Aspects And Use Cases

Preserving Use Cases in Code and OOram

Use Cases Preserved in Code

Thanks to Aspects

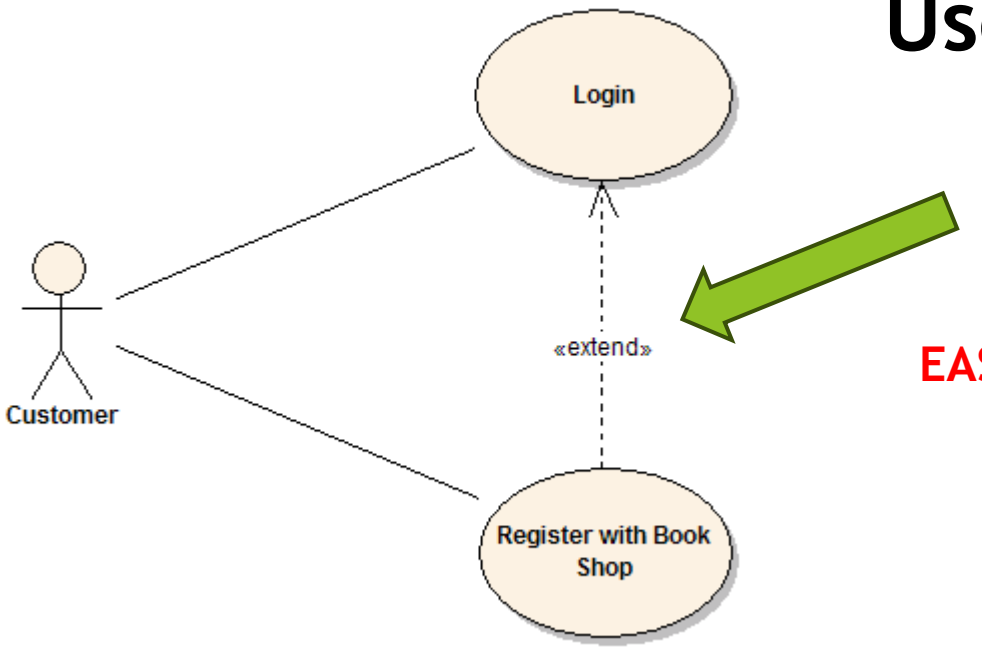
Agenda

-**modularizing use cases:** Use-Case modularity problem -
-**similarly tackling with abstractions:**
demonstration on aspect refactoring of object-oriented patterns
-**peer use cases:** symmetric aspect-oriented programming
-**use cases behind themes**
-**handling include relation**
-**extend relation:** asymmetric aspect-oriented programming
-**aspects as roles collaborations:** presenting on OOram

AOSD via Use Cases

- **Ivar Jacobson** - the one who first came with this idea in 2003

Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison Wesley Professional (2004), ISBN 0-321-26888-1.



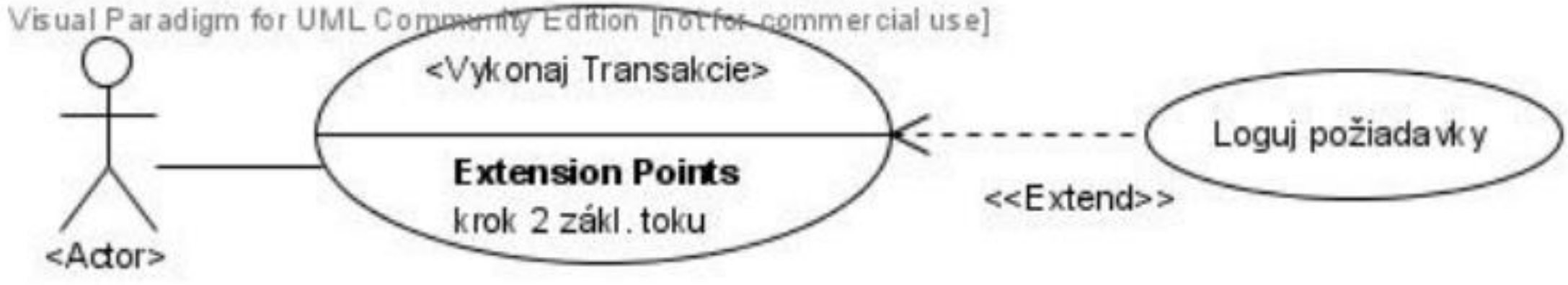
Use-case modularity problem

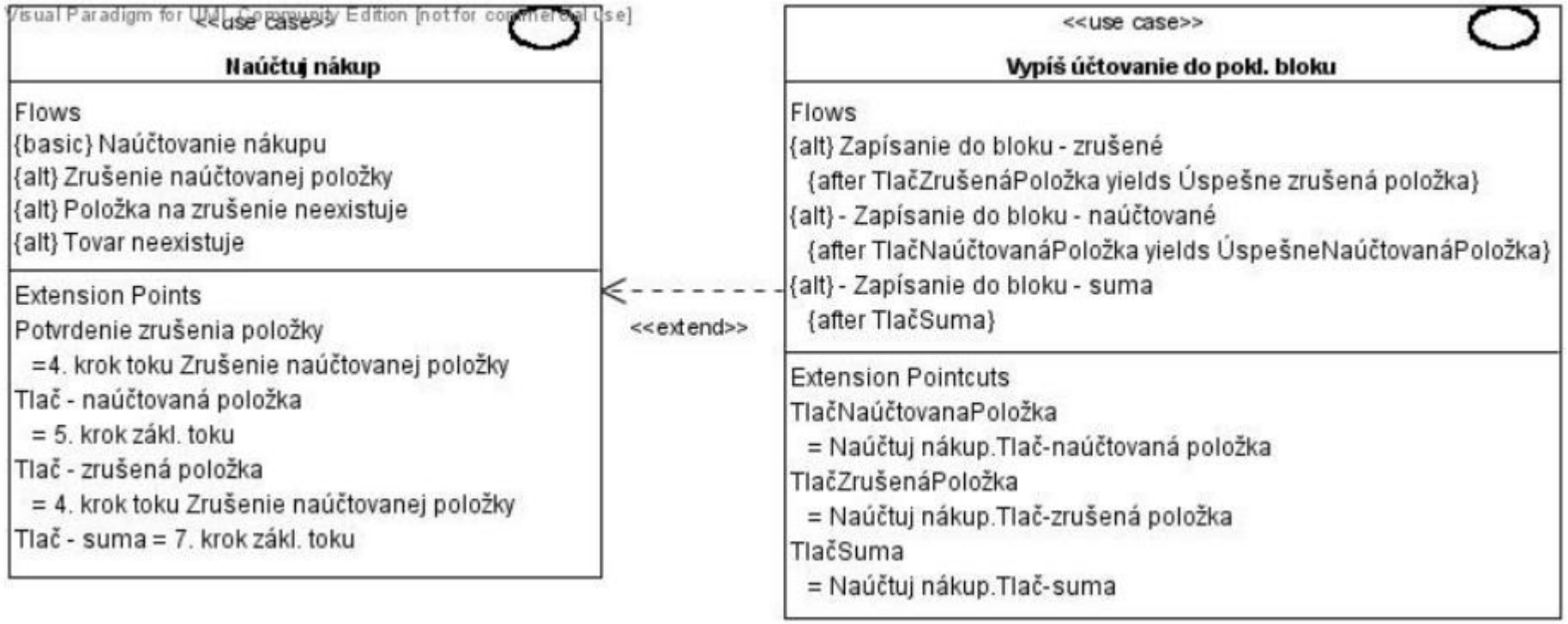
Previously unsupported in analytical models and in implementational environments

EASILY representable with aspect-oriented programming - solved by AOP programming language

-moved to implementation level

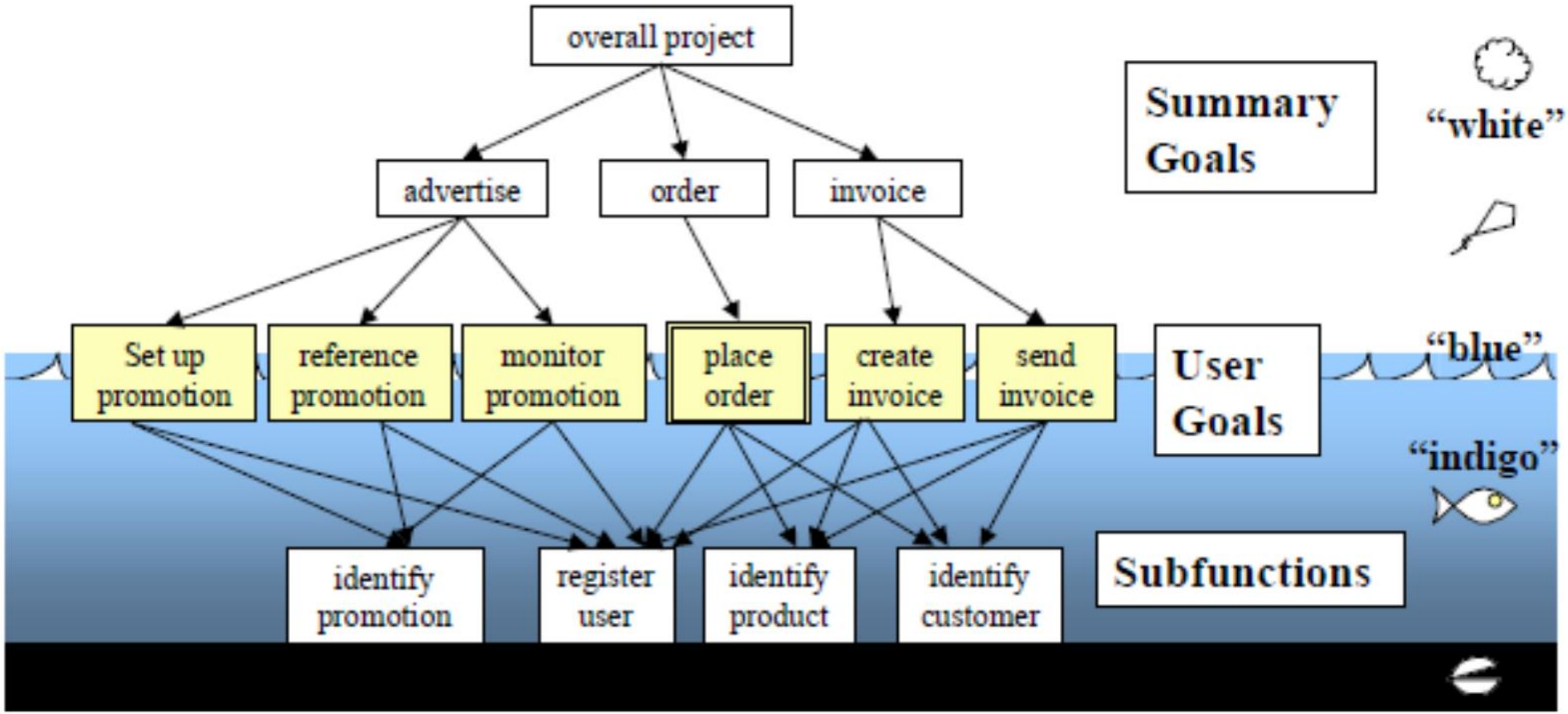
Prebrané z https://sparxsystems.com/images/screenshots/UCLogin.gif





Prebrané z [Bc. Pavol Michalco: PRÍPADY POUŽITIA A TÉMY V PRÍSTUPE THEME/DOC]

Similarly Tackling With Abstractions



Source: Cockburn, A.. *Writing Effective Use Cases*.Addison-Wesley, 2001.

A. Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.

Similarly Tackling With Abstractions

```
01 public abstract aspect ObserverProtocol {
02
03   protected interface Subject  { }
04   protected interface Observer { }
05
06   private WeakHashMap perSubjectObservers;07
08   protected List getObservers(Subject s) {
09     if (perSubjectObservers == null) {
10       perSubjectObservers = new WeakHashMap();
11     }
12     List observers =
13       (List)perSubjectObservers.get(s);
14     if ( observers == null ) {
15       observers = new LinkedList();
16       perSubjectObservers.put(s, observers);
17     }
18     return observers;
19   }
20
21   public void addObserver(Subject s,Observer o){
22     getObservers(s).add(o);
23   }
```

Example: Observer Pattern

ABSTRACTION:

*-adaptable to **various contexts***

=> **SUPPORTING REUSE**

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Similarly Tackling With Abstractions

```
24    public void removeObserver(Subject s,Observer o){
25      getObservers(s).remove(o);
26    }
27
28    abstract protected pointcut
29      subjectChange(Subject s);
30
31    abstract protected void
32      updateObserver(Subject s, Observer o);
33
34    after(Subject s): subjectChange(s) {
35      Iterator iter = getObservers(s).iterator();
36      while ( iter.hasNext() ) {
37        updateObserver(s, ((Observer)iter.next()));
38      }
39    }
40  }
```

Figure 2: The generalized ObserverProtocol aspect

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc.
of 17th ACM SIGPLAN Conference on Object-Oriented
Programming, Systems, Languages, and Applications,
OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Similarly Tackling With Abstractions

```
01 public aspect ColorObserver extends ObserverProtocol {
02
03   declare parents: Point  implements Subject;
04   declare parents: Line   implements Subject;
05   declare parents: Screen implements Observer;
06
07   protected pointcut subjectChange(Subject s):
08     (call(void Point.setColor(Color)) ||
09      call(void  Line.setColor(Color)) ) && target(s);
10
11   protected void updateObserver(Subject s,
12                                 Observer o) {
13     ((Screen)o).display("Color change.");
14   }
15 }
```

```
16 public aspect CoordinateObserver extends
17   ObserverProtocol {
18
19   declare parents: Point  implements Subject;
20   declare parents: Line   implements Subject;
21   declare parents: Screen implements Observer;
22
23   protected pointcut subjectChange(Subject s):
24      (call(void Point.setX(int))
25       || call(void Point.setY(int))
26       || call(void Line.setP1(Point))
27       || call(void Line.setP2(Point)) ) && target(s);
28
29   protected void updateObserver(Subject s,
30                                 Observer o) {
31     ((Screen)o).display("Coordinate change.");
32   }
33 }
```

Figure 3. Two different Observer instances.

DIFFERENT CONTEXTS

Source: J. Hannemann and G. Kiczales,
“Design pattern implementation in Java and AspectJ,” in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

TAKING OVER/PLAYING THE ROLES

```
01 public aspect ScreenObserver
02                extends ObserverProtocol {
03
04  declare parents: Screen implements Subject;
05  declare parents: Screen implements Observer;
06
07  protected pointcut subjectChange(Subject s):
08    call(void Screen.display(String)) && target(s);
09
10  protected void updateObserver(
11    Subject s, Observer o) {
12      ((Screen)o).display("Screen updated.");
13  }
14 }
```

Figure 4. The same class can be Subject and Observer

Source: J. Hannemann and G. Kiczales,
"Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Peer Use Cases

PEER USE CASES: *Use cases without binding between each other*

-> independent of each other

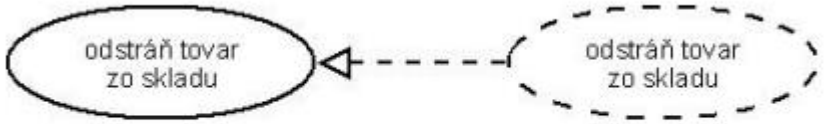
-> can be processed in paralell

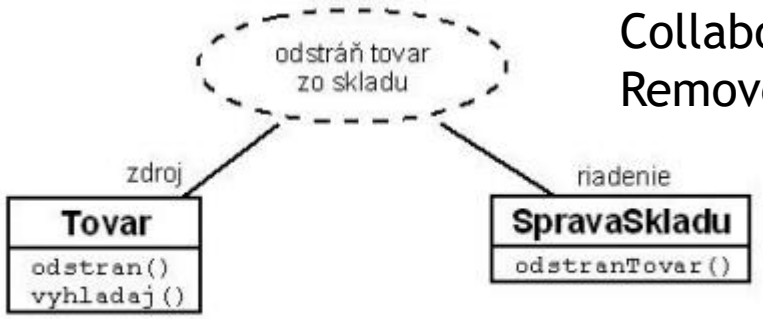
-> have an affect on shared/common entity

Separation of concerns can be problematic in peer/extension use cases

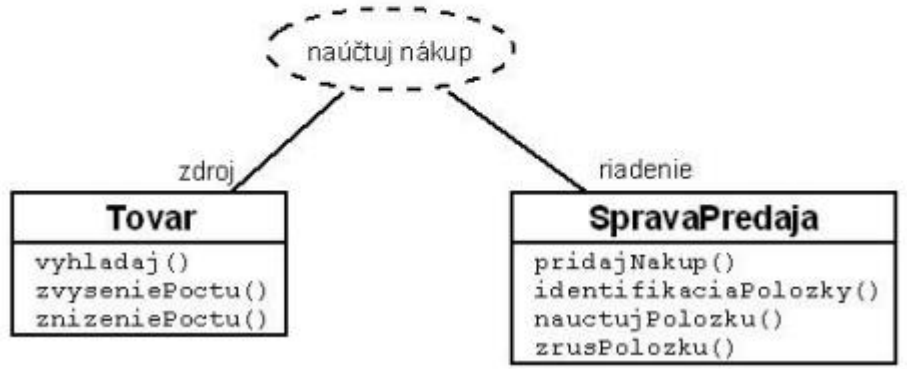
Collaboration diagrams:

