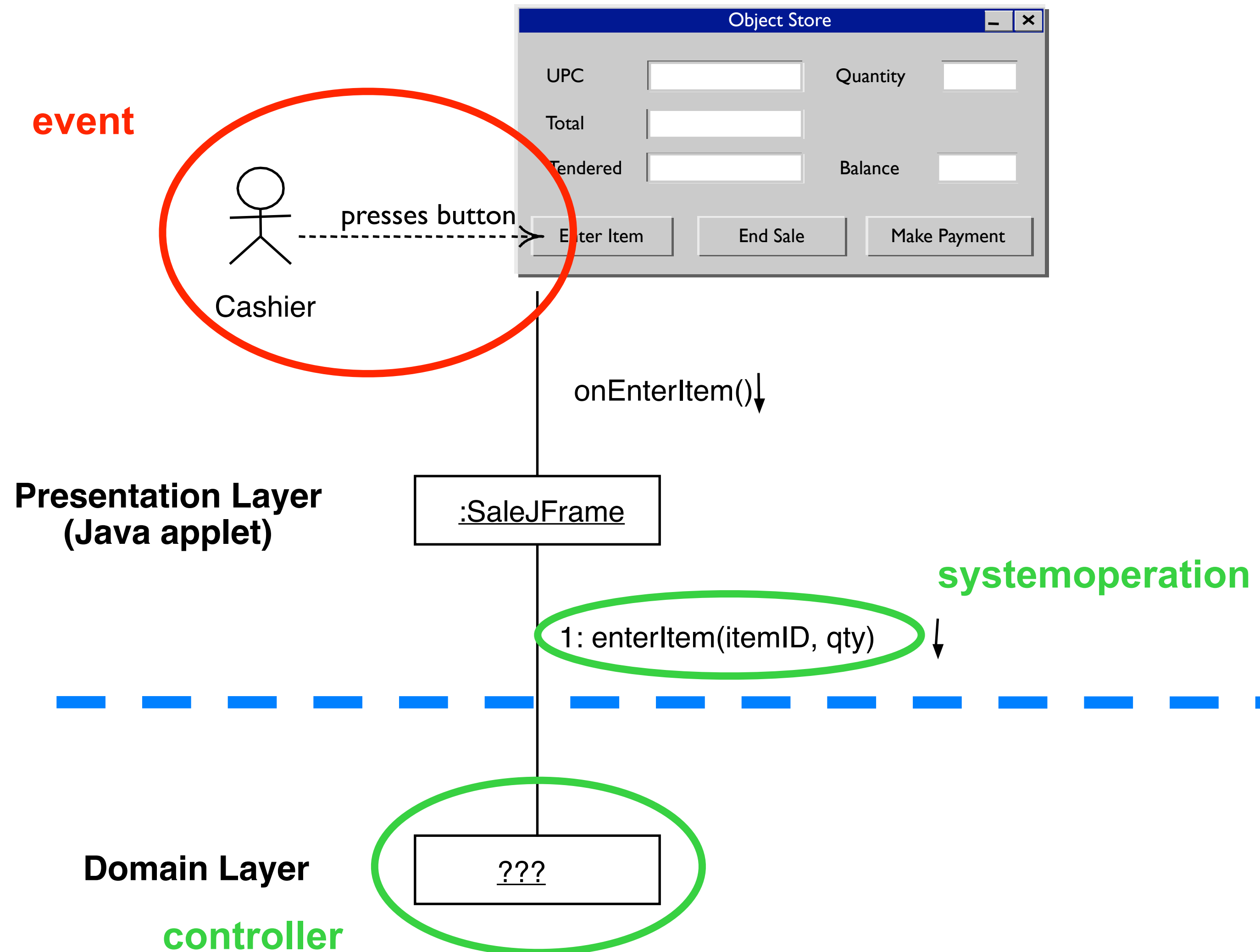
High Cohesion Pattern: remarks

- 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
  - *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.
    - klasse RDBInterface: partially responsible for interacting with relational databases

© imec restricted 2007

1. Controller Pattern

- Who is responsible for handling *Systemoperations* ?



© imec restricted 2007

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:

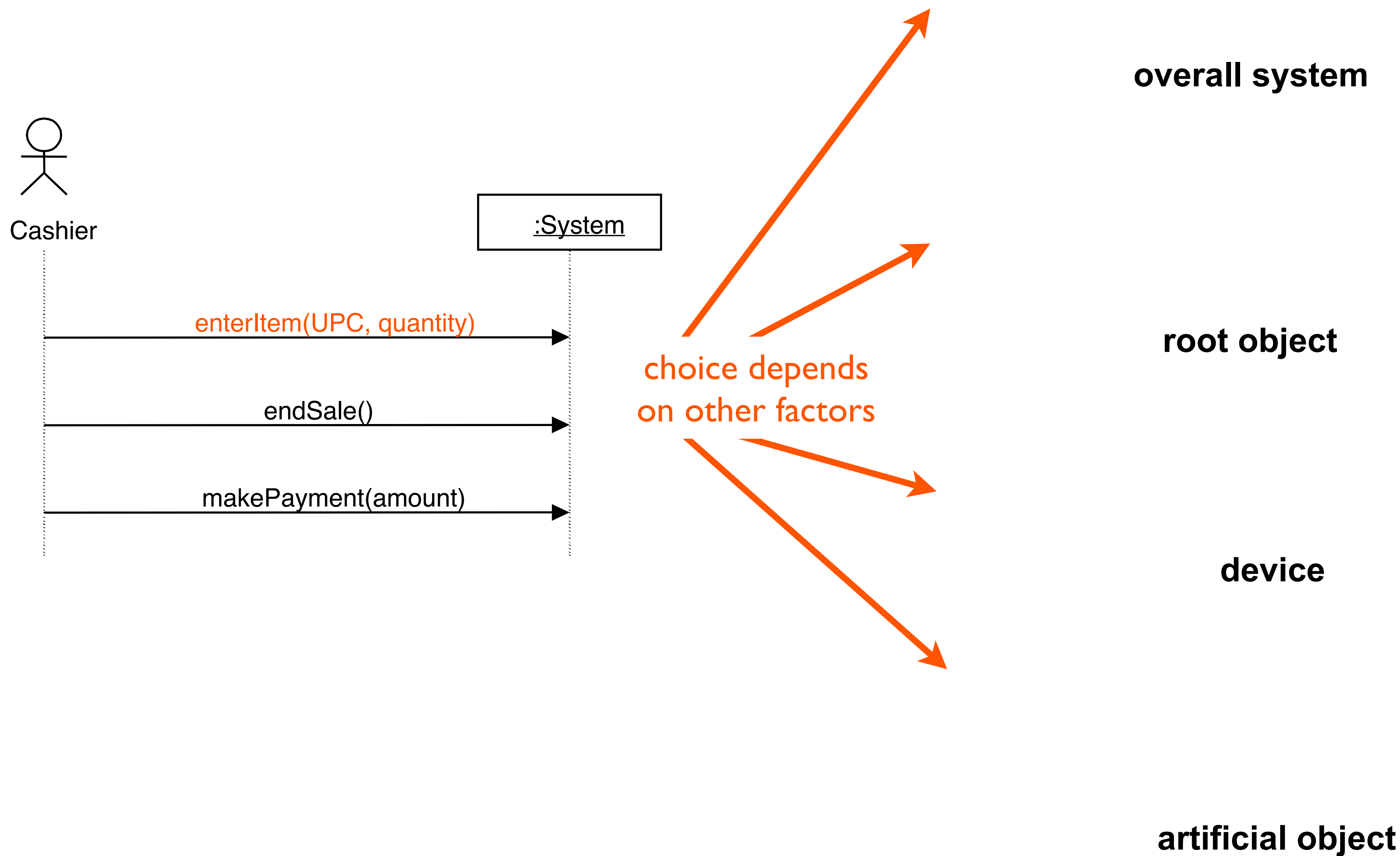
- C is a *facade controller*: it represents the overall system, a root object, the device that runs the software, or a major subsystem.
- C is a *use case or session controller*: it represents an artificial objects (see *Pure Fabrication* pattern) that handles all events from a use case or session

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)

© imec restricted 2007

Who controls System events?



overall system

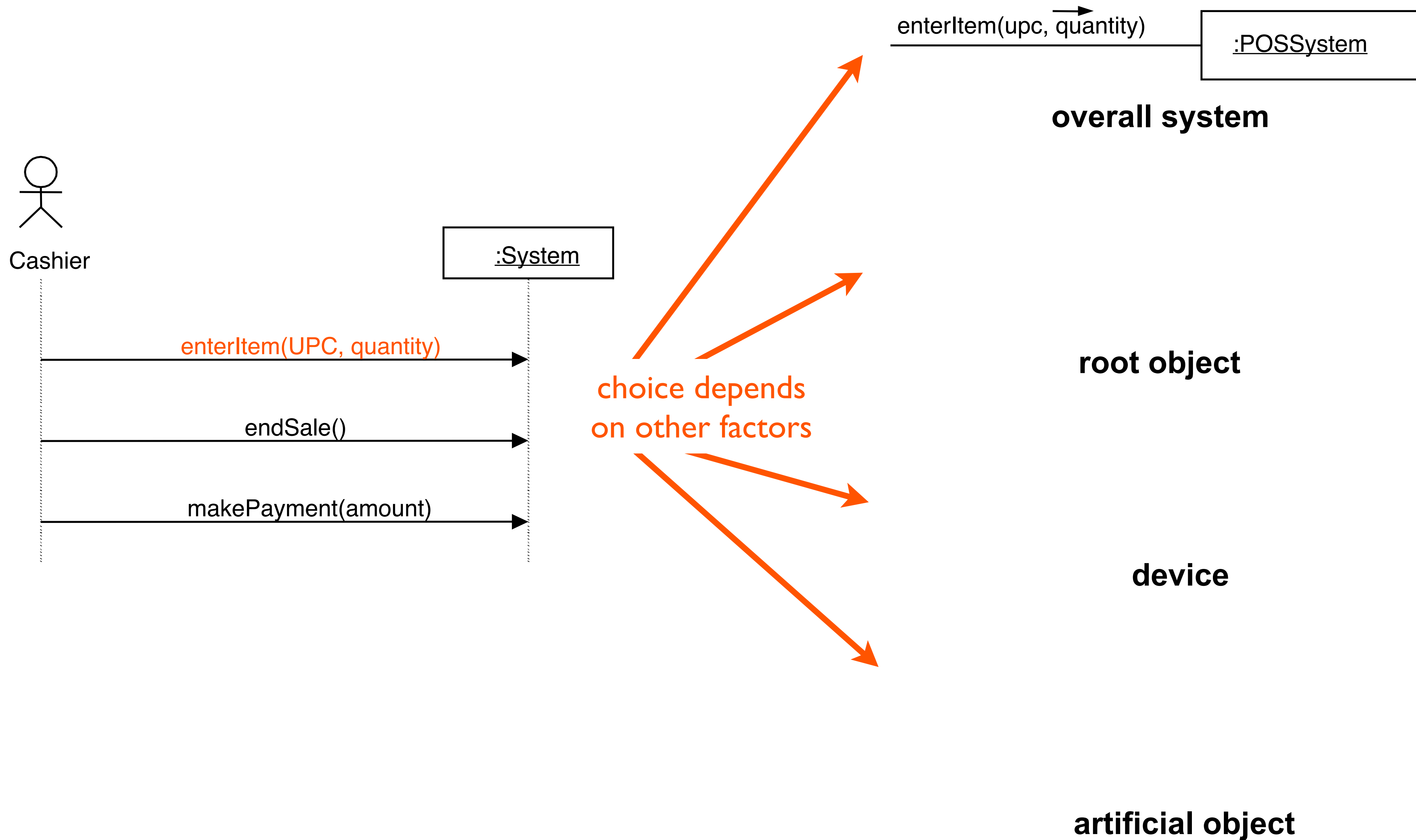
root object

device

artificial object

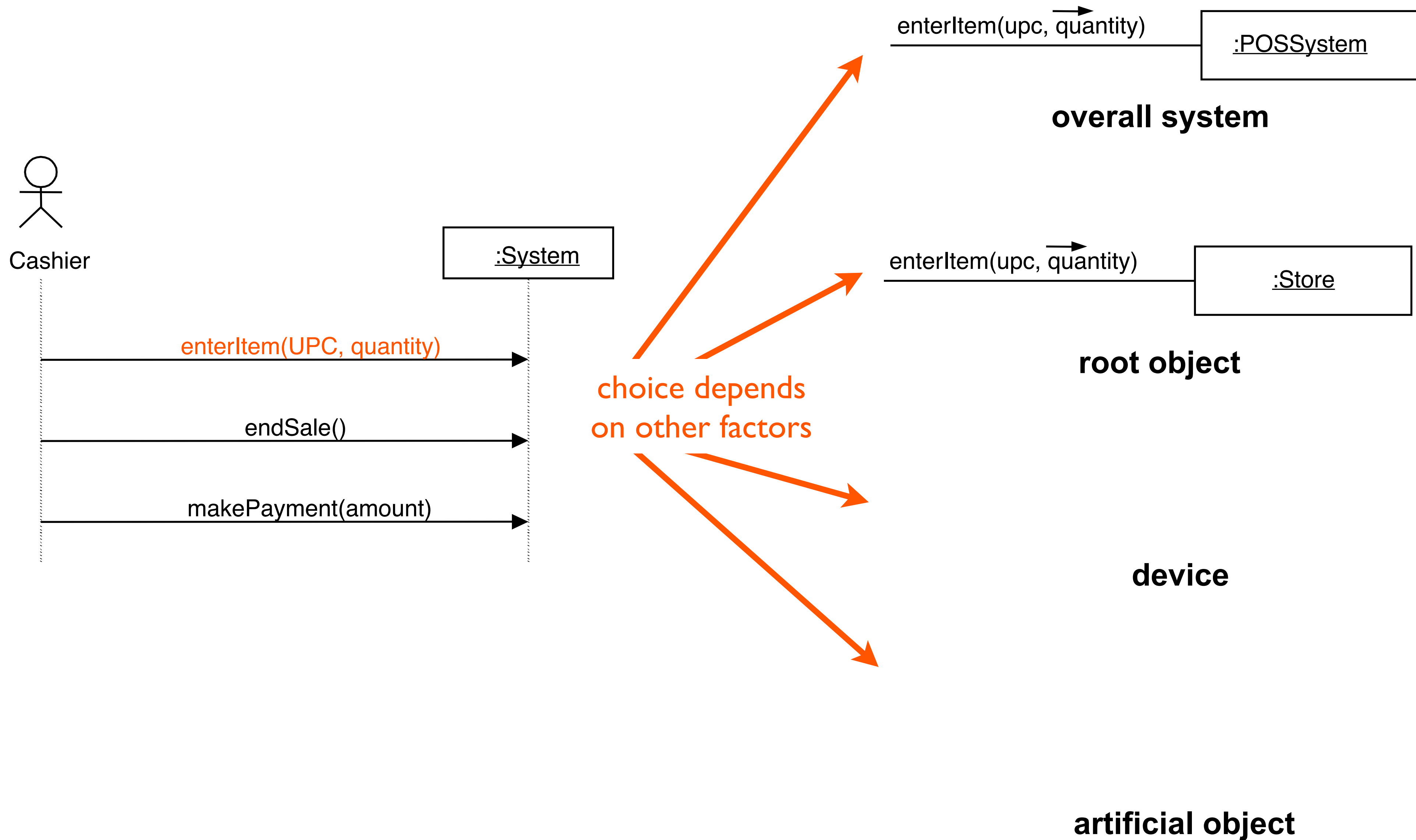
© imec restricted 2007

Who controls System events?