Collaboration diagram: Accounting the purchase

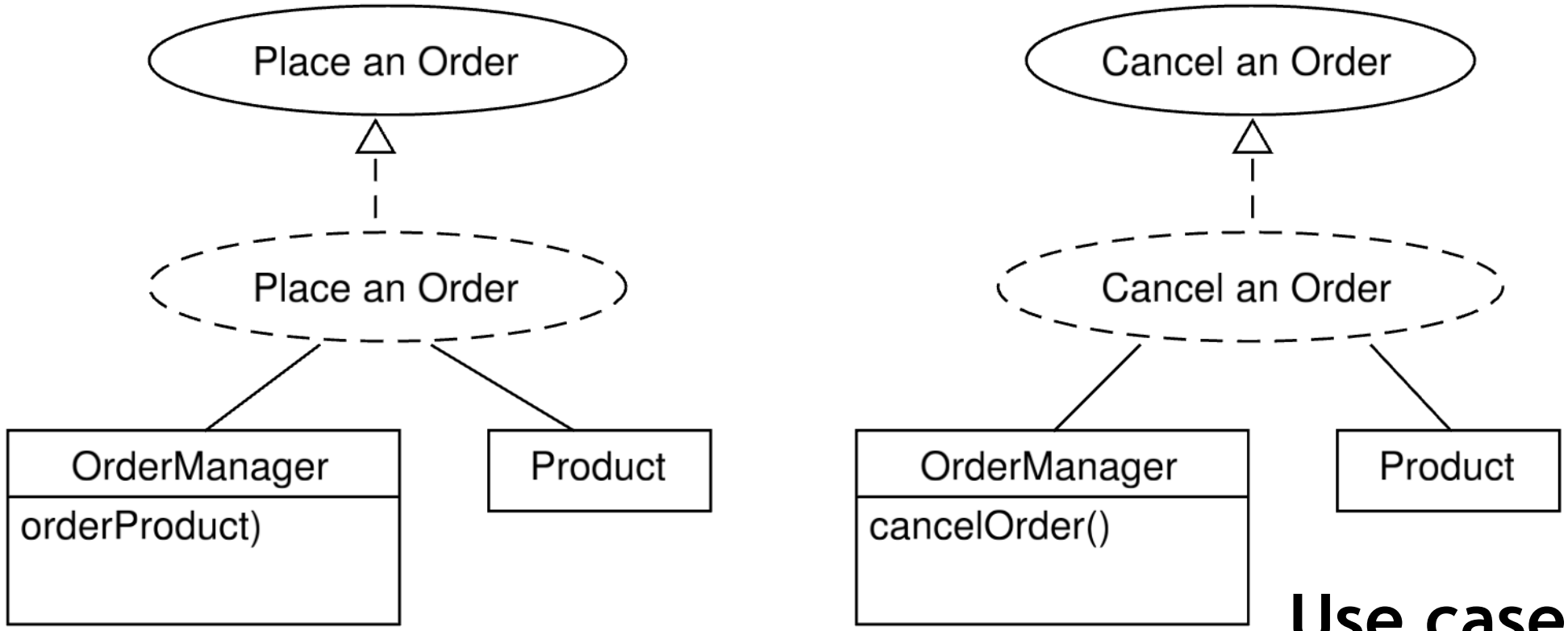


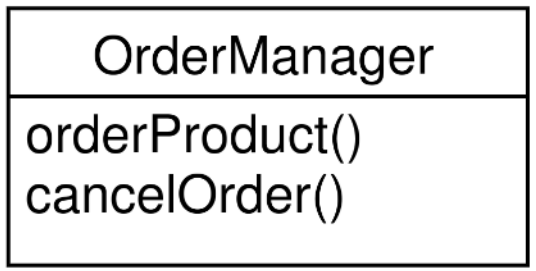


Collaboration diagram:
Remove goods from warehouse

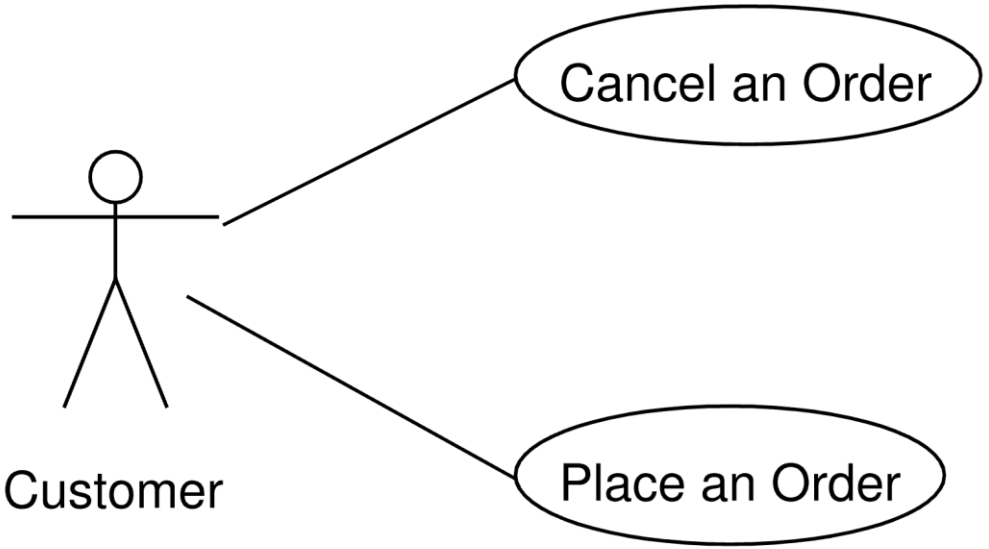


Zdroje z Bc. Pavol Michalco: PRÍPADY POUŽITIA A TÉMY V PRÍSTUPE THEME/DOC


Place an Order
Place an Order
OrderManager
orderProduct)
Product
Cancel an Order
Cancel an Order
OrderManager
cancelOrder()
Product


OrderManager
orderProduct()
cancelOrder()

Use case:


Cancel an Order
Customer
Place an Order

Dynamically extending class

Prototype-based programming

1. Declaring class

```
class VerticePair {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    getX() { return this.x; }
    getY() { return this.y; }
}
```

2. Instantiating class

```
var verticePair = new VerticePair(5, 6);
var newX = verticePair.x;
console.log(newX); //newX //(or) //to prints newX
```

3. Extending class dynamically

-possibly with new features… that can be then evolved independently

```
VerticePair.prototype.checkPoint = function() { if (this.x > 2) { throw new Error('Coordinate X is greater!');} }
```

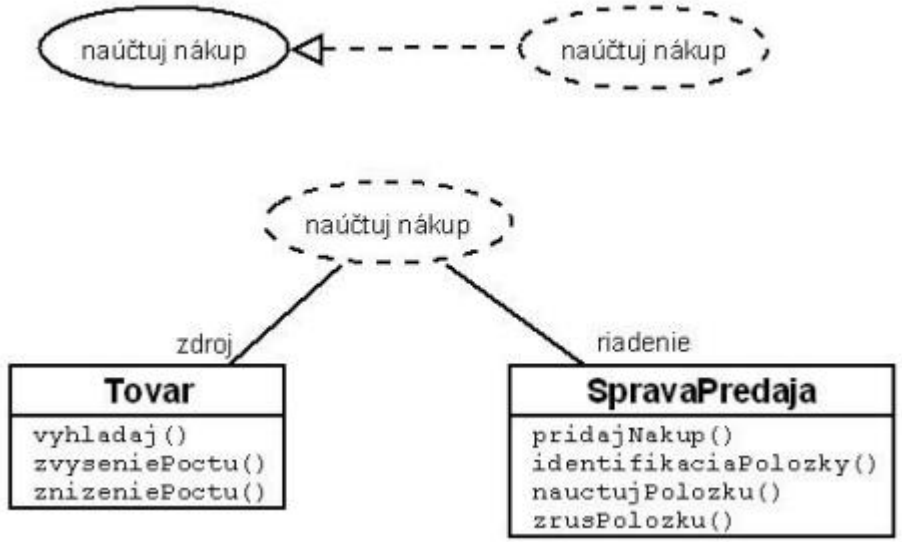
4. Using the extension

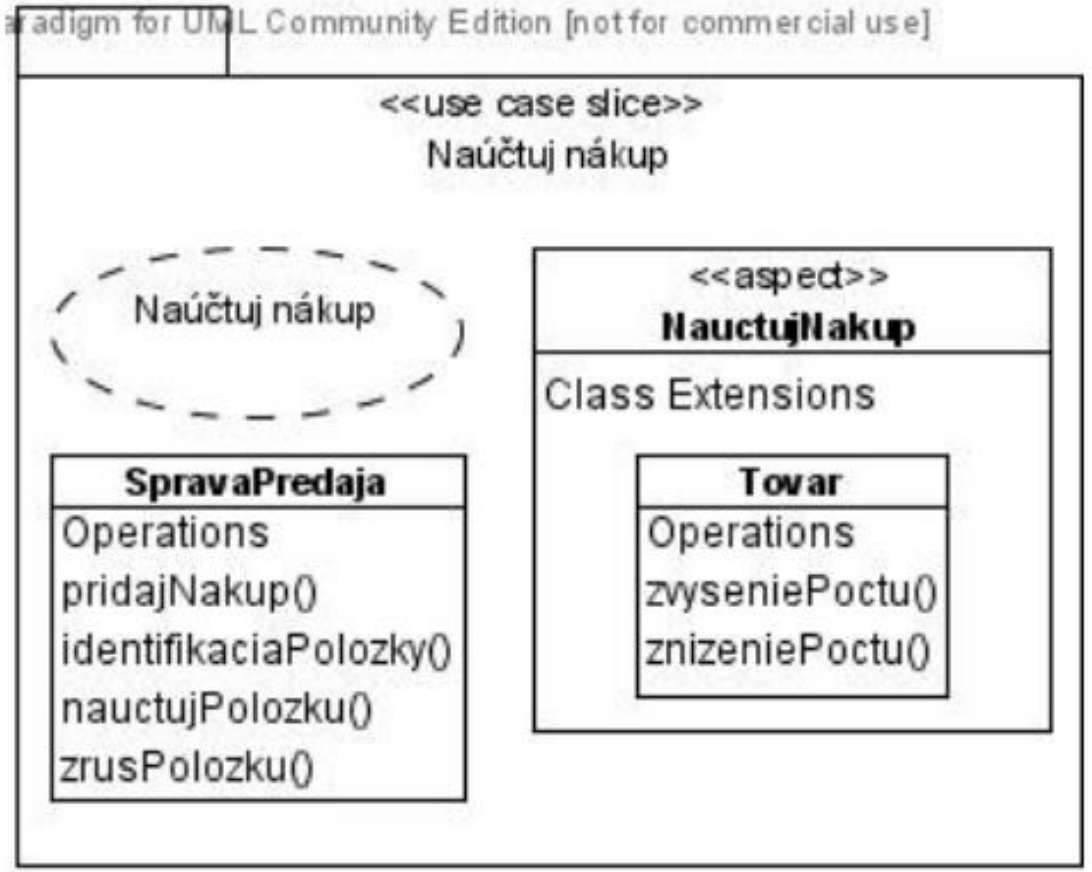
```
verticePair.checkPoint();
```

Solution to Peer Use Cases: *Intertype Declaration*

-we create use case slice…

…containing only specifics for this use case (Accounting the purchase in Figure)

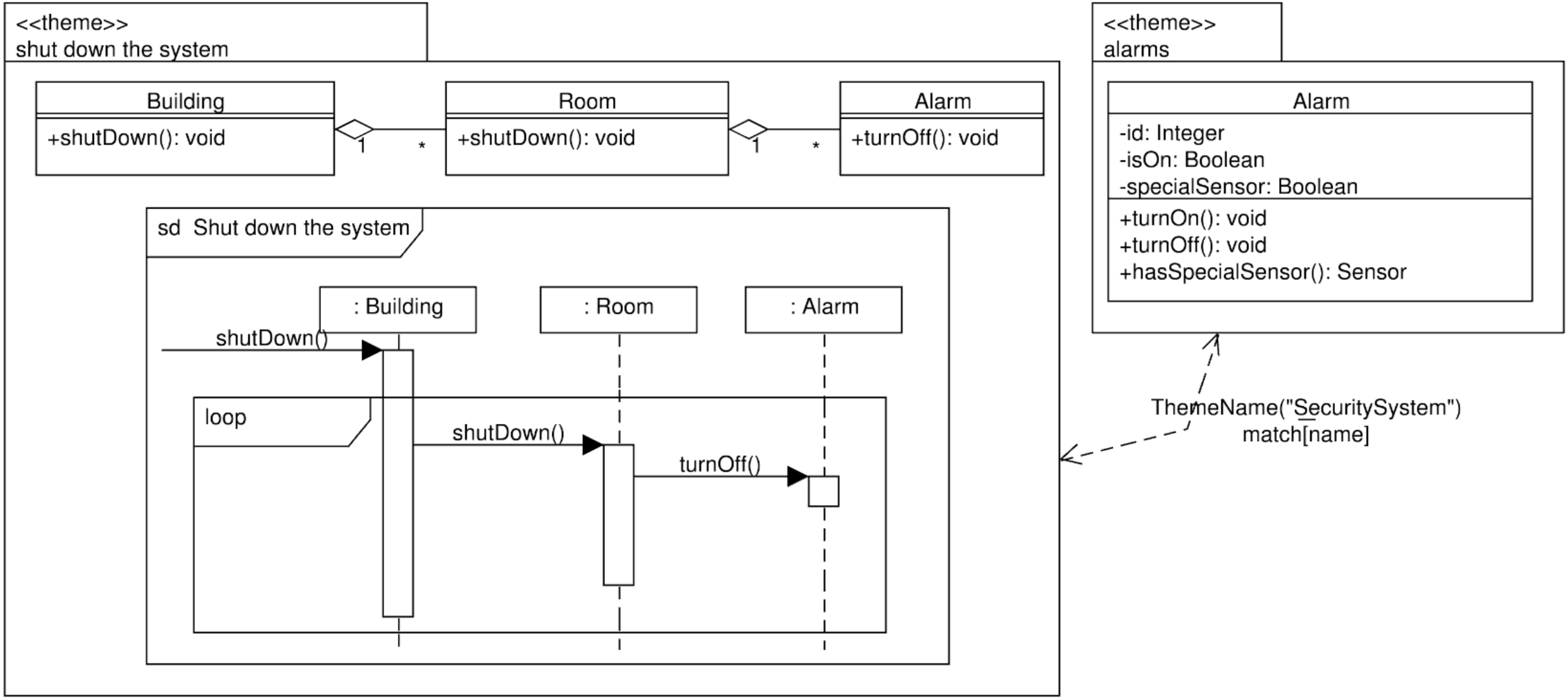




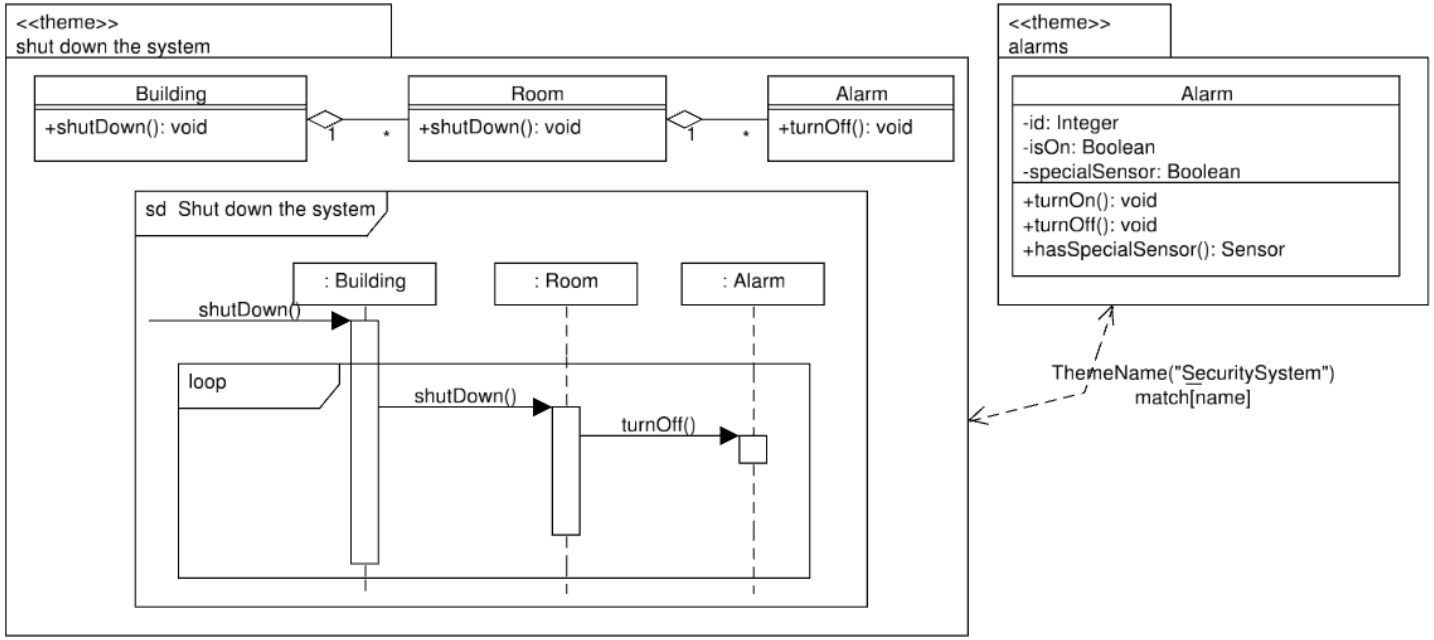
Zdroje z Bc. Pavol Michalco: PRÍPADY POUŽITIA A TÉMY V PRÍSTUPE THEME/DOC

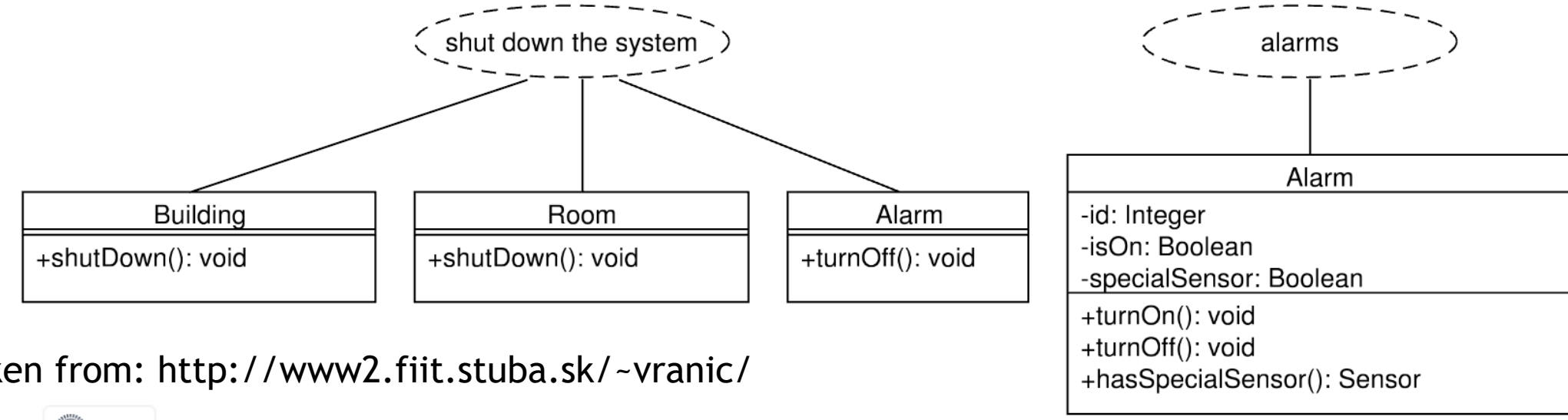
Use Cases Behind Themes

Example taken from: http://www2.fiit.stuba.sk/~vranic/



Use Cases Behind Themes





Example taken from: http://www2.fiit.stuba.sk/~vranic/

Are Use Cases and Themes the Same?

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

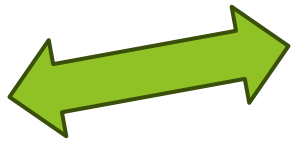
Transformation from the Themes/UML to use cases and vice versa

-directly applied in the majority of cases

Extensive similarities  **Differences**

- aspect-oriented decomposition
- relationship to functional decomposition
  - -crosscutting extend relationship
  - -functional decomposition as chain include relationship
- similar identification (of themes/use cases)
- -theme generalization/creating abstract use case

- lack of actors in themes
- naming conventions
- lower level character of some themes

Use Cases  **Themes**

-rather textual

-easily explainable

-not any functionality

-lack direct description

-hard to understand, requirements are needed

-can represent any functionality
(only holds for initial phase of their identification)

Example: Store Management

- Requirements

1. The application will record and maintain the product quantity in the stock in the central database.
2. The storekeeper can remove products from the database.
3. The storekeeper can add products into the database.
4. The storekeeper can change the product quantity in the database.
5. The cashier can bill the item by manually entering the bar code or with a bar code reader.
6. Only the products recorded in the database can be billed.
7. The billed items can be removed from the bill until it has been closed.
8. The billed item removal must be approved by a store manager by entering his authentication data.
9. The billed items will be printed on the cash desk bill as they are entered. The bill will consist of the store name, billed items, information on removed billed items, the total amount of money to be paid, and date and time.
10. The product price can be entered or modified only by a properly authenticated store manager.