© imec restricted 2007

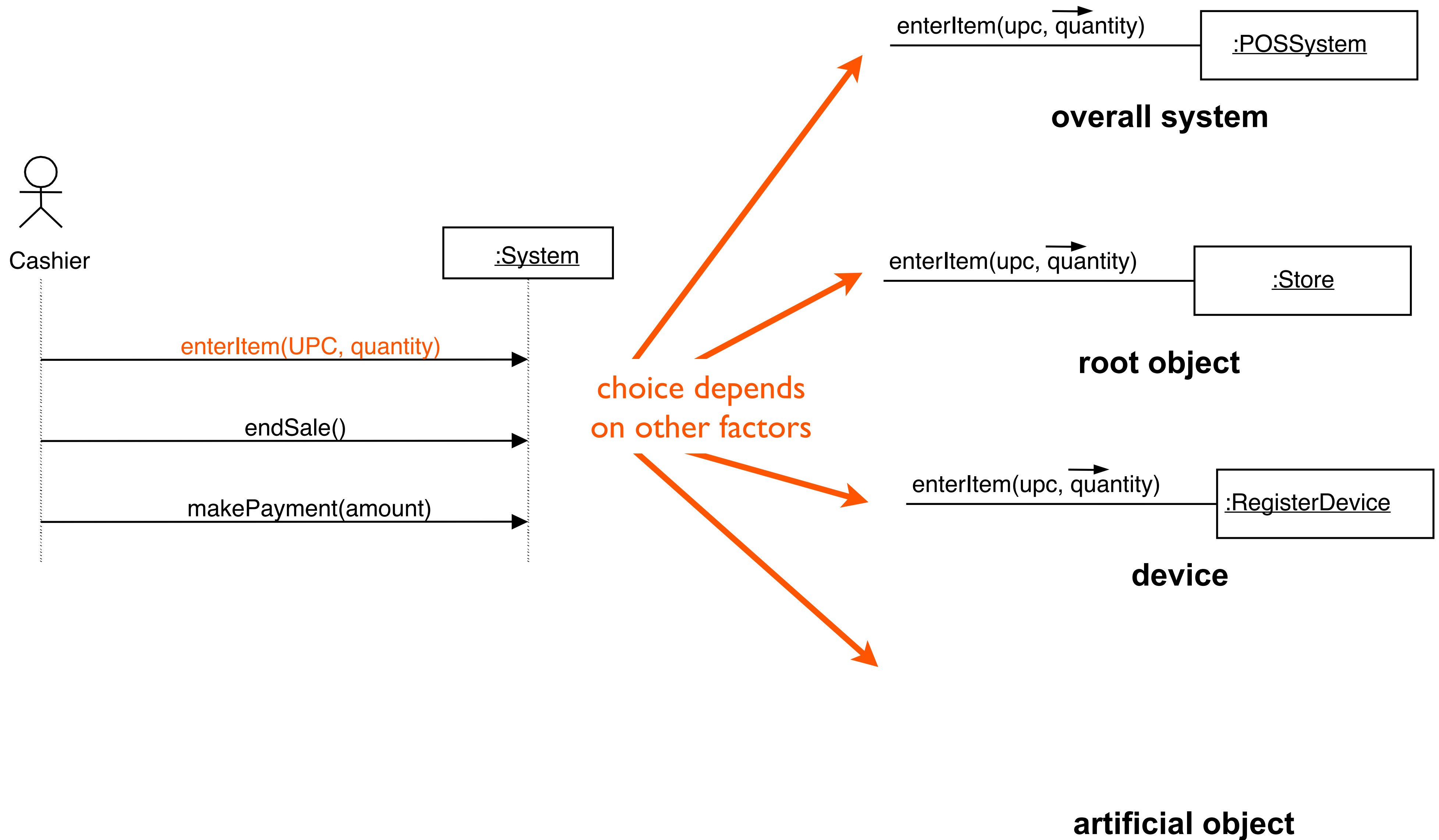
Who controls System events?



© imec restricted 2007

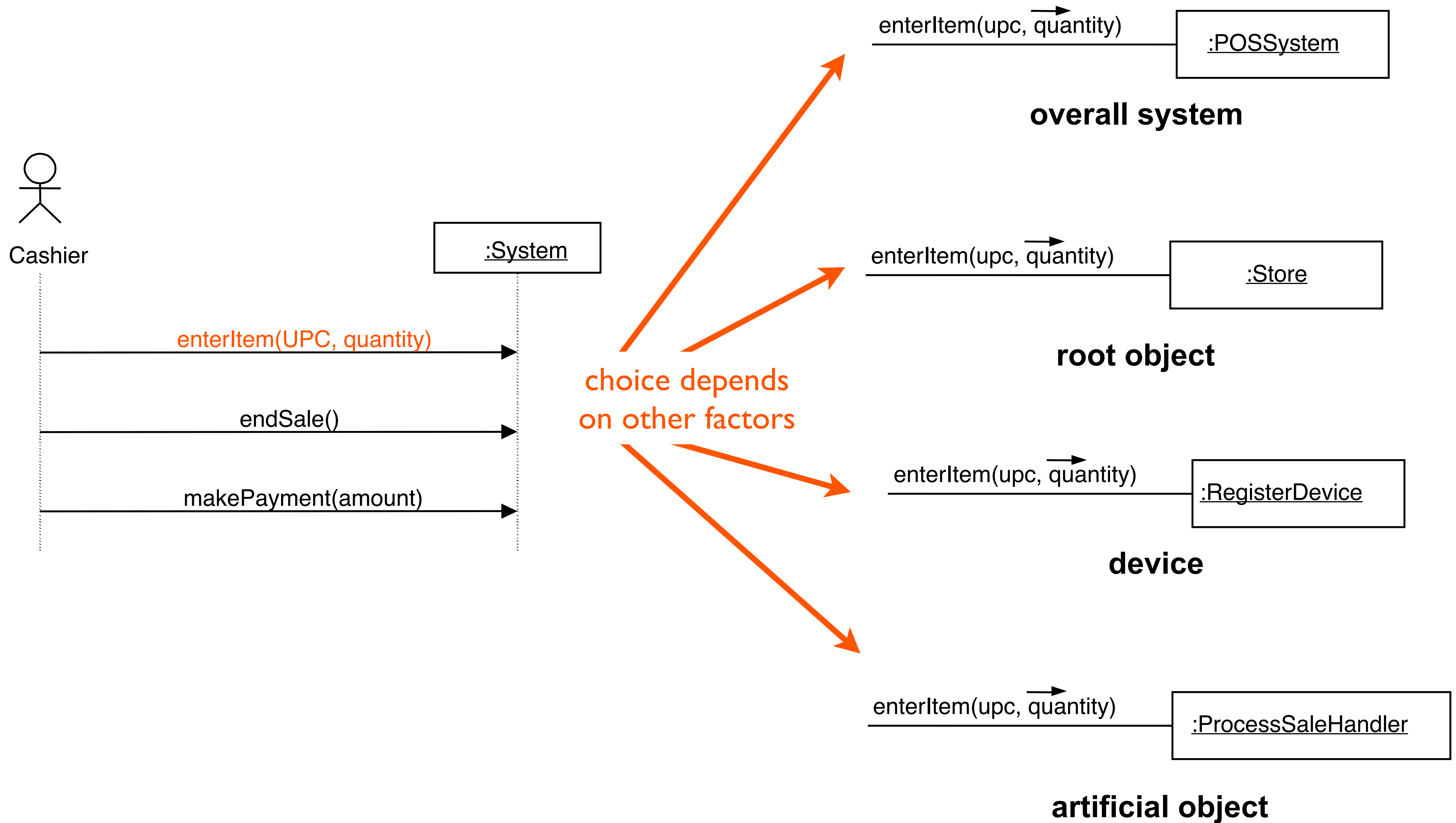
Thursday 26 September 13

Who controls System events?



Wuyts Roel
© imec restricted 2007

Who controls System events?



Wuyts Roel
© imec restricted 2007

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

© imec restricted 2007

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)

© imec restricted 2007

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

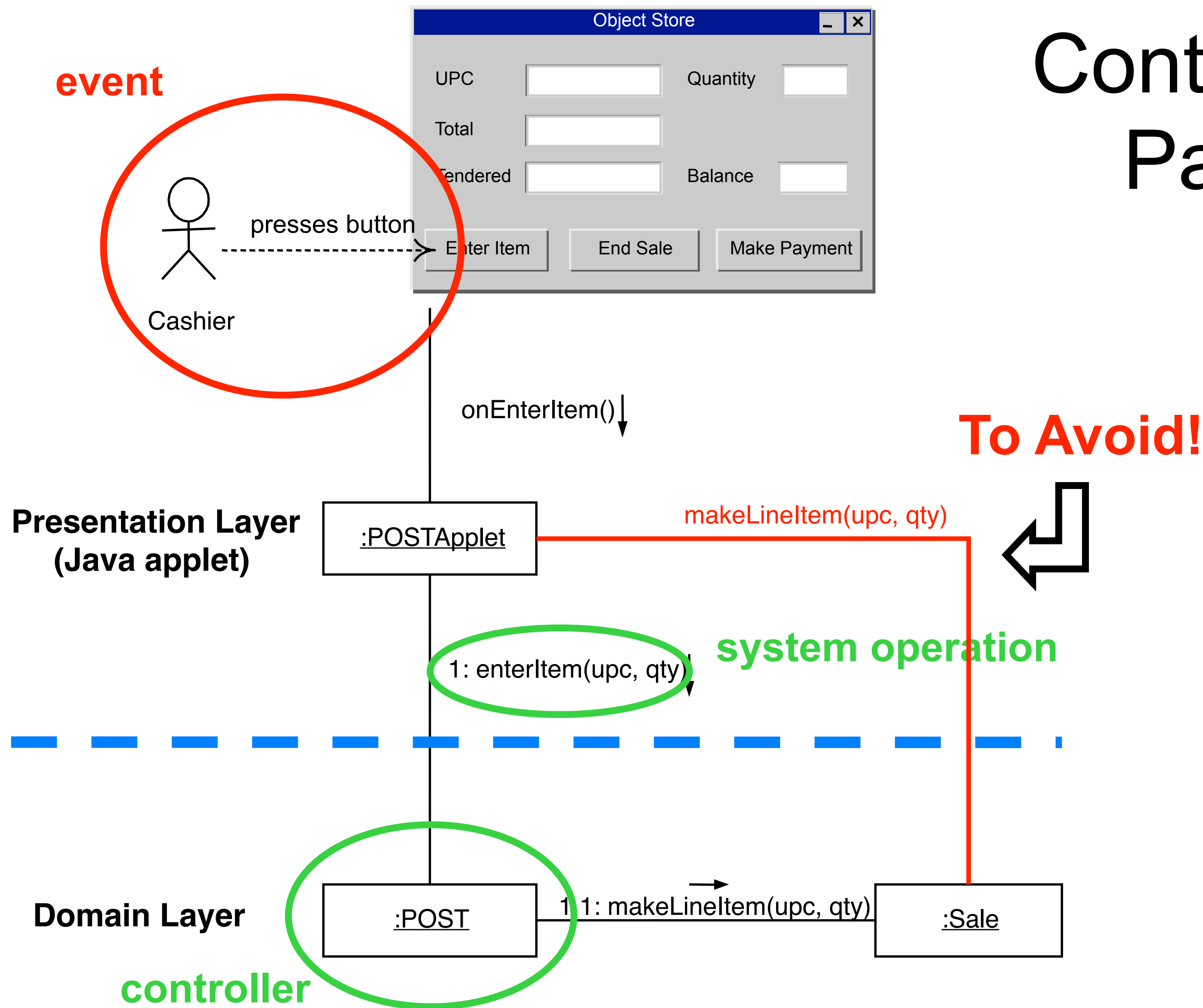
© imec restricted 2007

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

© imec restricted 2007

Example



Controller Pattern

© imec restricted 2007

2. Creator Pattern

Pattern **Creator**

Problem Who is responsible for creating instances of classes ?