Figure 1: The retail support application requirements.

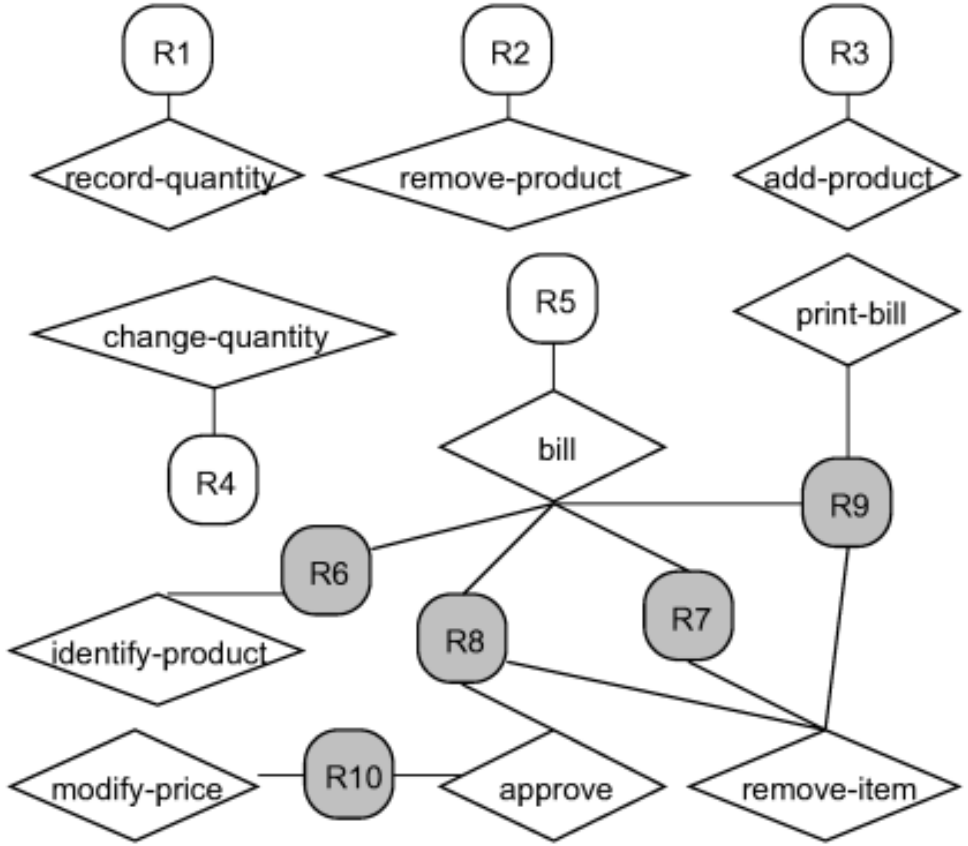


Figure 2: Themes in the retail support application.

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Themes/DOC - Views

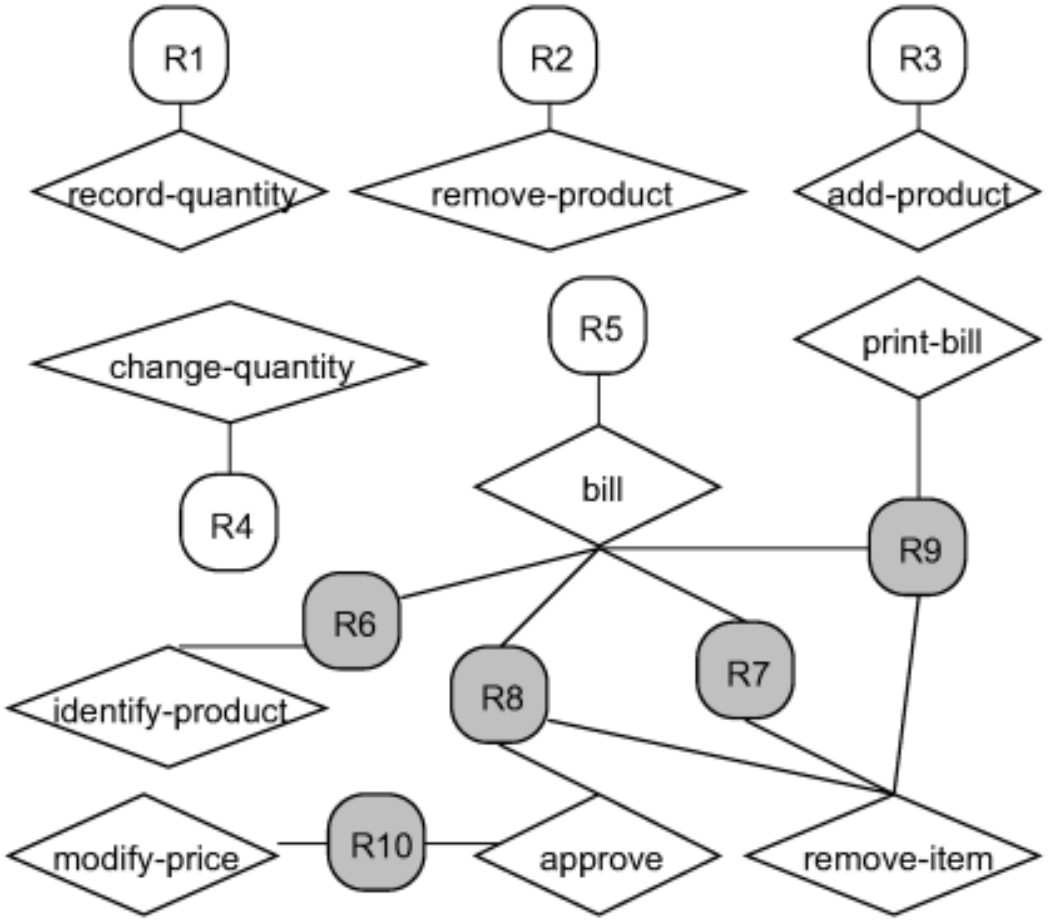


Figure 2: Themes in the retail support application.

Basic view

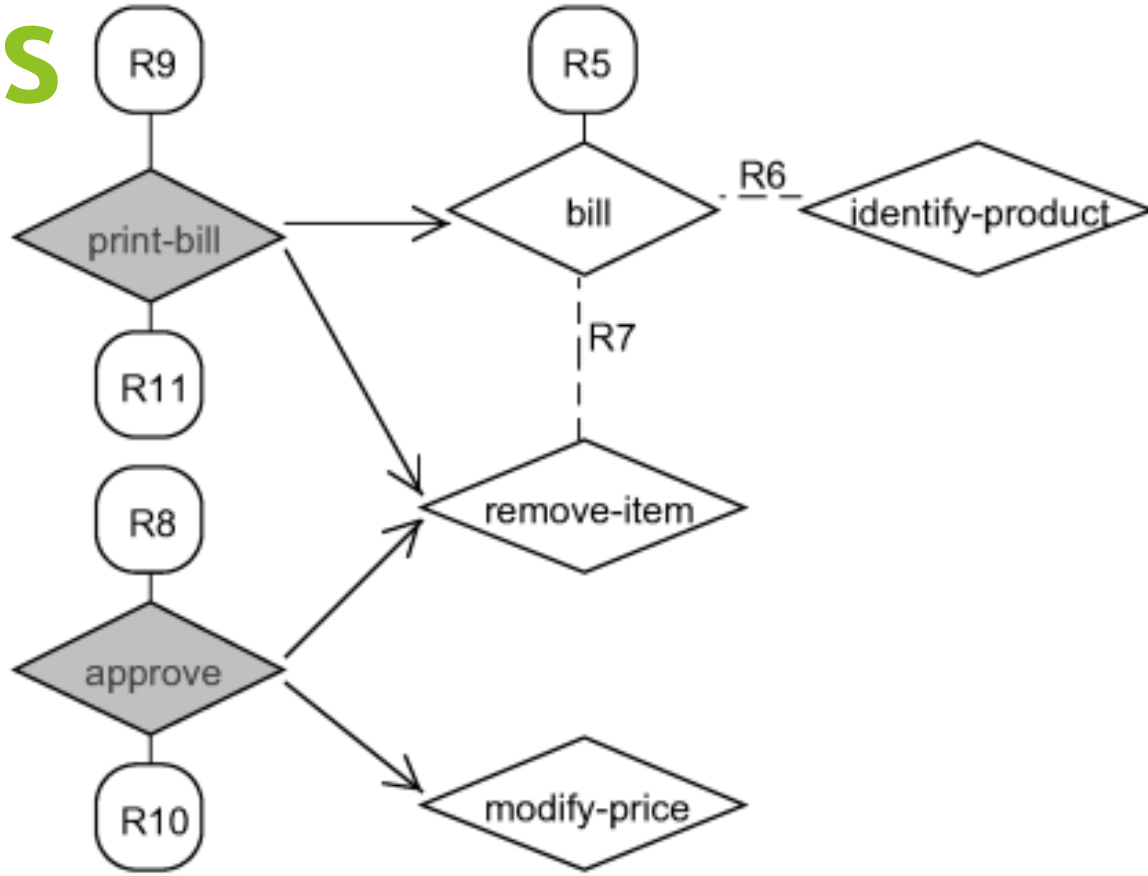


Figure 4: The crosscutting theme view.

Crosscutting view

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Transformation from Theme UML to Use Cases

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

- 1. Create a use case for each theme. Identify actors in requirements..
- 2. Create an extend relationship for each crosscut relationship found in the crosscutting view preserving its direction.
- 3. Consider splitting themes. Identify grouped themes in individual theme views (both the existing ones and those obtained in step 1). Consider transforming each theme-subtheme relationship into an include relationship or into a generalization relationship if the theme and subtheme conceptually represent the same theme. Deciding not to transform the subtheme means deciding its functionality will be an integral part of the existing use case possibly as a separate flow.
- 4. Consider unifying themes. Identify unified themes in the history of the operations performed upon the theme model if it is available. Consider transforming unified themes into generalizations.
- 5. Consider the granularity of the obtained use cases and restructure them as necessary by including too low level use cases as flows of regular ones.
- 6. If not resolved by previous steps, resolve the postponed relationships as include, extend, generalization, general relationship, or dismiss them.

Themes→Use Cases Transformation

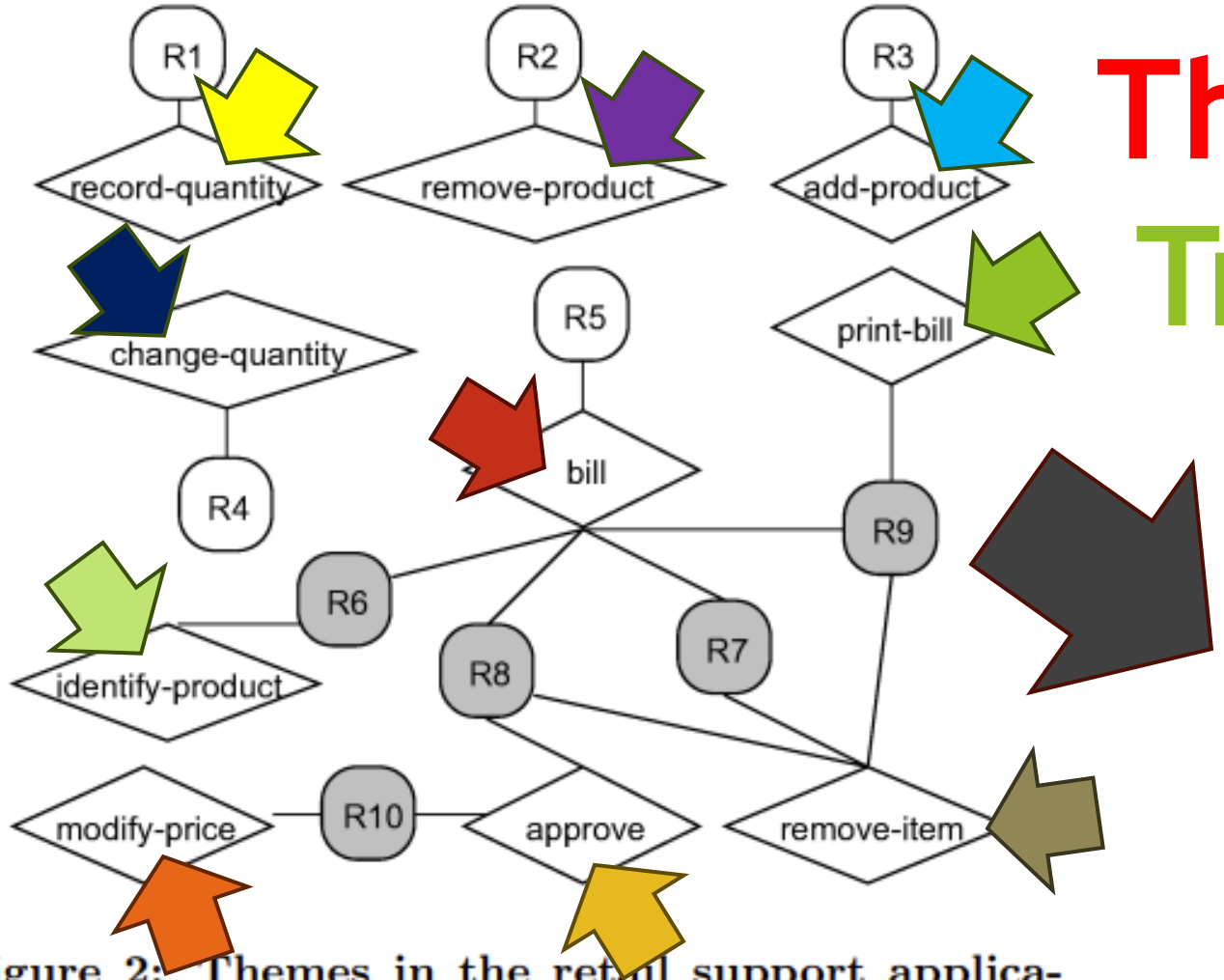
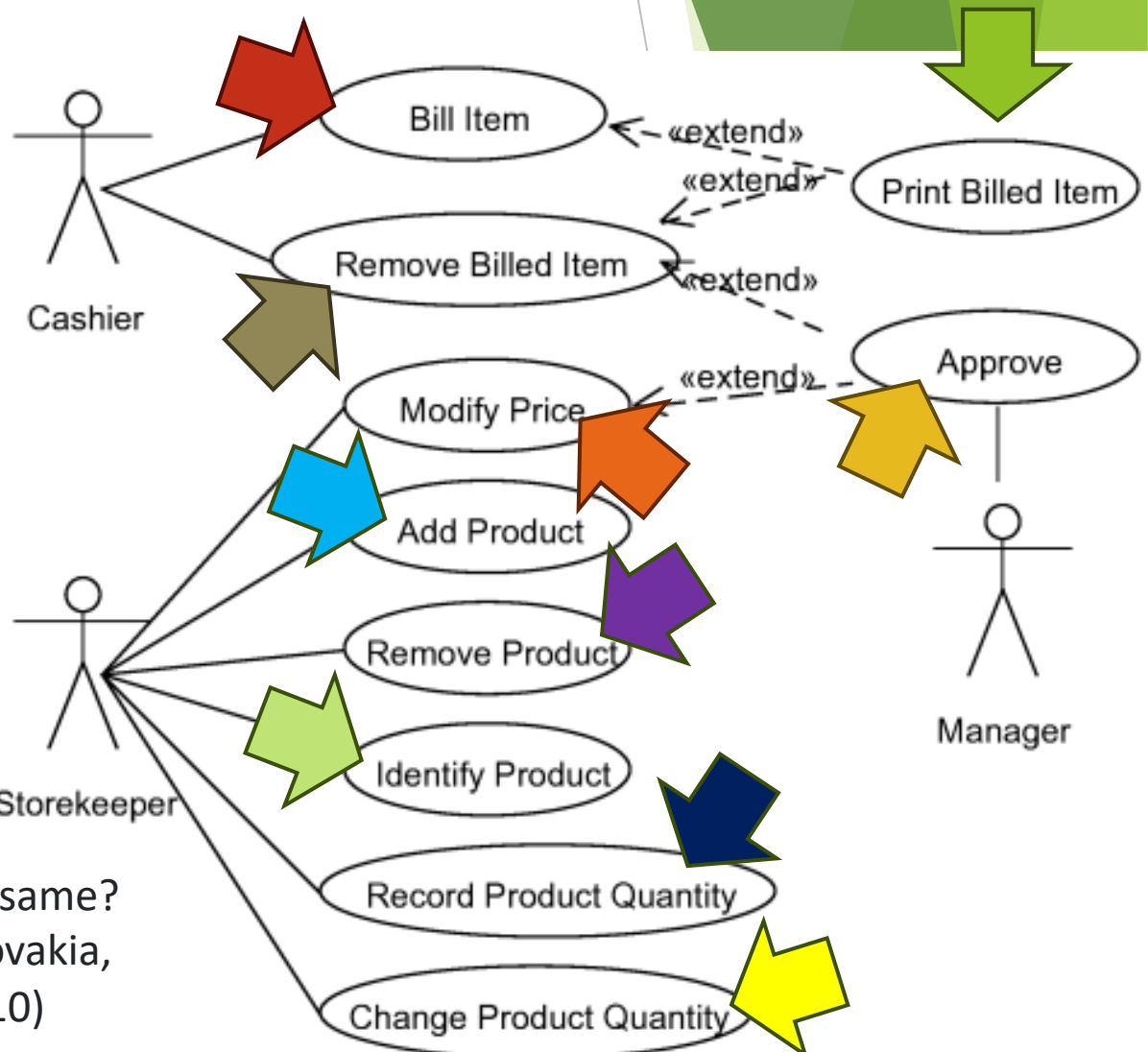


Figure 2: Themes in the retail support application.



1. Create a use case for each theme. Identify actors in requirements. ….

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Themes→Use Cases Transformation

1. The application will record and maintain the product quantity in stock in the central database.
2. The storekeeper can remove products from the database.
3. The storekeeper can add products into the database.
4. The storekeeper can change the product quantity in the database.
5. The cashier can bill the item by manually entering the bar code or with a bar code reader.
6. Only the products recorded in the database can be billed.
7. The billed items can be removed from the bill until it has been closed.
8. The billed item removal must be approved by a store manager by entering his authentication data.
9. The billed items will be printed on the cash desk bill as they are entered. The bill will consist of the store name, billed items, information on removed billed items, the total amount of money to be paid, and date and time.
10. The product price can be entered or modified only by a properly authenticated store manager.

Figure 1: The retail support application requirements.