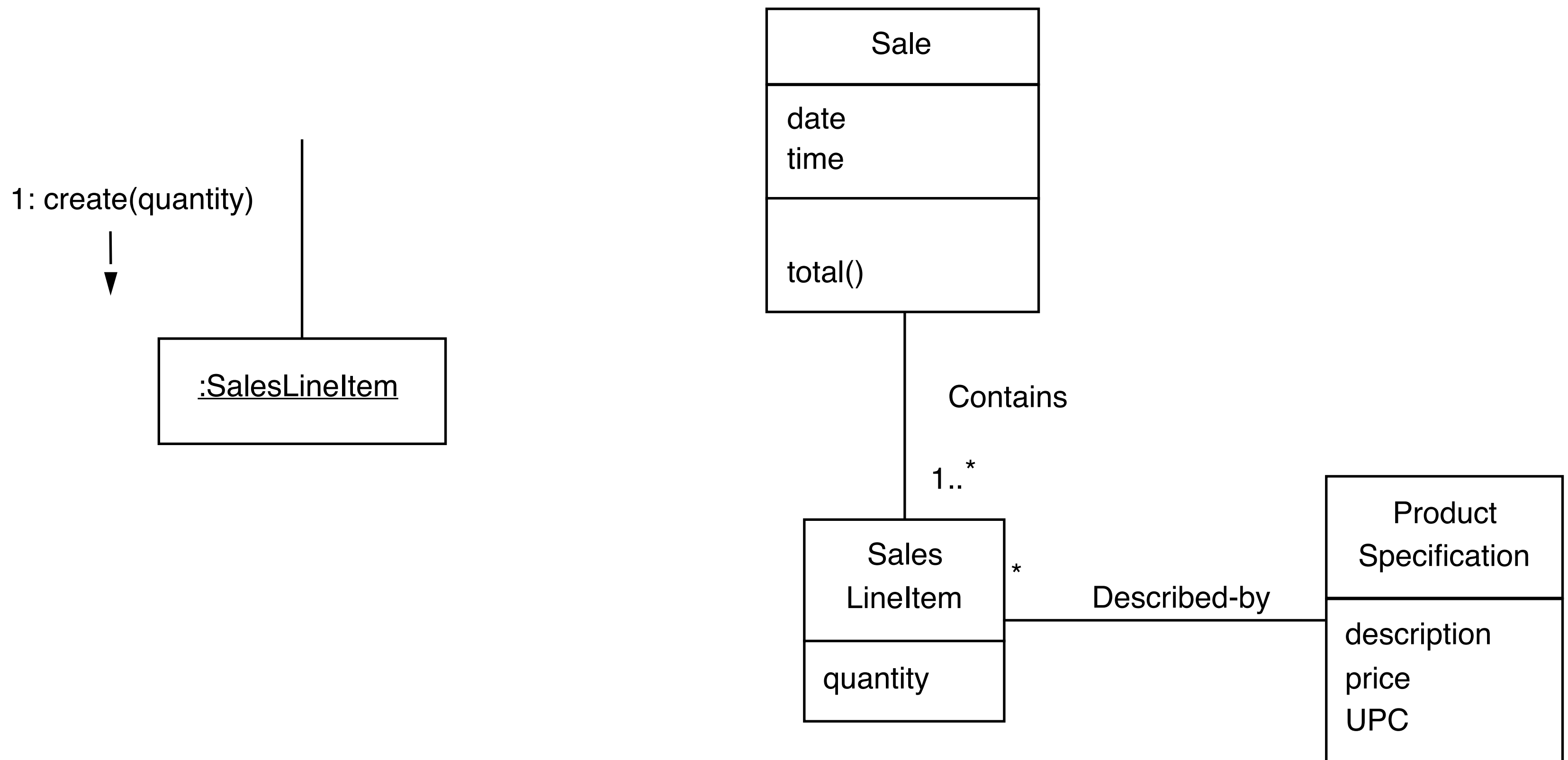
Solution Assign a class B to create instances of a class A if:

- 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

© imec restricted 2007

Creator Pattern: example

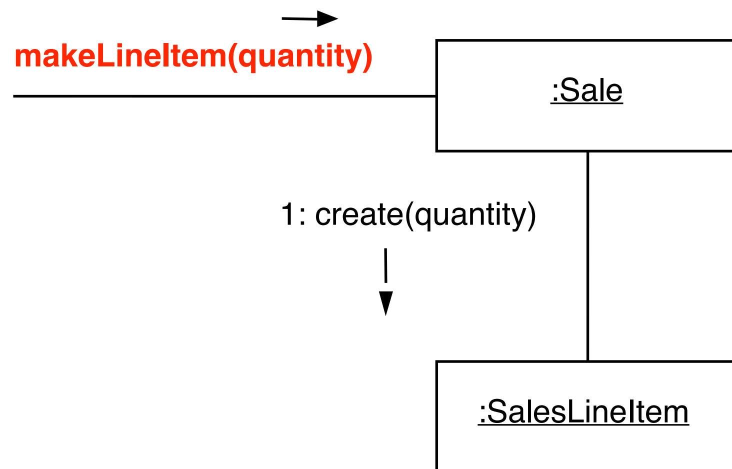
Creation of “SalesLineItem” instances

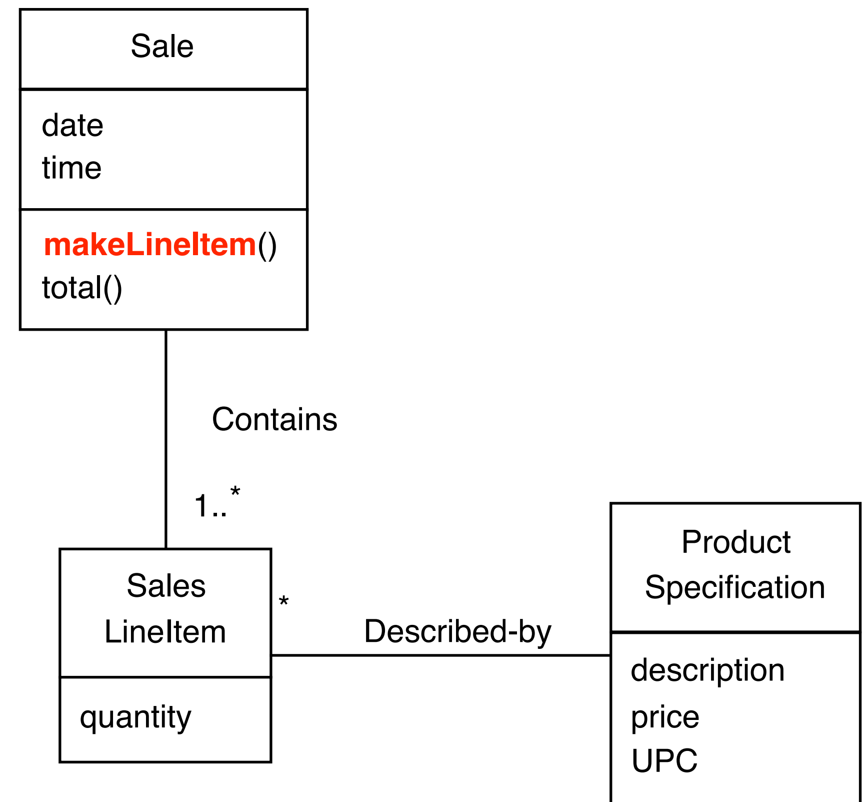


Wuyts Roel
© imec restricted 2007

Creator Pattern: example

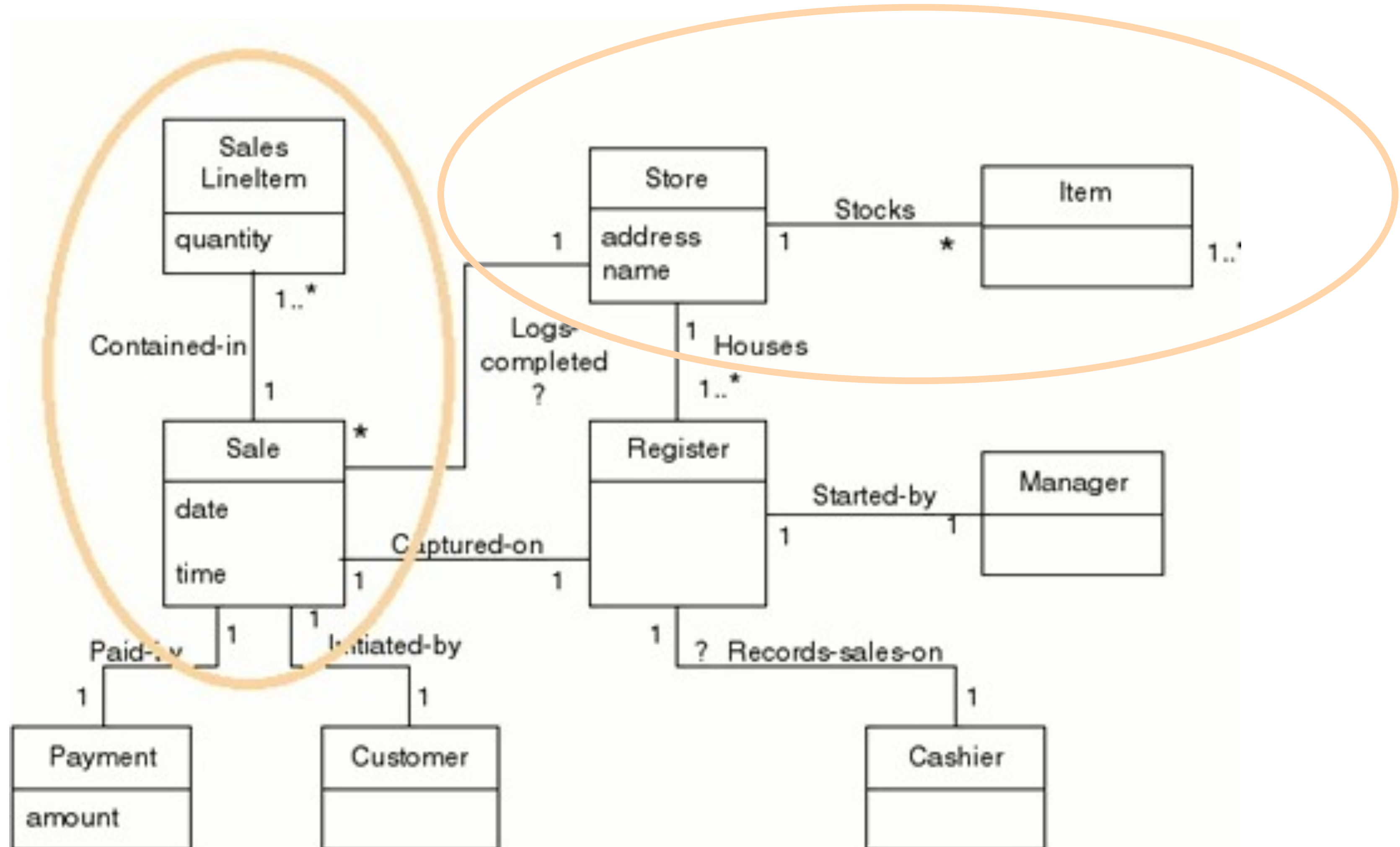
Creation of "SalesLineItem" instances





© imec restricted 2007

Creator Pattern: Inspiration from the Domain Model



© imec restricted 2007

Thursday 26 September 13

3. Information Expert Pattern

- A very basic principle of responsibility assignment
- 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

© imec restricted 2007

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)

© imec restricted 2007

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

© imec restricted 2007

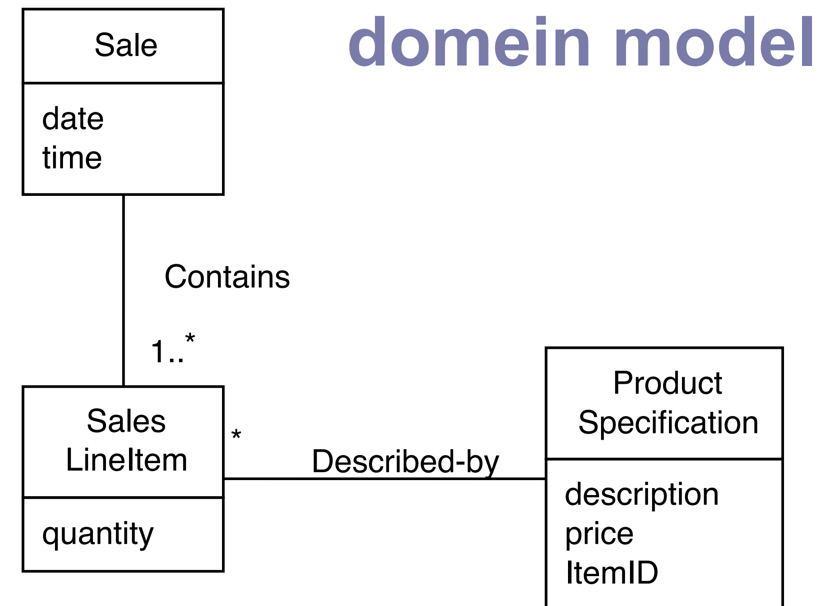
Expert Patroon: example 1

- Example: **Who is responsible for knowing the total of a "Sale"?**
- Who possesses the information?

Wuyts Roel
© imec restricted 2007

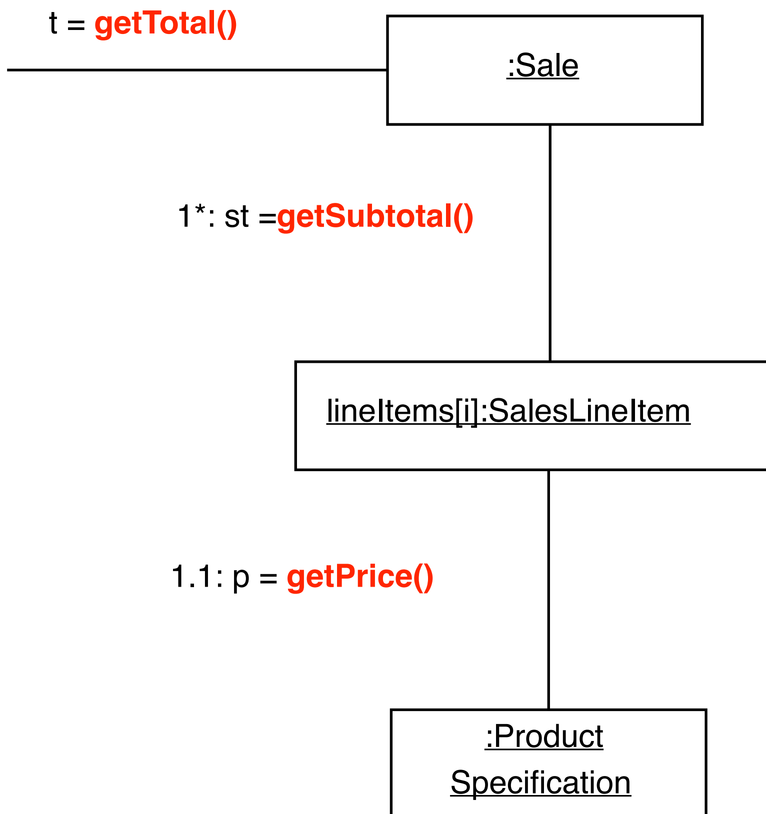
Expert Patroon: example 1

- Example: **Who is responsible for knowing the total of a “Sale”?**
- Who possesses the information?