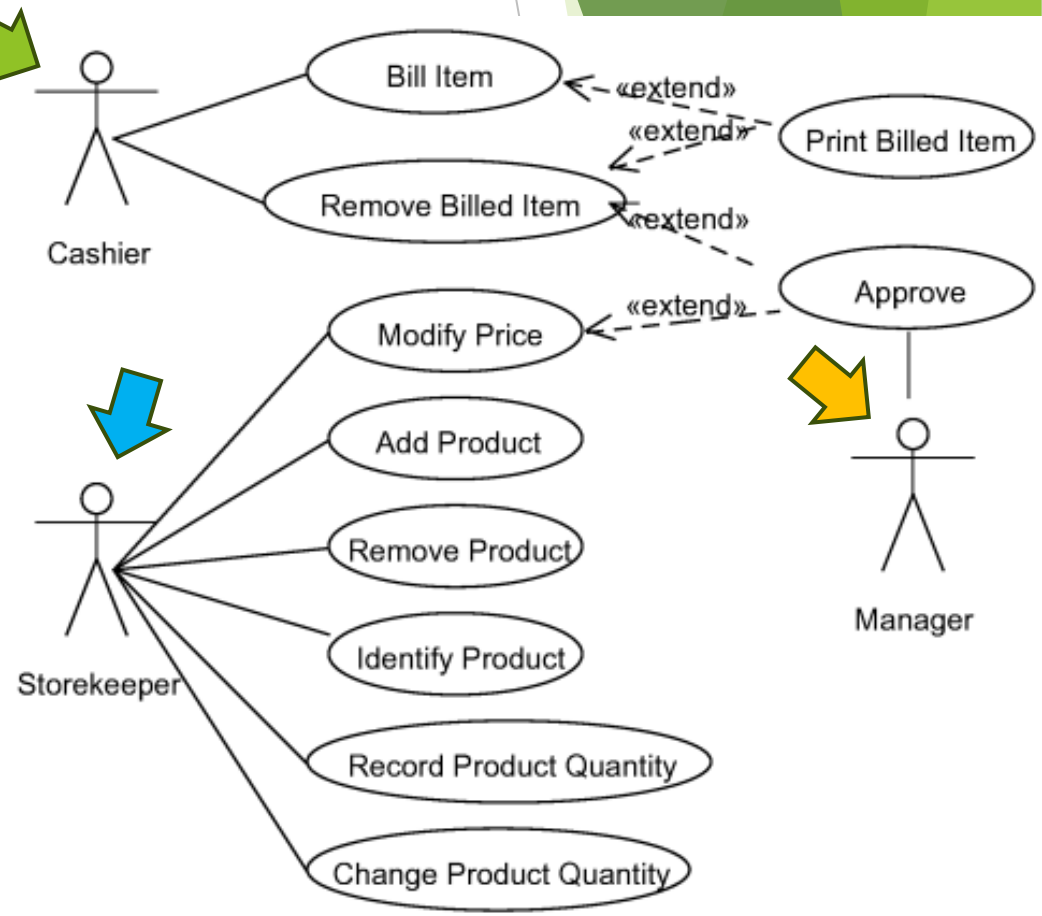


Figure 5: Identified use cases and extend relationships.

1. Create a use case for each theme. Identify actors in requirements. ….

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Themes→Use Cases Transformation

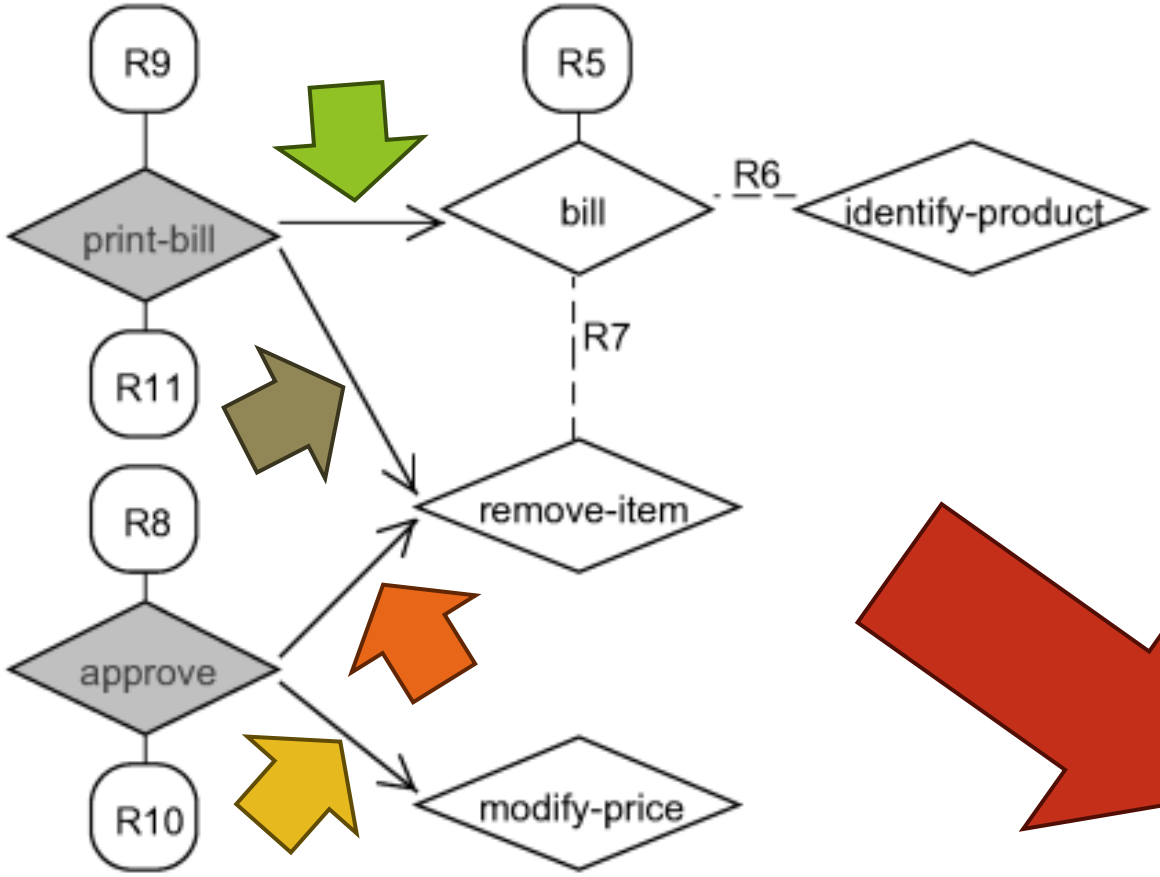
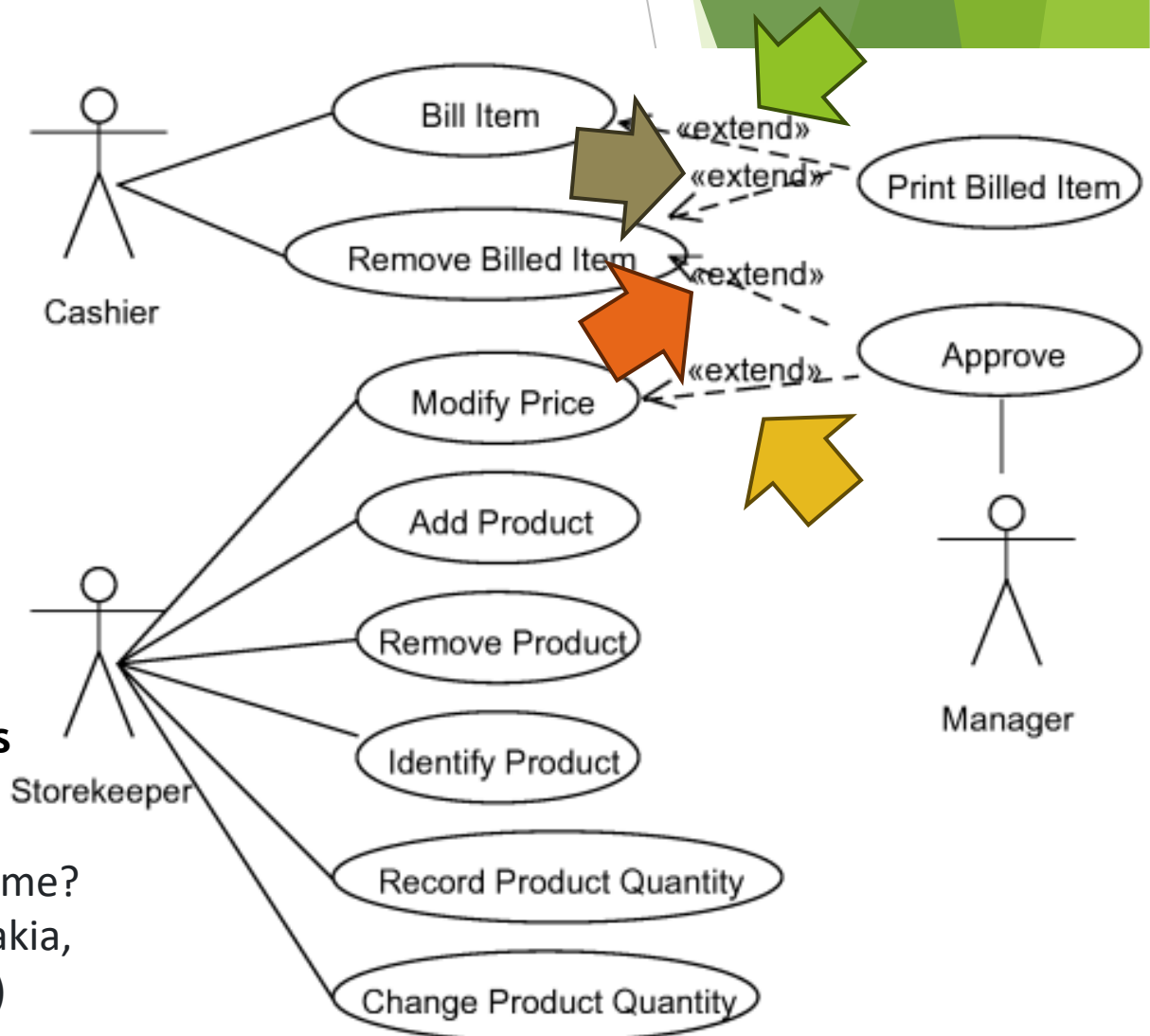


Figure 4: The crosscutting theme view.



2. Create an extend relationship for each crosscut relationship found in the crosscutting view preserving its direction. ….

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Themes→Use Cases Transformation

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

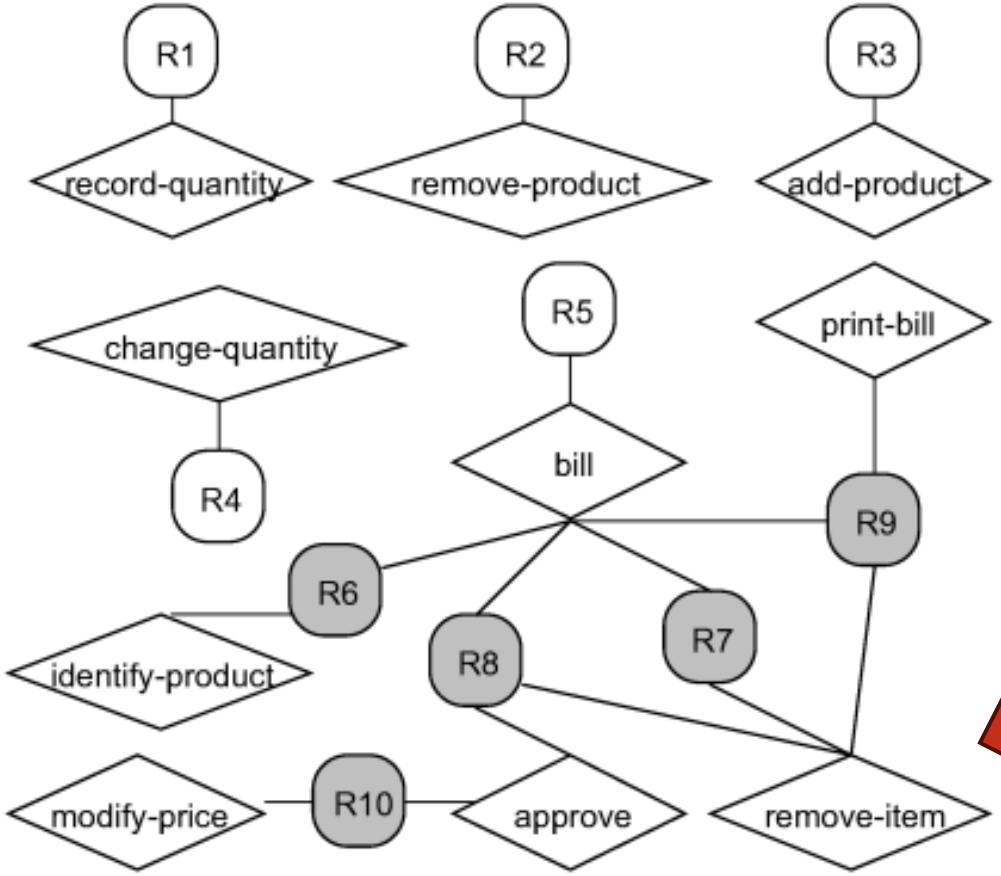


Figure 2: Themes in the retail support application.

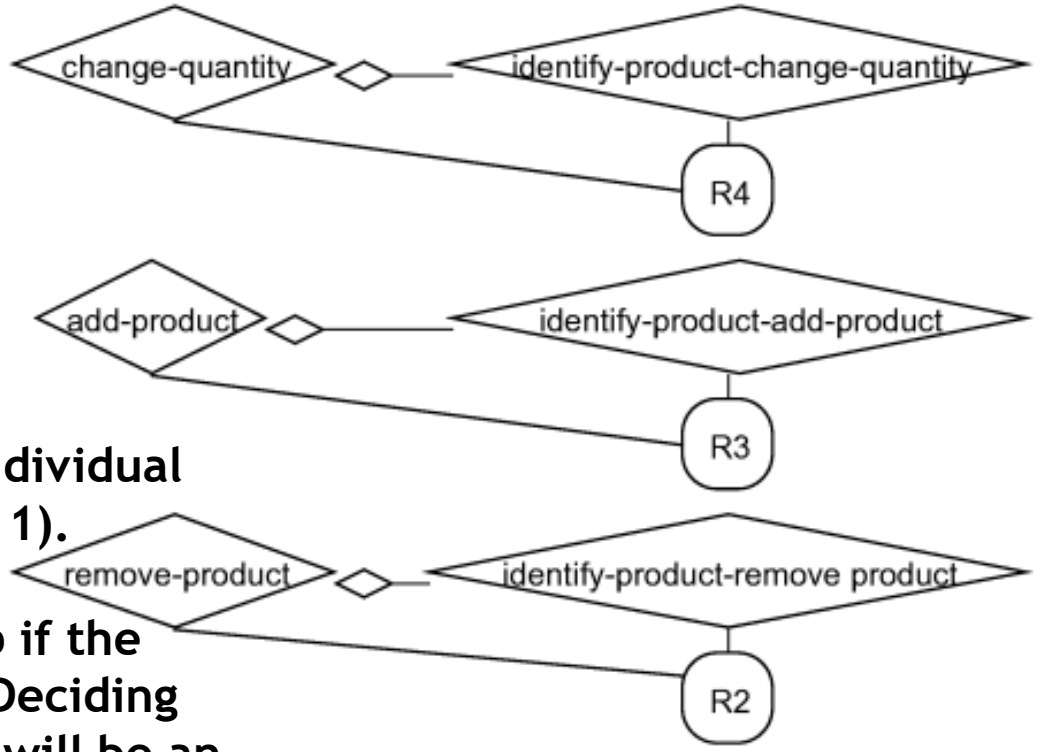


Figure 6: Splitting the themes.

3. Consider splitting themes. Identify grouped themes in individual theme views (both the existing ones and those obtained in step 1). Consider transforming each theme-subtheme relationship into an include relationship or into a generalization relationship if the theme and subtheme conceptually represent the same theme. Deciding not to transform the subtheme means deciding its functionality will be an integral part of the existing use case possibly as a separate flow.

Themes→Use Cases Transformation

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

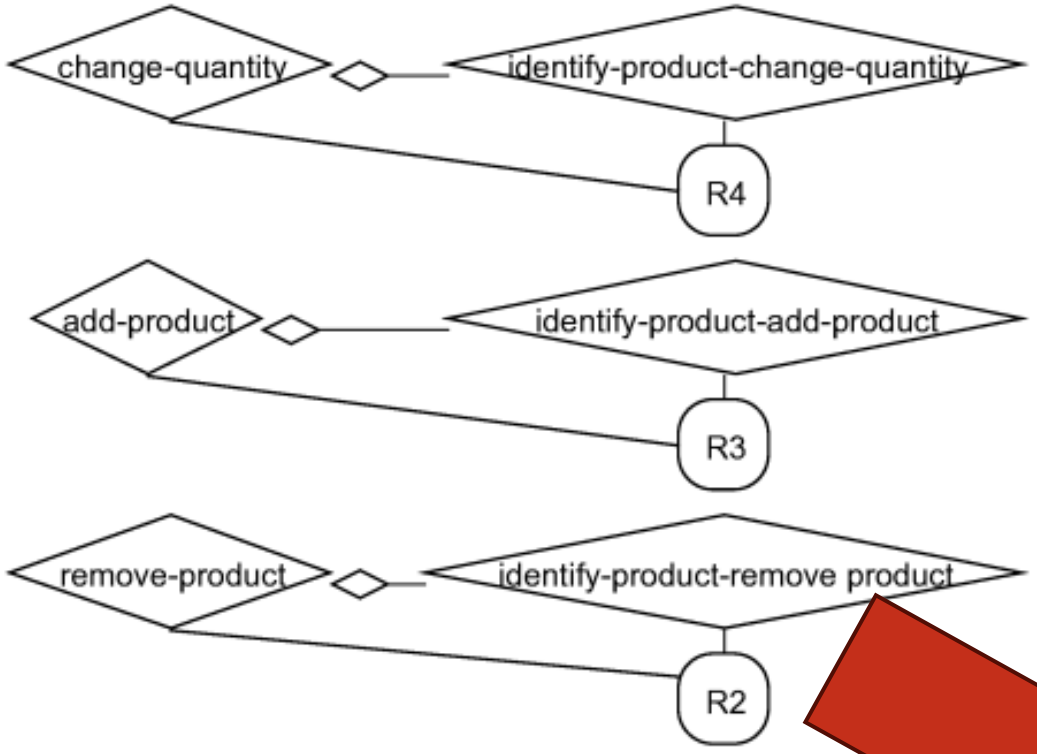


Figure 6: Splitting the themes.

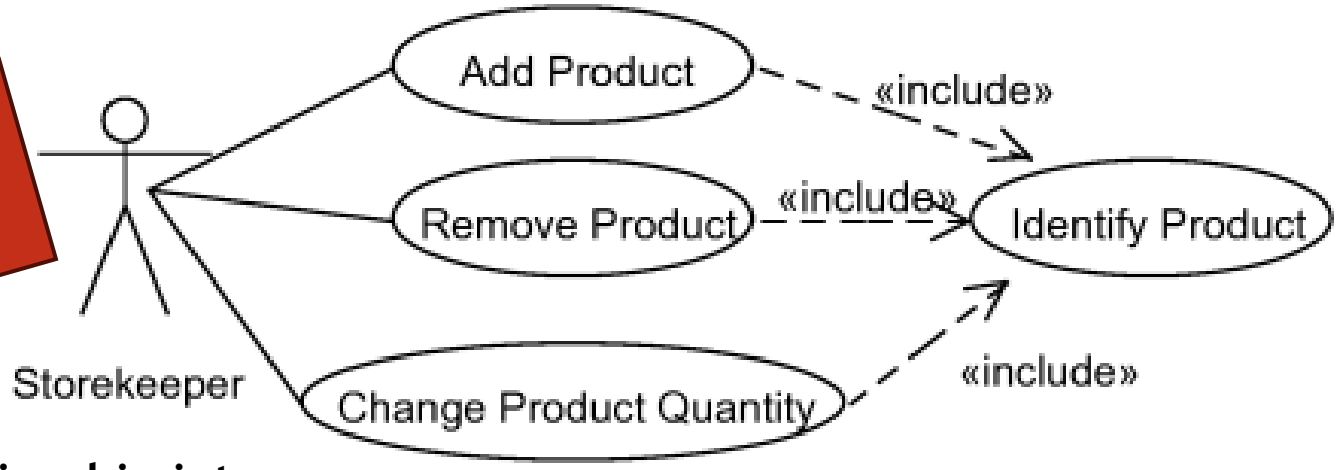


Figure 7: Inclusion.

3. Consider splitting themes. Identify grouped themes in individual theme views (both the existing ones and those obtained in step 1). Consider transforming each theme-subtheme relationship into an include relationship or into a generalization relationship if the theme and subtheme conceptually represent the same theme. Deciding not to transform the subtheme means deciding its functionality will be an integral part of the existing use case possibly as a separate flow.

Inclusion

Getting include relationship

Themes→Use Cases -generalization

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

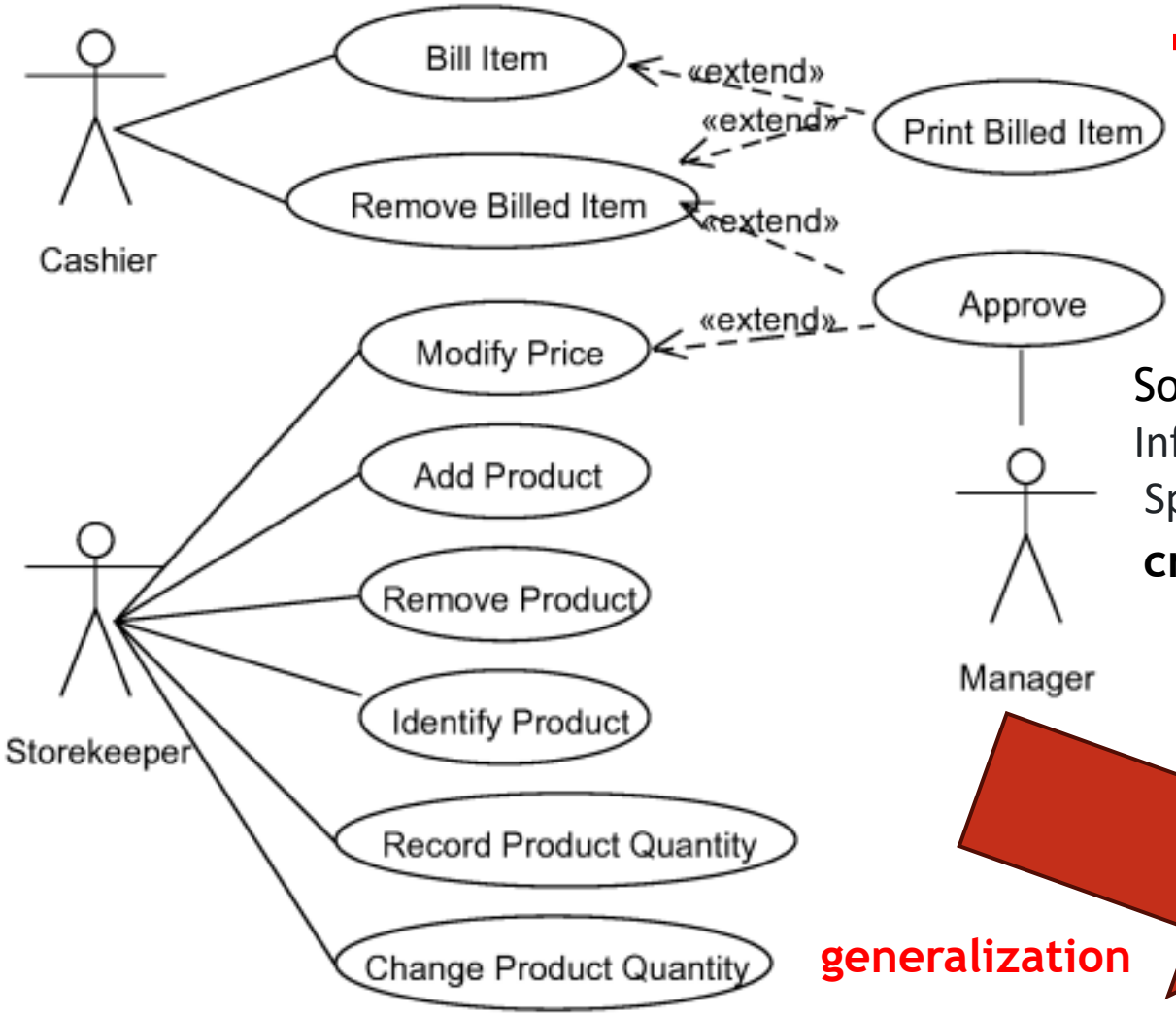


Figure 5: Identified use cases and extend relationships.

creating abstract use case

generalization

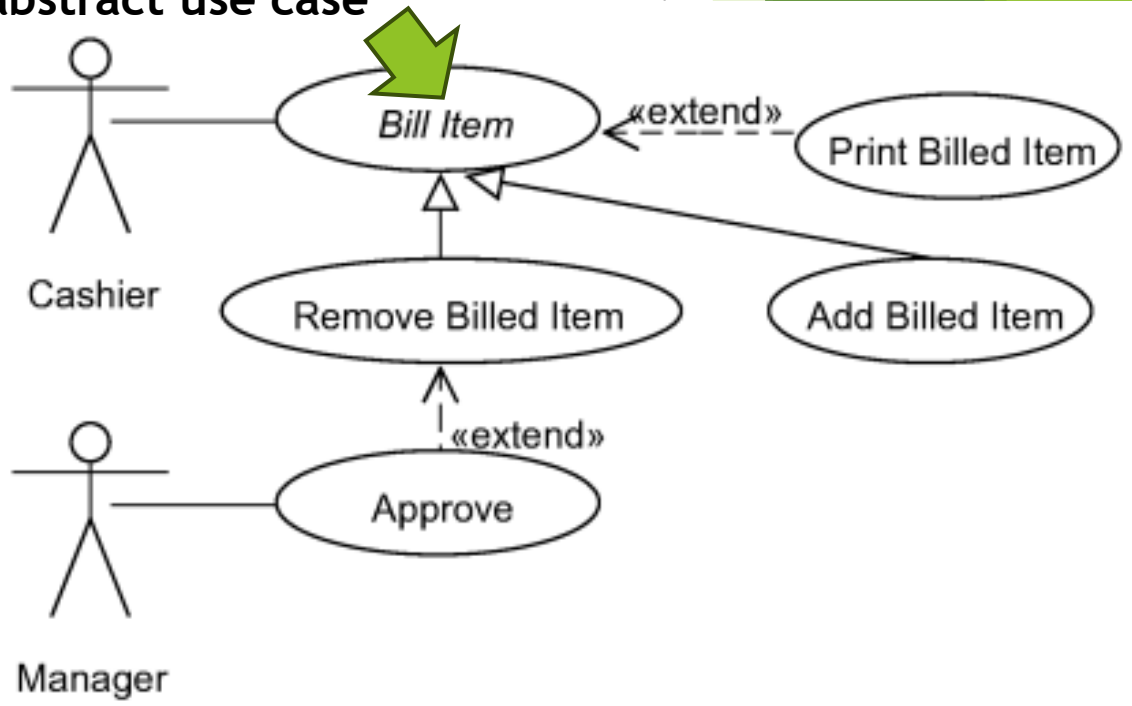


Figure 8: Generalization of use cases.

4. Consider unifying themes. Identify unified themes in the history of the operations performed upon the theme model if it is available. Consider transforming unified themes into generalizations. ….

Transformation From Use Cases to Themes/UML

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

- 1. Identify themes by transforming each use case not involved in a generalization into a theme and transforming each generalization among use cases into unified themes. Optionally rename themes by shortening the corresponding use case names. Drop actors.
- 2. Create the crosscutting view by transforming each extend relationship between use cases into a crosscutting relationship between the corresponding themes preserving its direction.
- 3. Create the individual view by transforming each include relationship between use cases into a theme-subtheme relationship preserving its direction. Derive the data entities the theme operates on from the use case flows and attach them to the corresponding themes.
- 4. Transform all requirements use cases refer to into requirements in the theme model. Transform each use case to requirement relationship into a relationship between the corresponding theme and requirement.
- 5. Derive the theme-relationship view by including all the themes in the crosscutting view and identifying shared requirements. Transform each unspecified dependency between use cases into a postponed relationship between the corresponding themes preserving its direction.

Equivalence

Table 1: Equivalence of Theme/Doc and use case modeling mechanisms.

Theme/Doc	Use Case Modeling
base theme	peer use case
requirement	brief description/flow
crosscutting theme	extending use case
grouping theme	including use case
grouped theme	included use case
unifying theme	general use case
unified theme	special use case
subtheme	inclusion use case
crosscutting relationship	extend relationship
theme–subtheme rel.	include relationship
theme–requirement rel.	use case to requirement link
postponed relationships	any/no relationship
n/a	actor

Source: Vranic, V., Michalco, P.: Are themes and use cases the same? Information Sciences and Technologies, Bulletin of the ACM Slovakia, Special Section on Early Aspects at AOSD 2010 2(1),66–71 (2010)

Handling Include Relationship From Use Cases

UC Place an Order

Basic Flow: Place an Order

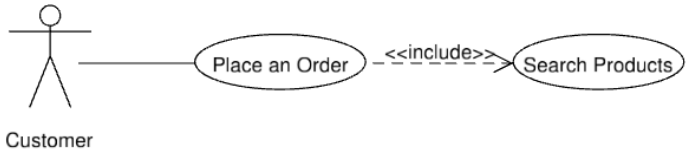
1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.

Example taken from: http://www2.fiit.stuba.sk/~vranic/

UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.



Handling Include Relation From Use Cases

In themes: individual views of theme-subtheme relationship with preserving its direction

```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
    }
    ...
}
```

Grouped themes/ subthemes

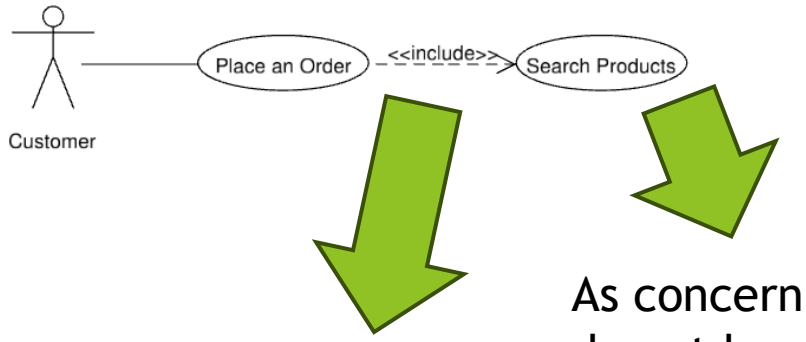
-resemble include relationship between use cases.

Example taken from: http://www2.fiit.stuba.sk/~vranic/

UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.



Some kind of weaving

As concern that do not know about Place An Order concern

```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
    }
    ...
}
```

Example taken from: http://www2.fiit.stuba.sk/~vranic/

Handling Extend Relationship From Use Cases

UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.

Extension points:

- *Checking Product Availability*: Step 4

Example taken from: http://www2.fiit.stuba.sk/~vranic/

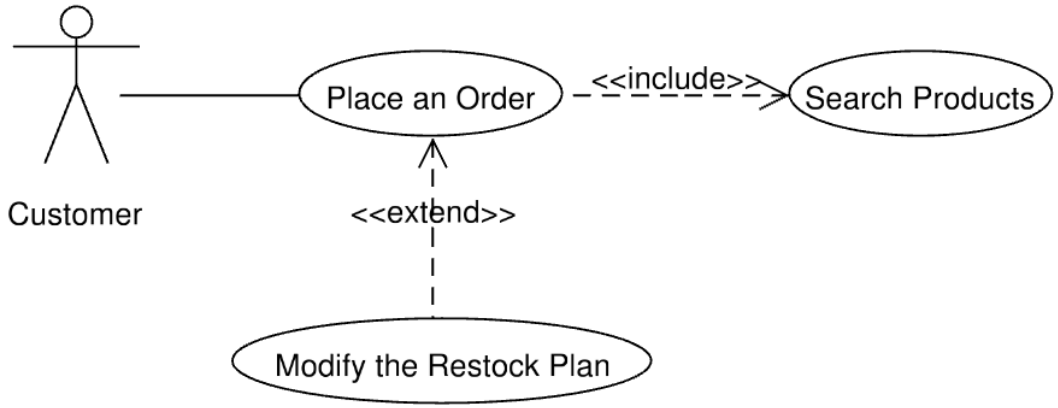
UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.

Extension points:

- *Checking Product Availability*: Step 4



UC Modify the Restock Plan

Alternate Flow: Modify the Restock Plan

After the *Checking Product Availability* extension point of the Place an Order use case:

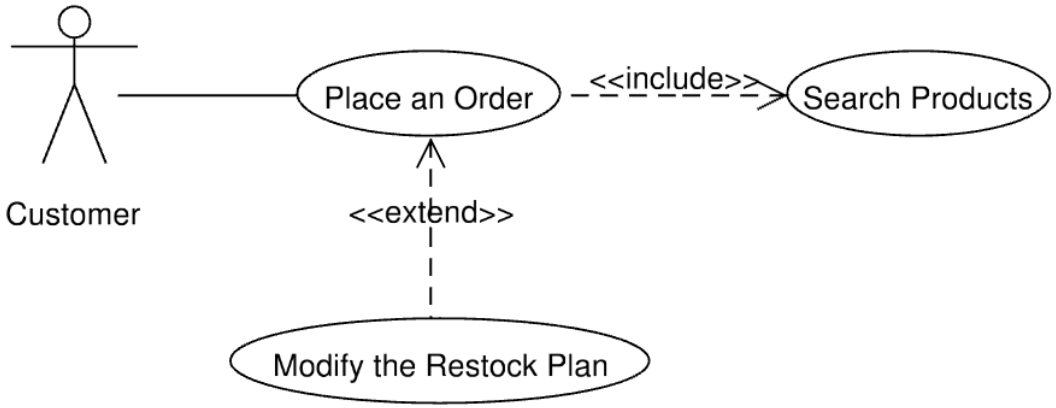
1. System checks the available quantity of the product being ordered.
2. If the quantity is below the limit, System adds the quantity under demand to the restock plan.
3. The flow continues with the step that follows the triggering extension point.

Example taken from: http://www2.fiit.stuba.sk/~vranic/

UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC *Search Products* is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.



Extension points:

- *Checking Product Availability*: Step 4

After Step 4:

UC Modify the Restock Plan

Alternate Flow: Modify the Restock Plan

After the *Checking Product Availability* extension point of the Place an Order use case:

1. System checks the available quantity of the product being ordered.
2. If the quantity is below the limit, System adds the quantity under demand to the restock plan.
3. The flow continues with the step that follows the triggering extension point.

Example taken from: http://www2.fiit.stuba.sk/~vranic/

Object-Oriented Implementation

```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
        if (productAvailable(product)) {
            ...
        } else...
    }
    ...
```

Example taken from: http://www2.fiit.stuba.sk/~vranic/

Aspect-Oriented Implementation

```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
        if (productAvailable(product)) {
            ...
        } else...
    }
    ...
}
```

```
public aspect RestockPlan {
    ...
    void around(Product product):
        call(* Ordering.productAvailable(..) && args(tovar) {

        // increase the quantity in the restock plan
        ...
    }
    ...
}
```

Example taken from: http://www2.fiit.stuba.sk/~vranic/

Use Cases Preserved in Code

Thanks to Aspects:

Symmetric Aspect-Oriented Modularization

Peer Use Cases

Use Case Extending Another

Asymmetric Aspect-Oriented Modularization

OORam: Object-Oriented Role Analysis And Modeling

ROLE MODELS

⇒ *Permits inheritance of object properties*

-beneficially applicable in context of frameworks

=> *permits inheritance of system properties:*
-behavior
-constraints

OBJECT-ORIENTATION

Representation as Structure of Interacting objects

Designed for reuse

➡ Synthesis + Ensemble of classes designed for subclassing

KEY ABSTRACTIONS

➡ ***Common class abstraction***
(can be perceived as entity relation modelling)

➡ ***Type abstraction***
(interfaces, externa properties of objects)

➡ ***Role abstraction***
(structure and activities in patterns of interacting objects)

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

OORam: Object-Oriented Role Analysis And Modeling

- Developed by Trygve Reenskaug (MVC pattern), Taskon
- To describe complex software systems (software product lines fall within this category)
- For requirements supporting rapid construction of specialized software

Capturing the synergy of particular pattern of the interdependent parts

their value (the value of a system) is greater

than the sum of the values of its (interdependent) parts

Abstractions for Concerns Separation and Reuse

Type abstraction

(interfaces, externa properties of objects)