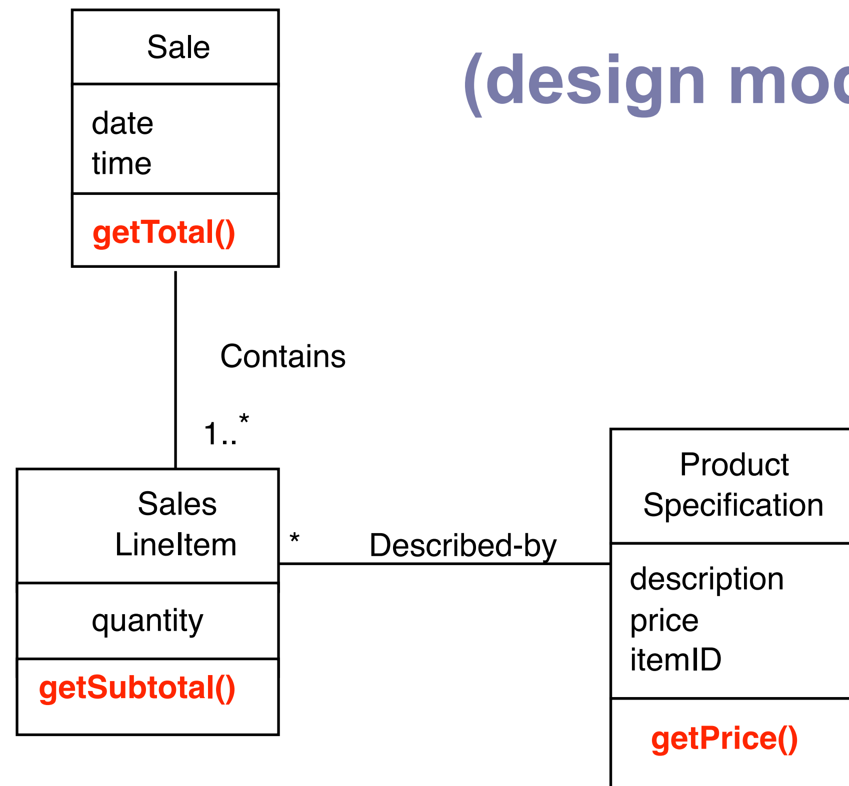
© imec restricted 2007

Expert Pattern



class diagram

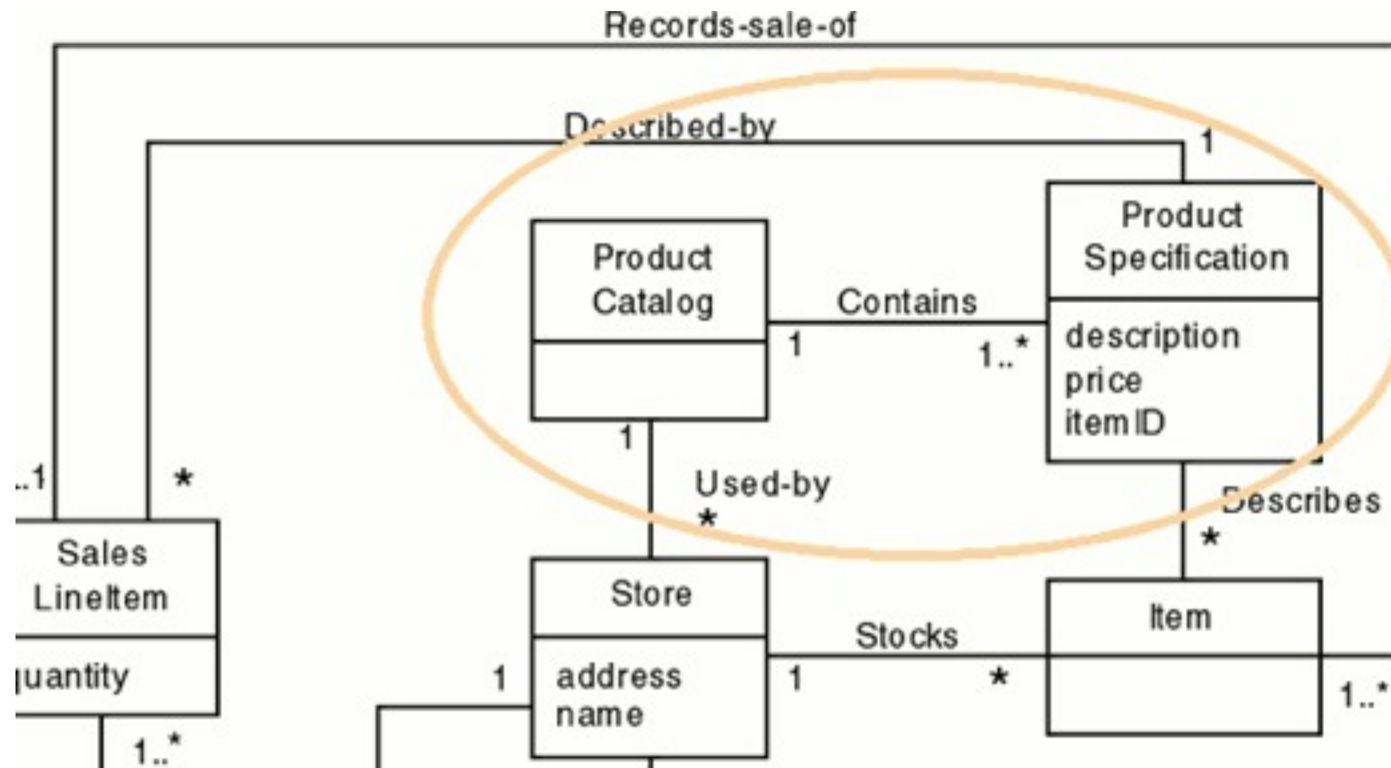
(design model)



© imec restricted 2007

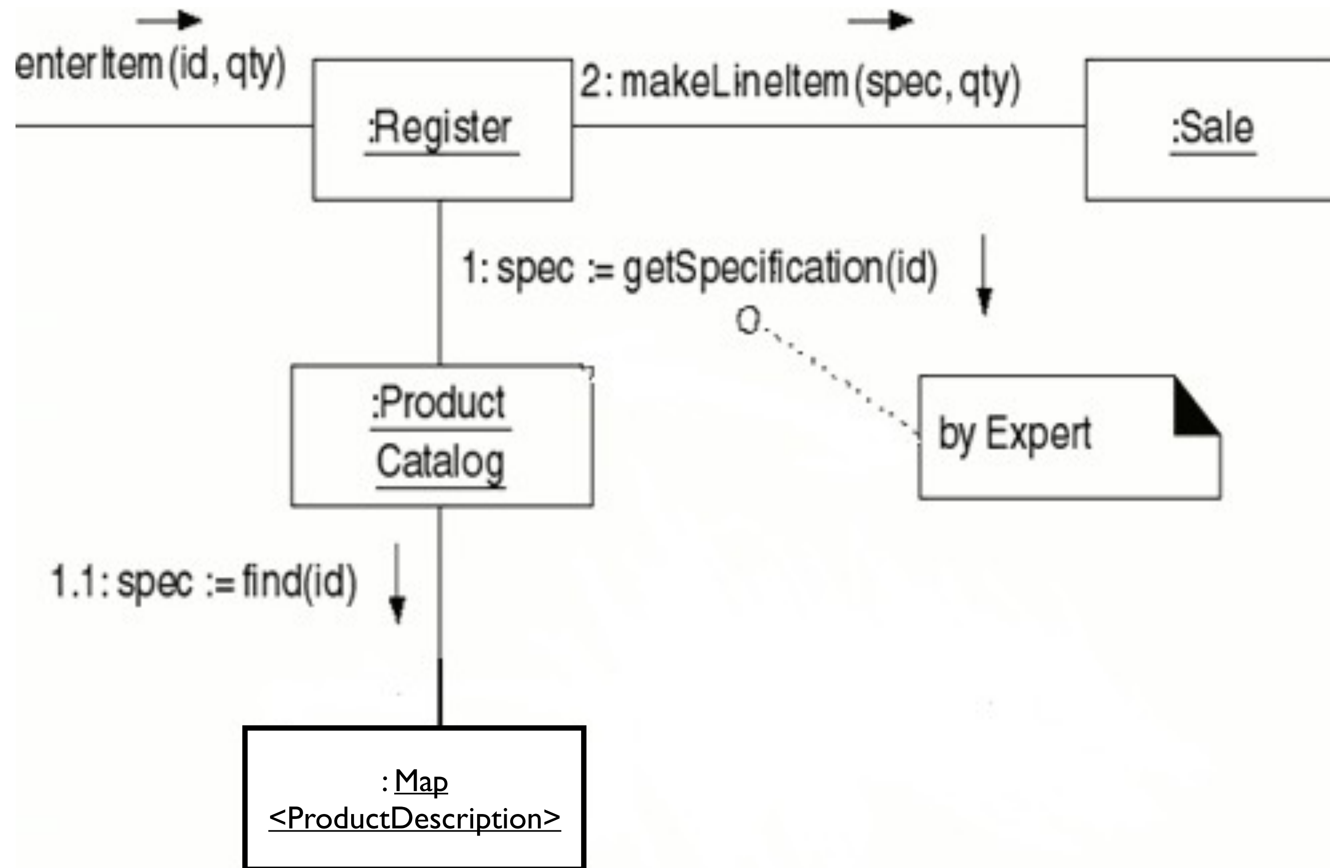
Expert Pattern: Example 2

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



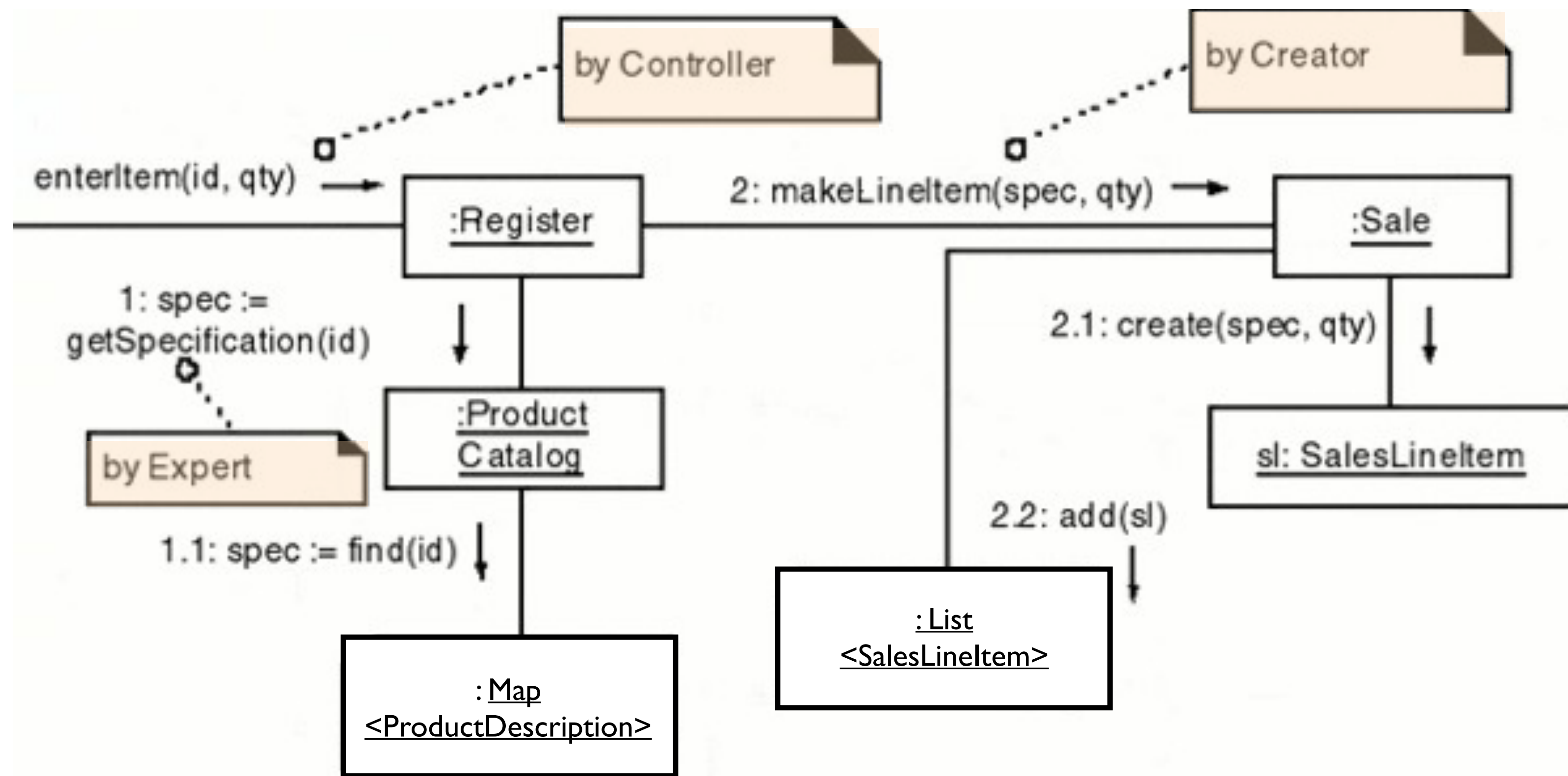
© imec restricted 2007

Applying Information Expert



© imec restricted 2007

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 (Controller – High Cohesion)

Hs 25 (Polymorphism – Protected Variations)

© imec restricted 2007

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.

Example

```
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; }

        // File info command is selected
         case EVideoCmdAppDocFileInfo:
                { DoGetFileInfoL(); break; }
......
```

Replace case by Polymorphism

```
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 ...};
```


© imec restricted 2007

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.

© imec restricted 2007

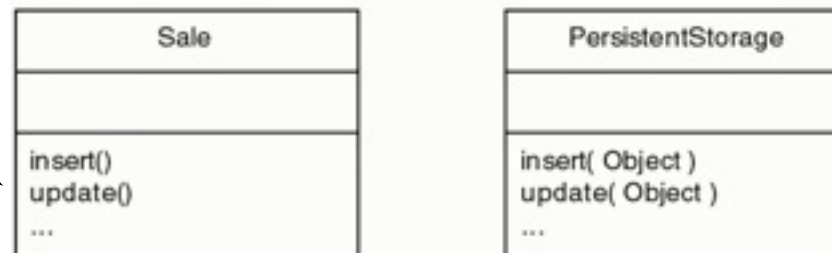
Pure Fabrication Pattern

- 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

Expert
=>High Coupling
Low Cohesion



Pure Fabrication
=> Low Coupling
High Cohesion

© imec restricted 2007

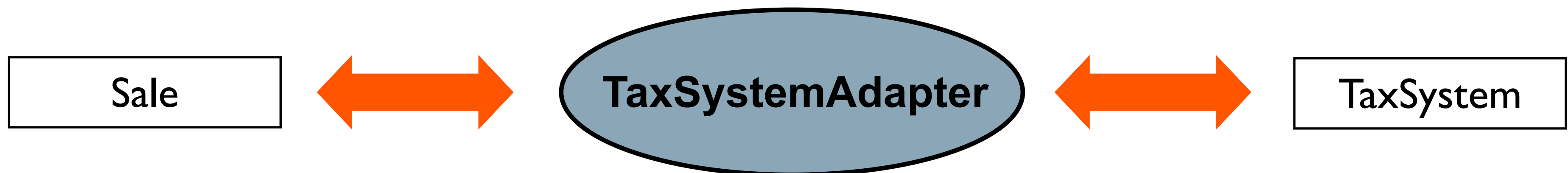
8. Indirection Pattern

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.

© imec restricted 2007

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)



© imec restricted 2007

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.

© imec restricted 2007

Protected Variations – voorbeeld

- 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

© imec restricted 2007

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