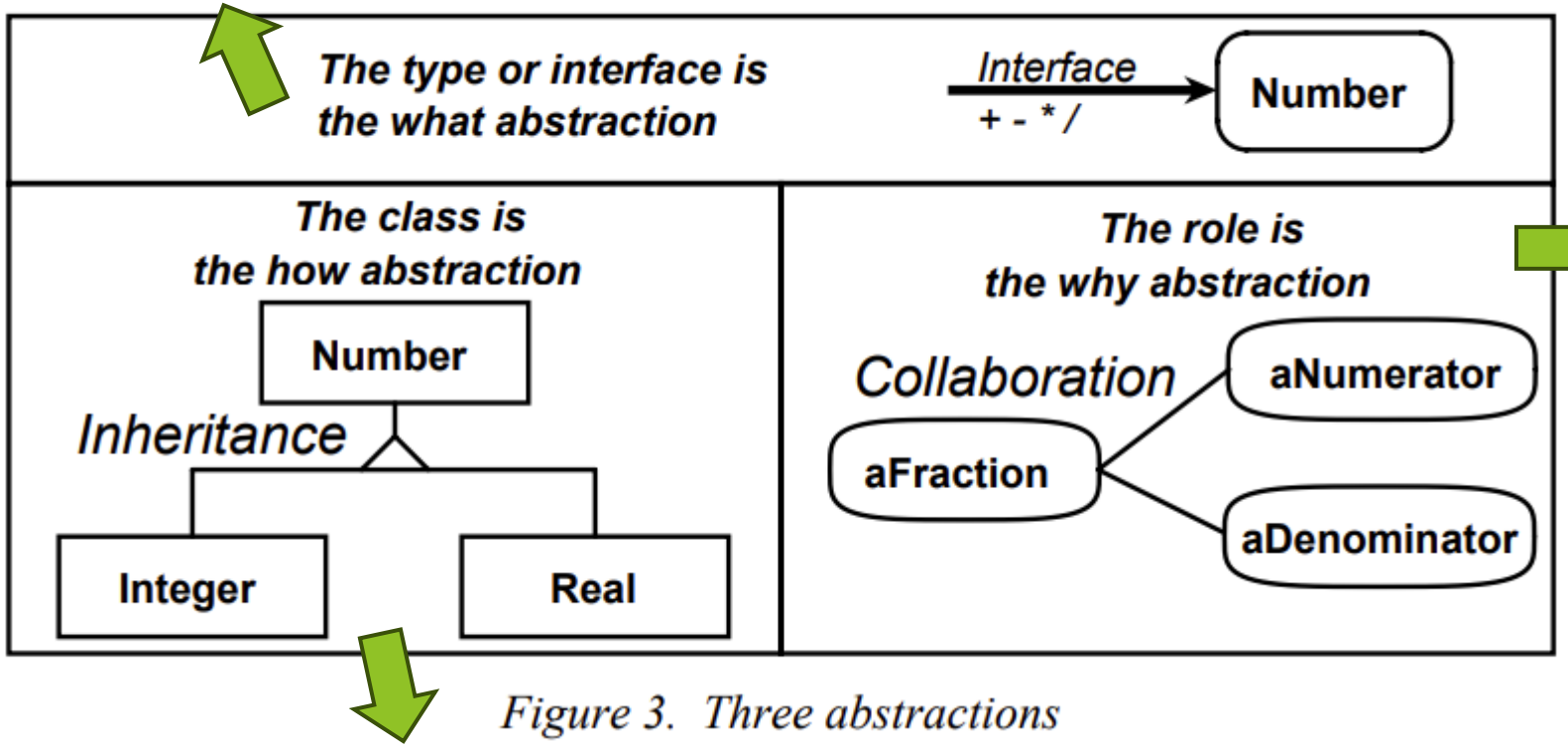


Figure 3. Three abstractions

Role abstraction

(structure and activities in patterns of interacting objects)

Common class abstraction

(can be perceived as entity relation modelling)

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Abstractions for Concerns Separation and Reuse

Type abstraction

Type

⇒ *Implementation-interdependent description of a set of objects*

- which **share external properties** (set of messages to understand)

-organized in type hierarchy

subtype ← → **supertype**

Liskov' Principle: all properties inherited/exhibits all messages
+ introduces new one

Reusable interfaces - ***used to expose functionality of components***

⇒ *Reusable components (encapsulated + hidden implementation details)*

⇒ *applicable in different contexts*

Abstractions for Concerns Separation and Reuse

Class abstraction

Class

⇒ *A set of objects sharing common implementation*

-meaning in programming:

-program controlling **class instances == the properties of a set of objects**

- allows to share concepts of code + supports reusable code

-organized in class hierarchy

subclass ← → superclass

Liskov' Principle: all code inherited/exhibits all messages + introduces or modified new methods and object variables

Reusable class libraries

⇒ *Reuse through application of classes*

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Abstractions for Concerns Separation and Reuse

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Role abstraction

Describes all static and dynamic properties of framework

Use Cases = Activities = Functionality realized by patterns of collaborating objects

Activity

⇒ *Task conducted by set of associated and cooperating objects*

In Role model

⇒ *Object identity is preserved => Object interaction patterns are preserved*

- **each role represents single object doing certain activities**
  - **such role represents only related object properties to these activities**

In Role model

- describes how patterns of objects perform specific task

Role

⇒ *Partial description/specification of corresponding object [, partial description of corresponding class]*

Abstractions for Concerns Separation and Reuse

Understanding complex systems

-good to separate concerns

DIVIDE AND CONQUER

-too complex whole

Role model synthesis

SELECTING **DESCRIBING**

Area of concern => objects playing roles within its context

System derivation:

Composition from base systems

CONTROLLABLY PRESERVING OBJECT PATTERNS AND ACTIVITIES

-> SYNTHESIZING REUSABLE COMPONENTS

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Interdependencies between subjects

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Managed subject interdependencies by objects

As Integration of each object behavior when playing multiple roles:

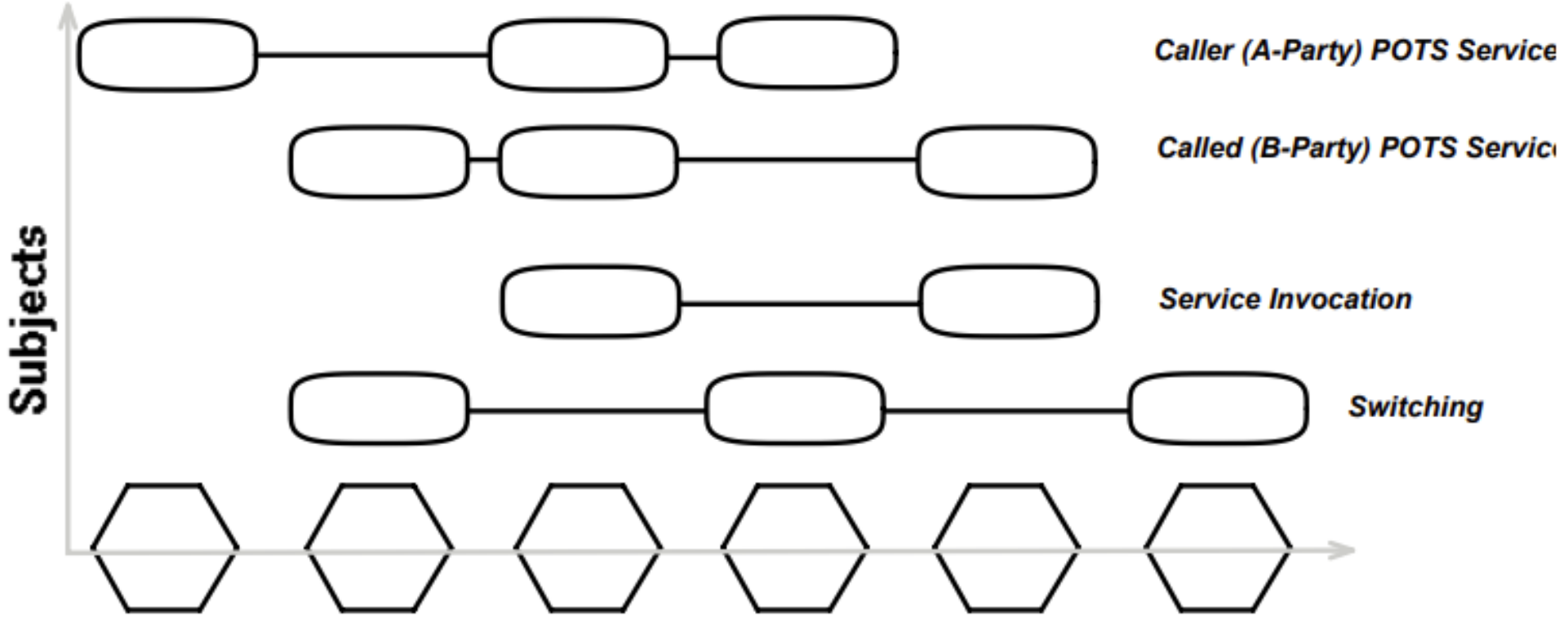


Figure 6. Subjects and objects

As role models

Each plays one or more roles

Not bound to single model

Many View of One Model

-to show static and dynamic properties of the system

1. Collaboration View

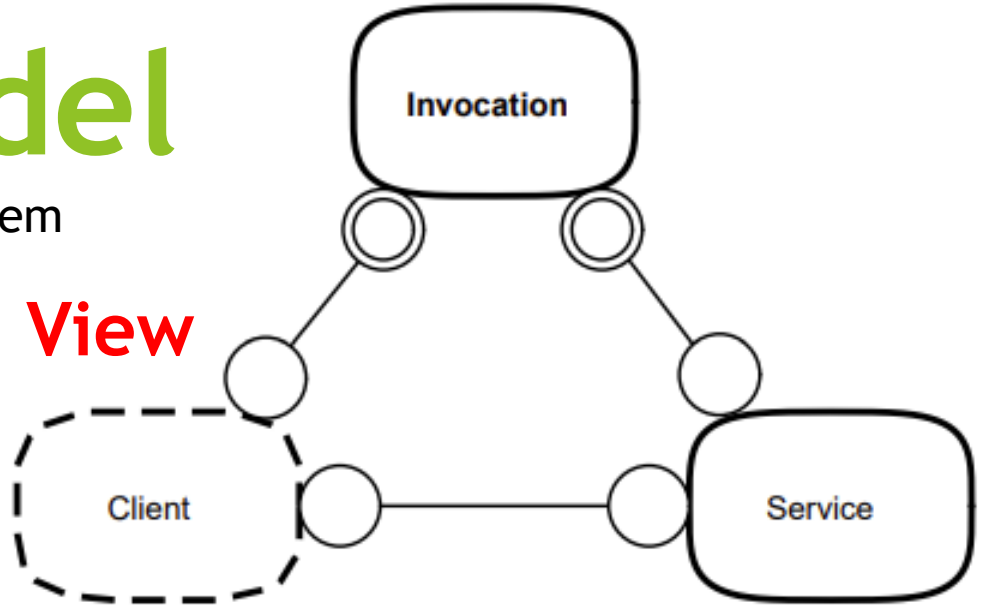


Figure 7. Basic Invocation Framework

2. Scenario View

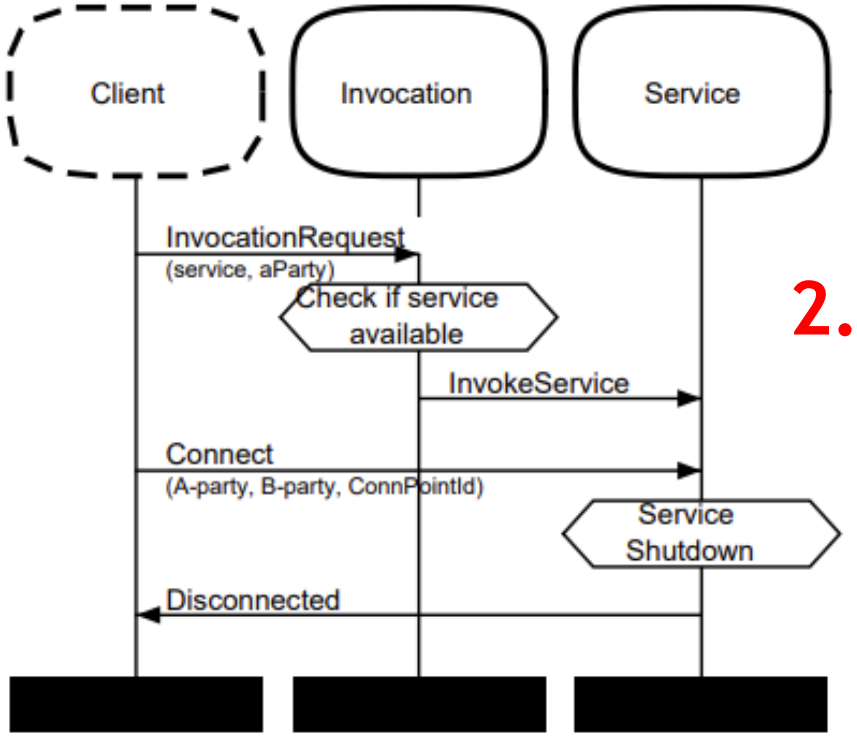


Figure 8. Basic Invocation Framework

3. Method View

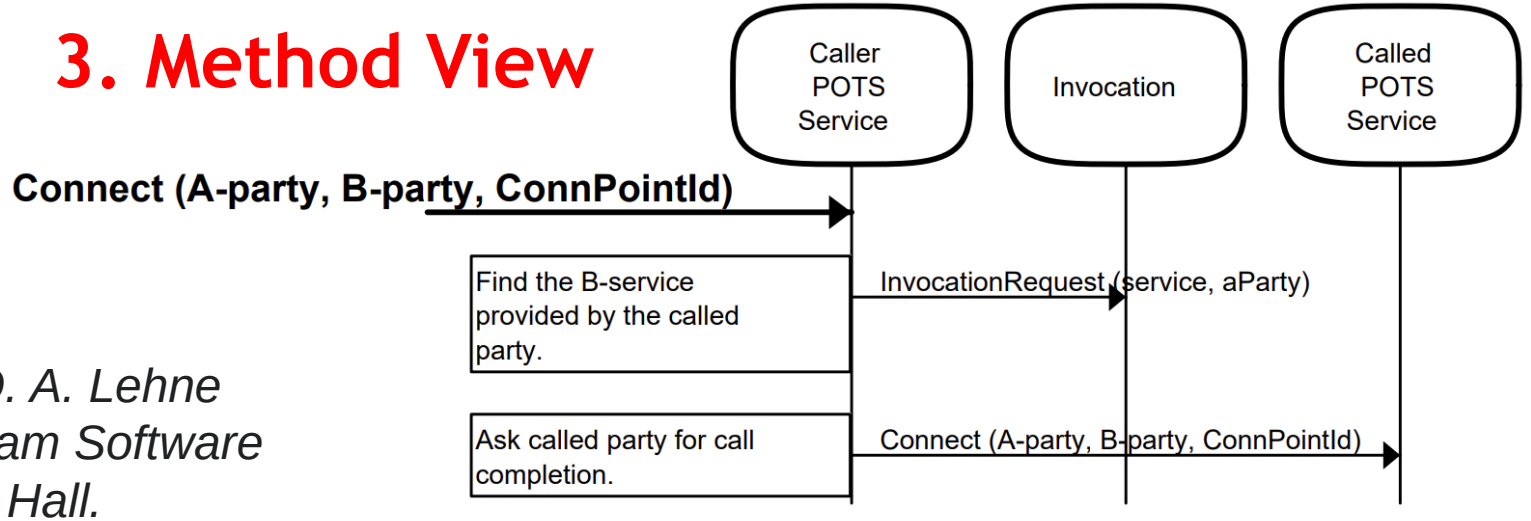


Figure 9. Sample Method: Establish communication with called party.

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Collaboration View

-collaboration structure incorporating system roles

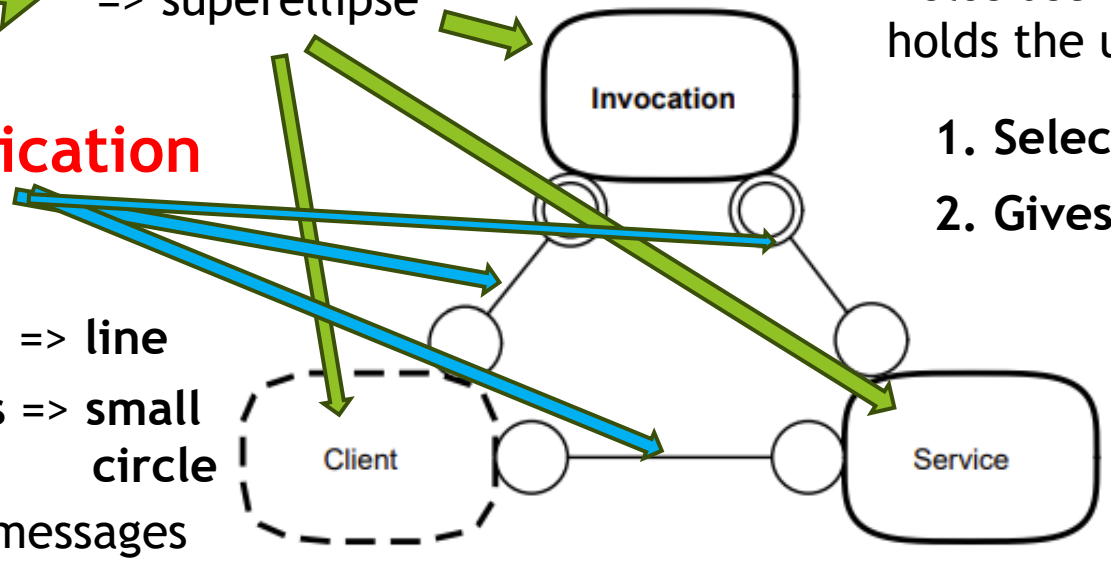
Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Example: Each user has own instance of **Invocation role**

Consists of

1. Roles
=> superellipse

2. Communication structure



Holds user set of available services, holds the user set of services

1. **Selects and Initializes Service object first**
2. **Gives Service reference to the client**

Communication path => **line**

Port to send messages => **small circle**

Role can send messages with its collaborator

Invocation is capable to respond on client request with Service object

Figure 7. Basic Invocation Framework

Many relation => **circle with concentric circle**

Example: invocation can handle many clients

Archetypical role => **dashed line** Example: archetypical client

Port

⇒ Abstraction of variable with possible mapping to program variable

Scenario View

-interaction of messages as trace of system activity

Roles

-horizontally

Message interactions

-horizontal arrows (from sender role to receiver)

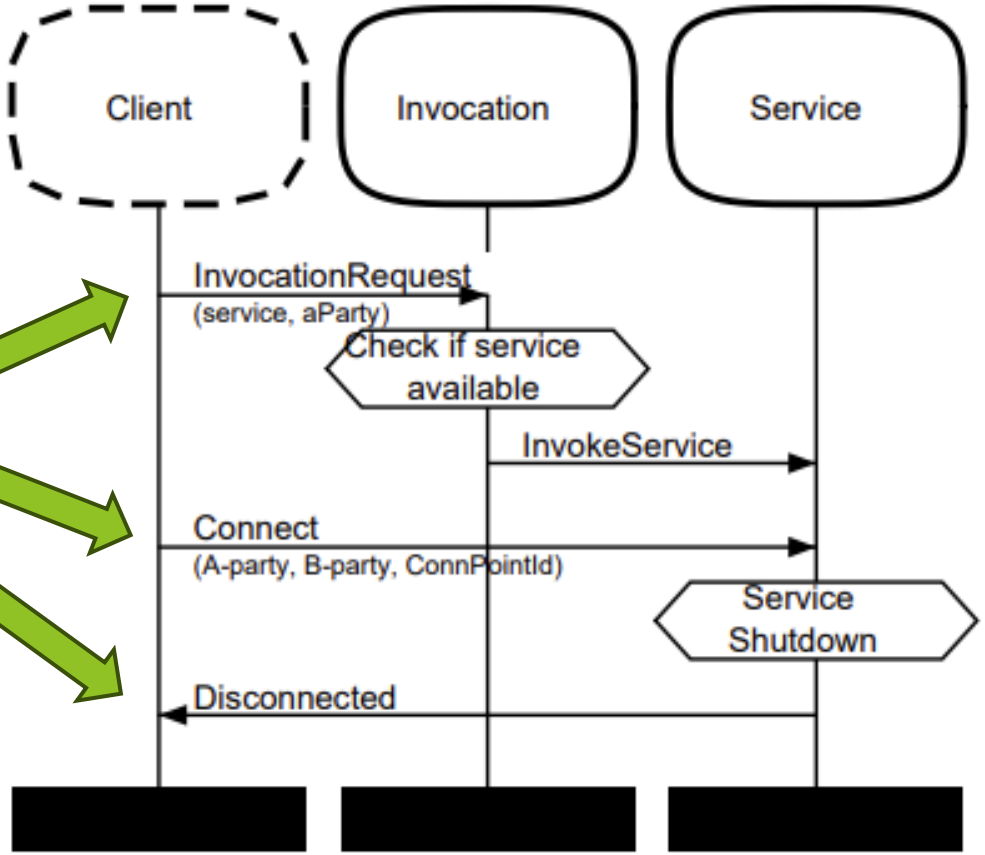


Figure 8. Basic Invocation Framework

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

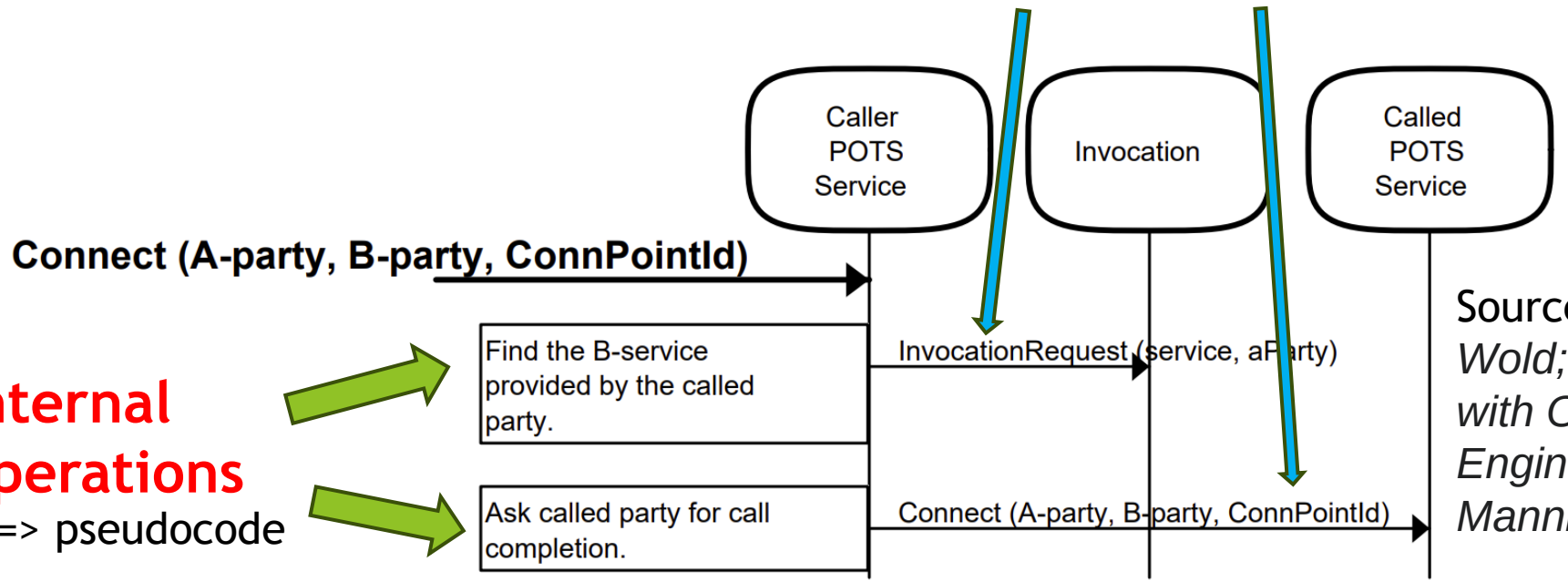
Method View

-to perceive how received message is handled by particular role

Example: establishing connection with POTS Service:

Sent messages

=> Explicitly shown



Internal operations

=> pseudocode

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Figure 9. Sample Method: Establish communication with called party.

Role Model Synthesis

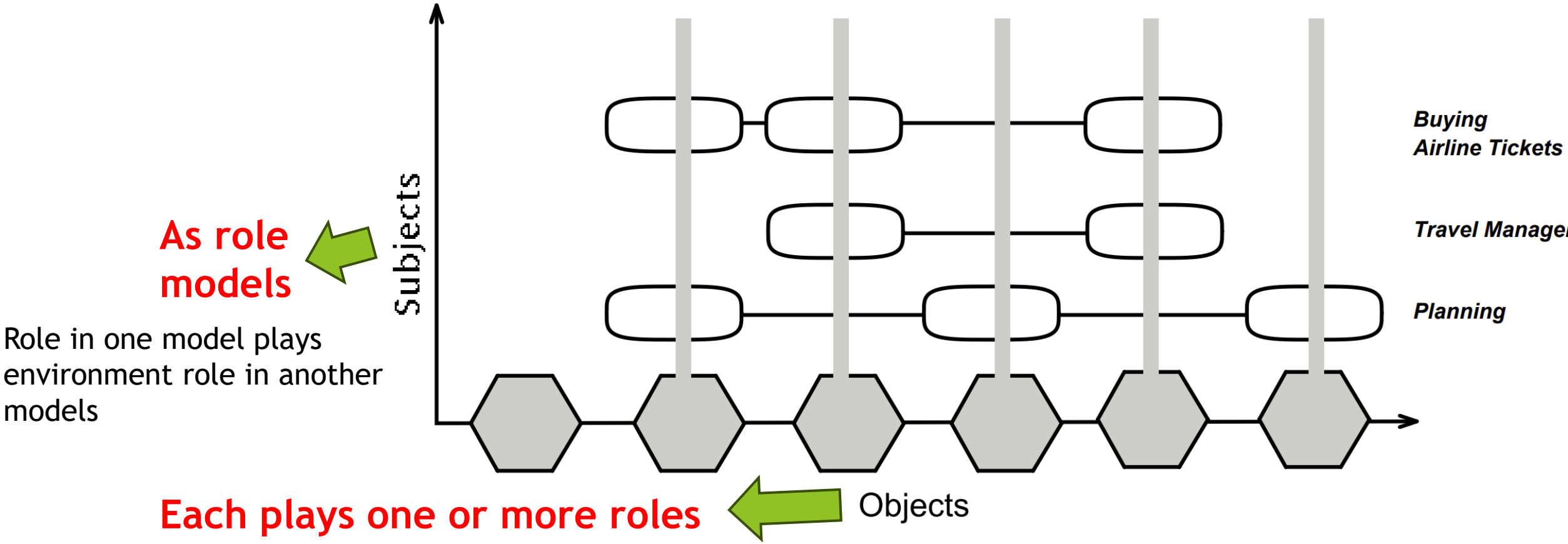


Figure 10. This 'hat stand' synthesis illustration is due to Philip Dellaferra.

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Synthesizing Reusable Components

SINGLE SYNTHESIS OPERATION:

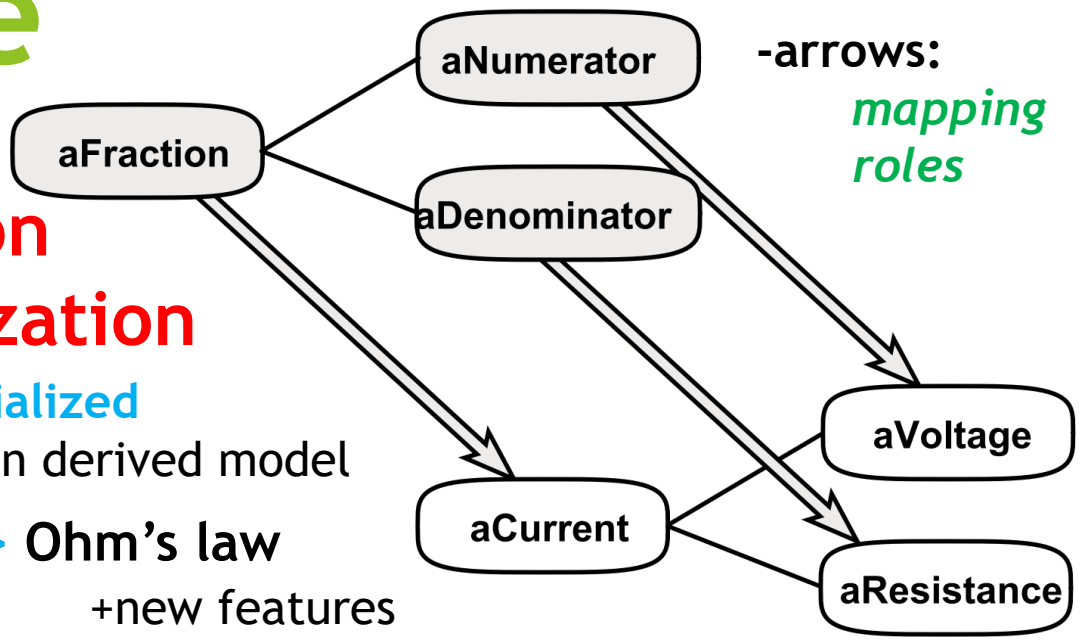


Figure 11. Specializing a general concept with synthesis.

1. Specialization - generalization

General concept is **specialized (inheritance)** in derived model

Example: fraction -> Ohm's law

-all properties (static dynamic) are preserved

+new features

Example: composite fraction

- double synthesis of fraction model

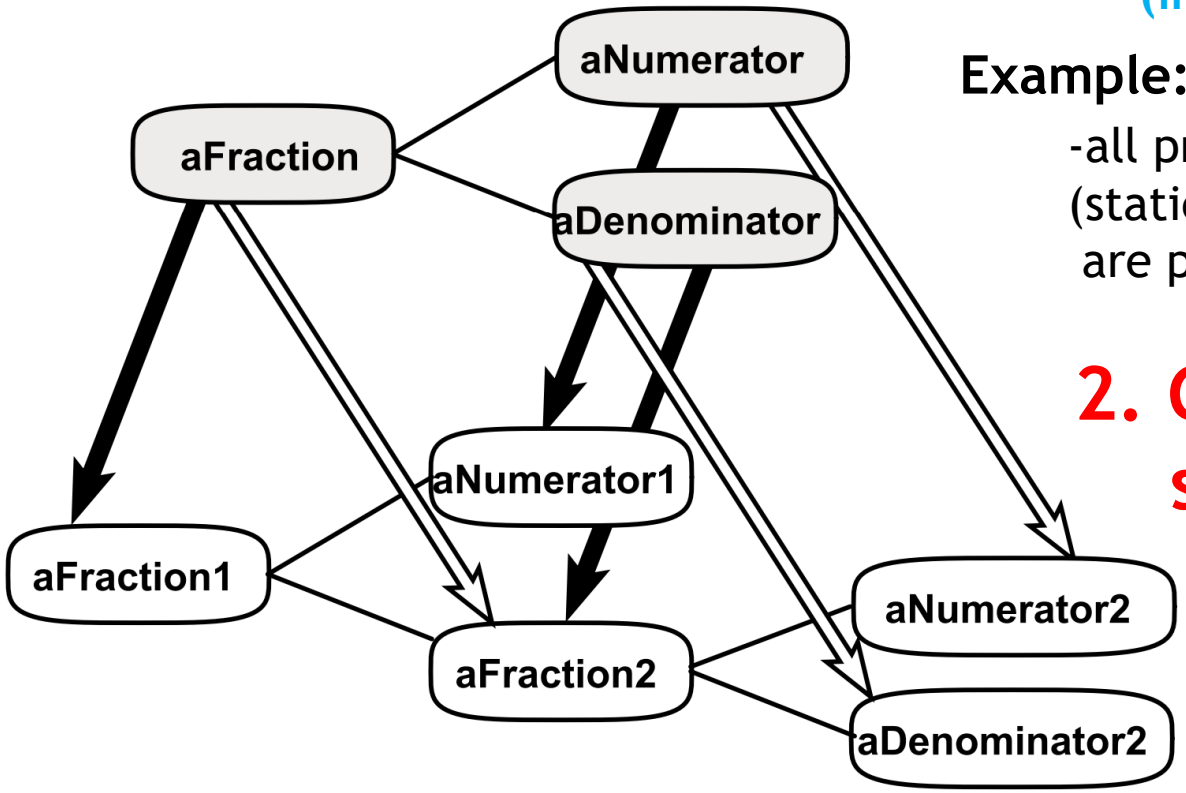


Figure 12. Composition on same level of abstraction.

2. Composition - on the same abstraction level

-composition of related models into composite derived model

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

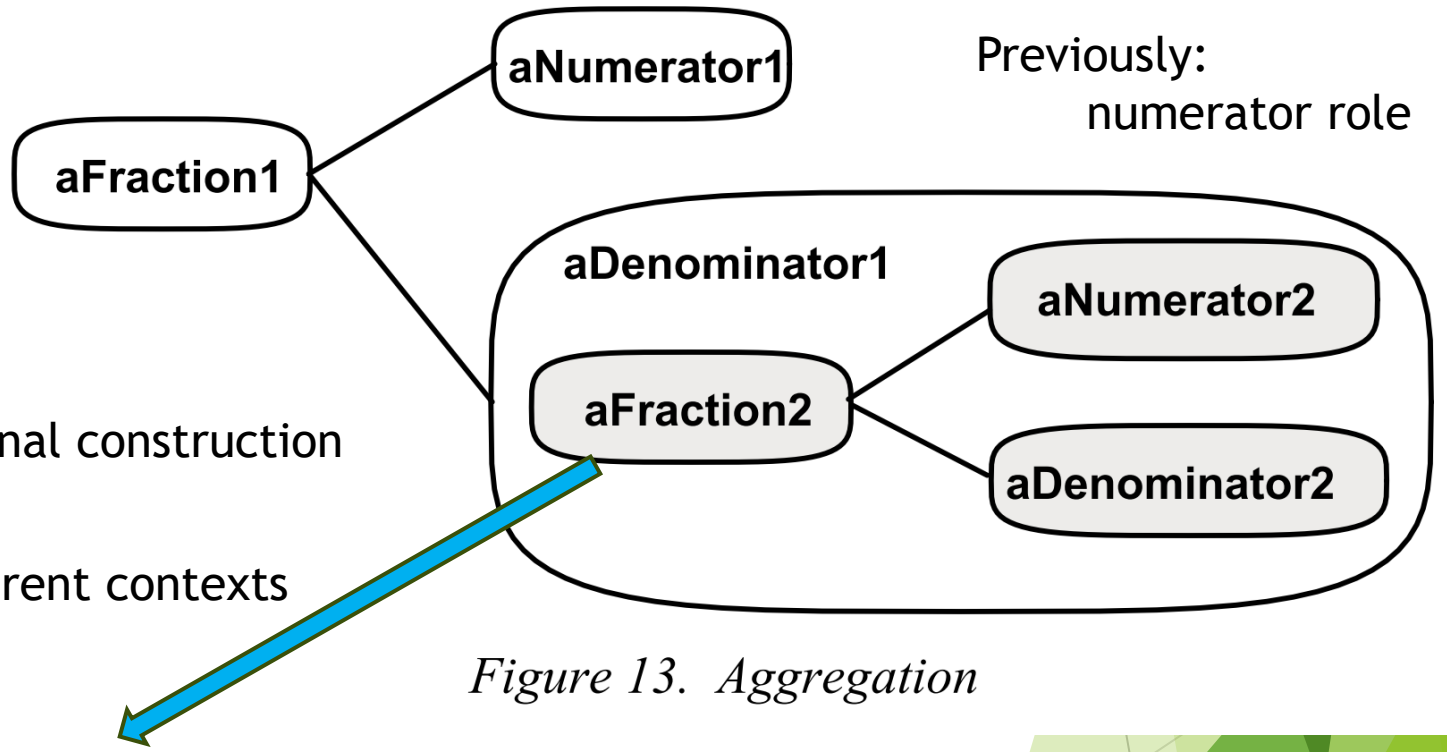
Synthesizing Reusable Components

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

3. Aggregation

-encapsulation – invisibility of internal construction from the outside

- **ENABLES REUSE** -> Application to different contexts



Previously:
numerator role

Figure 13. Aggregation

Replacement

1. **SIMPLE ROLE → ROLE ENCAPSULATING SEPARATE ROLE STRUCTURE**

Example:

numerator role -> role encapsulating another instance of fraction model

The same as composition from previous example, but **ENCAPSULATED**

= INVISIBLE TO OUTHER (aFRACTION) MODEL

Seamless Brigade to Implementation

Constructing **product classes**
by **subclassing the framework classes**

ABSTRACTIONS
related through
common concepts
of objects

Many-to-many
relationships

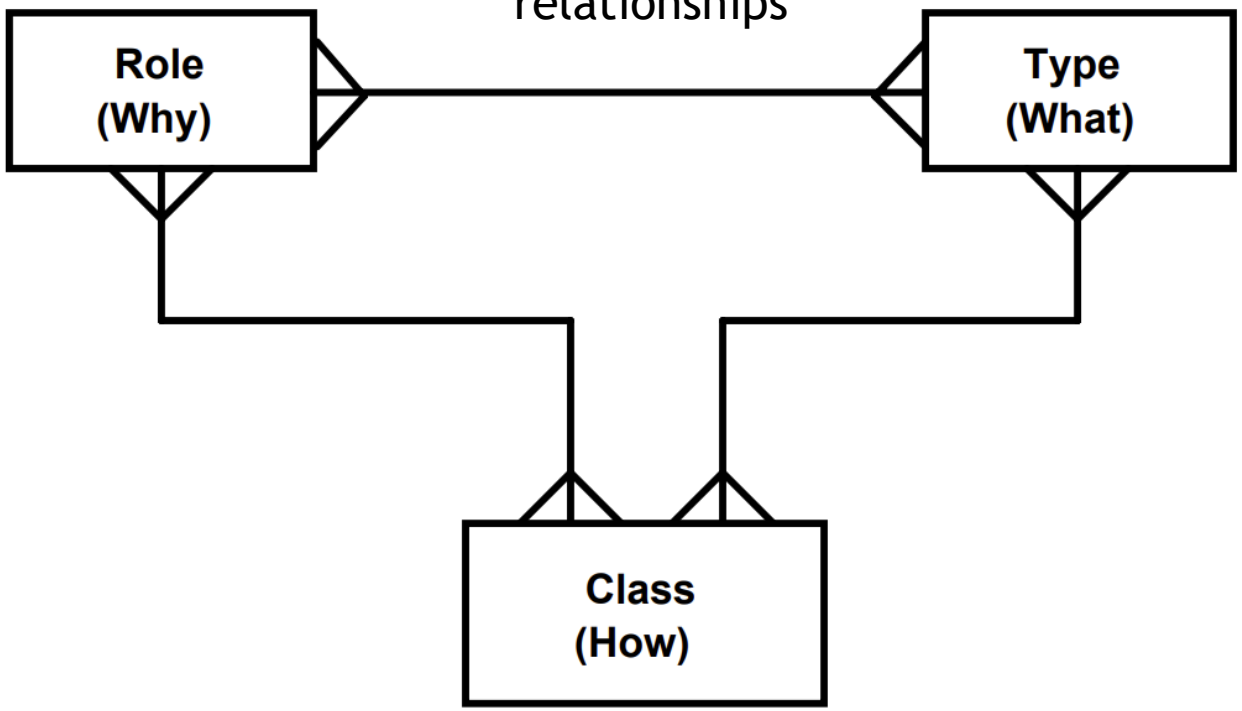


Figure 14. Semantic model of Role - Type - Class

Application: OOram

An OOram Framework is an encapsulated solution to a recurring problem.

It consists of

- ***Role models describing the static and dynamic properties of the solution***
- ***The role models are designed for specialization through synthesis.***
- ***A corresponding ensemble of coordinated classes***
- ***The classes are designed for specialization through subclassing.***

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

Application: OOram

Synthesis

1. **Synthesis 1:** Mapping to corresponding model - SPECIALIZATION

Default behavior of base model is preserved

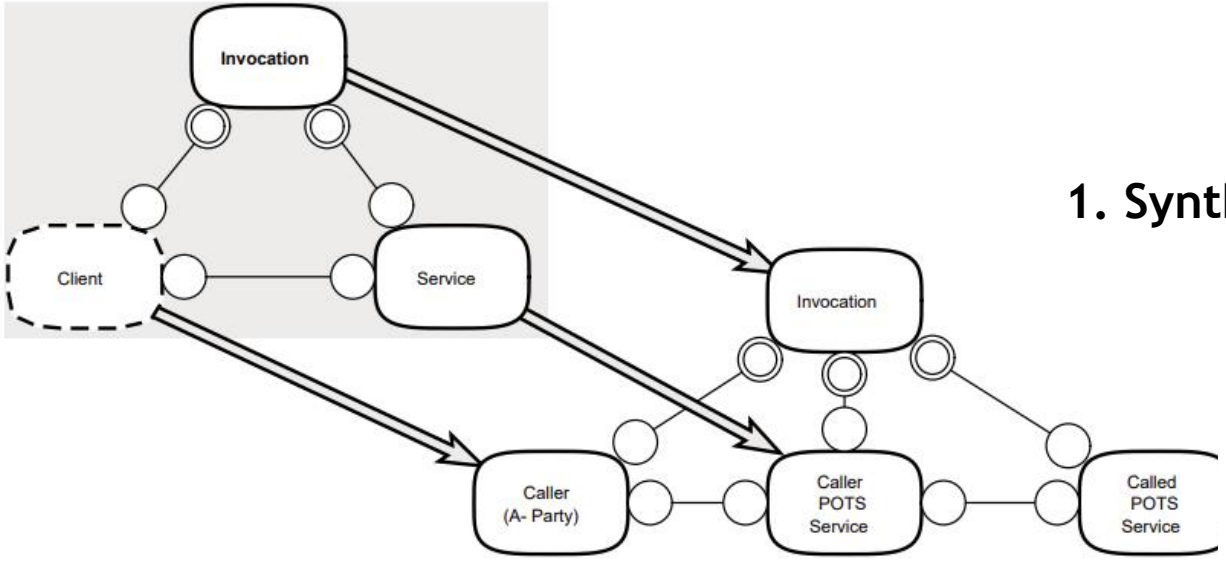


Figure 15. Synthesize POTS from reusable base models

Example: accessing caller POTS by Caller

1. **Synthesis 2:** Mapping to corresponding model - SPECIALIZATION

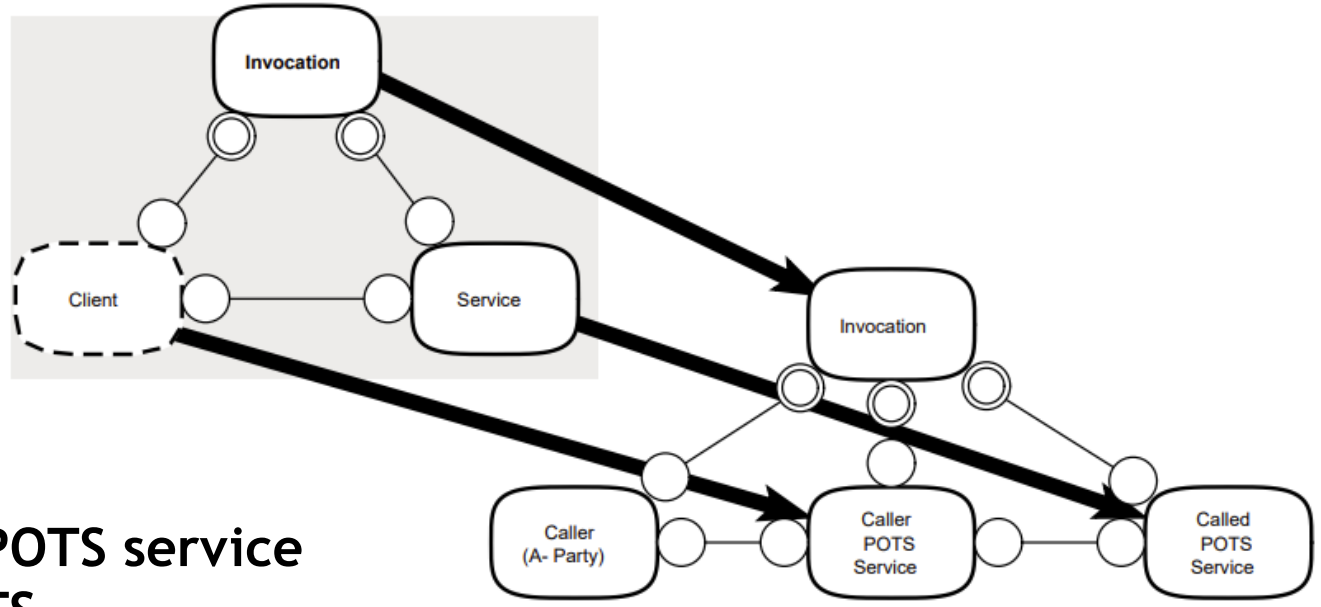


Figure 16. Repeat to synthesize B-side

Example: accessing called POTS service (user service) by caller POTS

Reusing Default Behavior Through Specialization

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

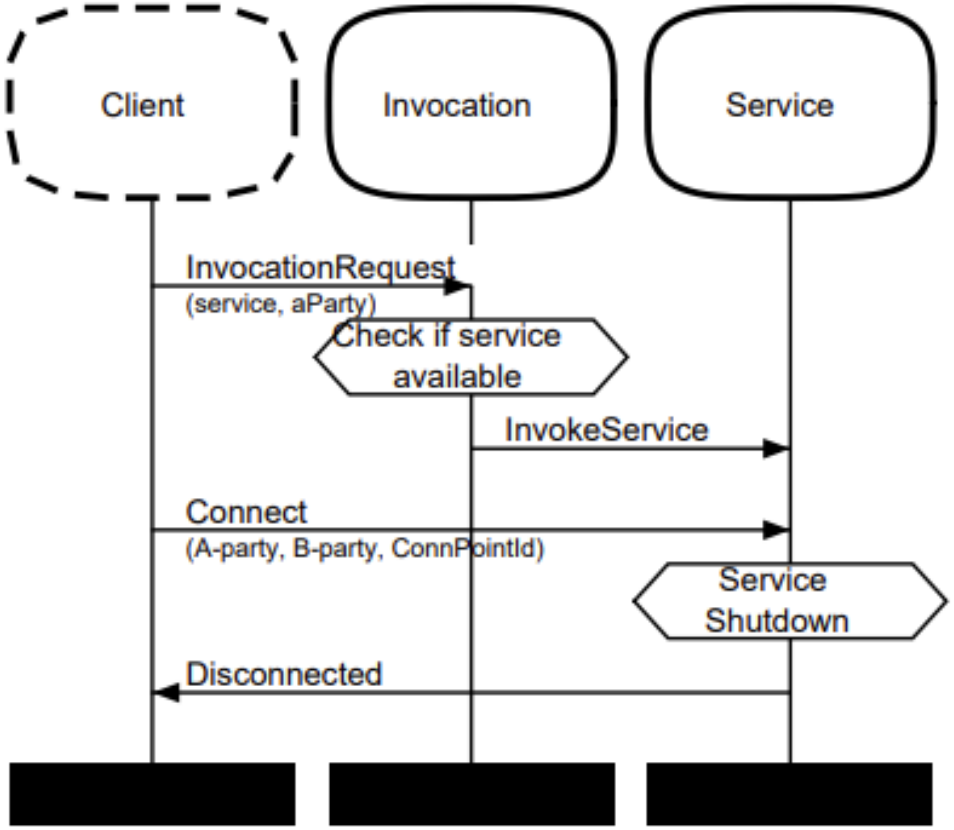


Figure 8. Basic Invocation Framework

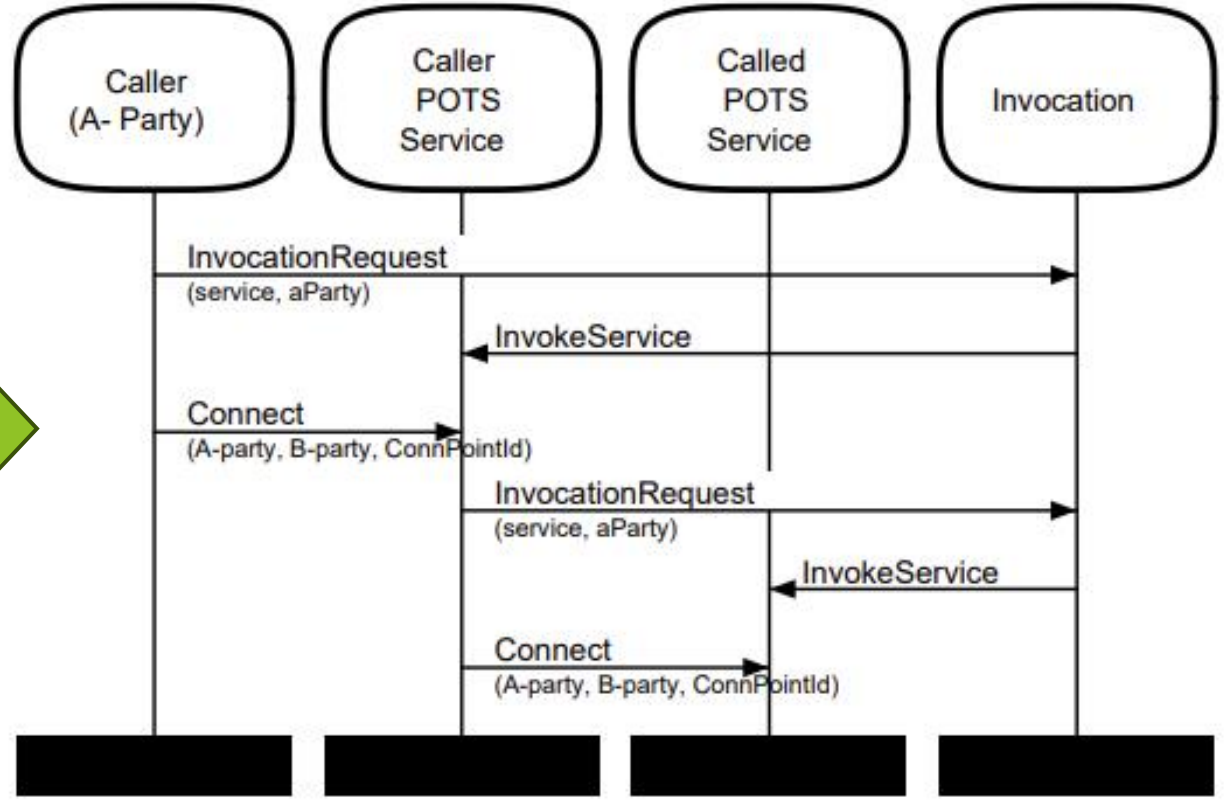


Figure 17. Sample Scenario: Open POTS Telephone Service

Application: OOram

Incorporating Call Forward Service

Source: *Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.*

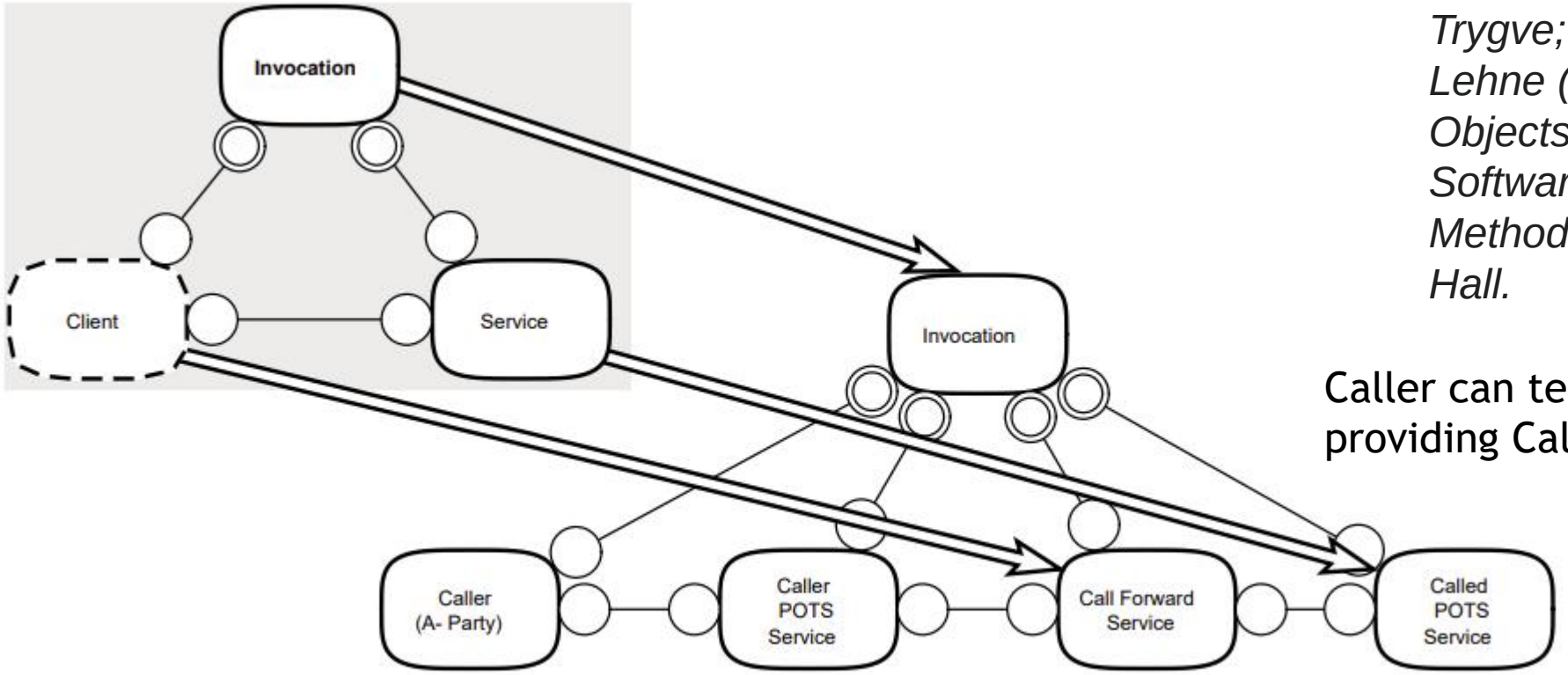
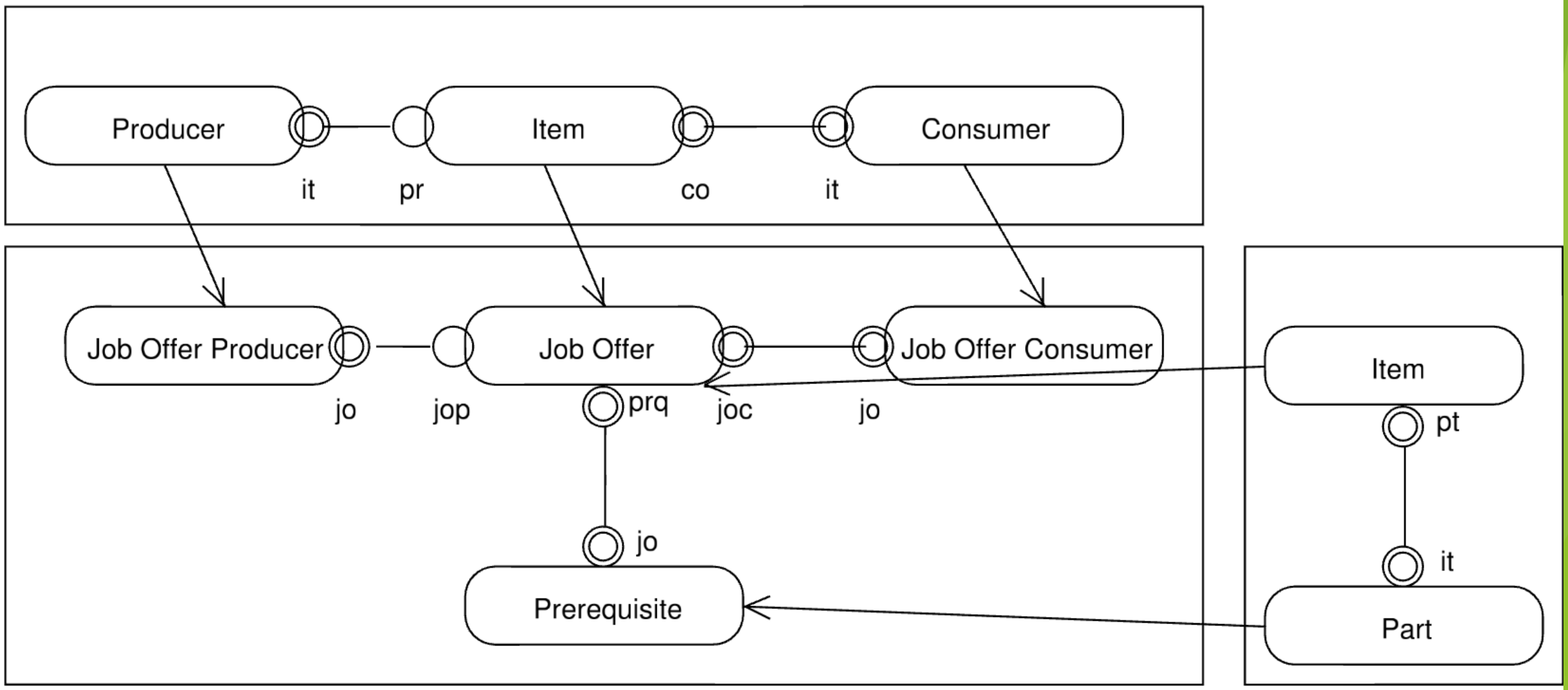


Figure 18. Construct Call Forward by replacing B-Service and repeat synthesis a third time

Caller can terminating search by providing Called POTS service

Caller can implement another forwarding step by another provided Call Forward Service



Elements in themes can be perceived as roles. Roles are known from:

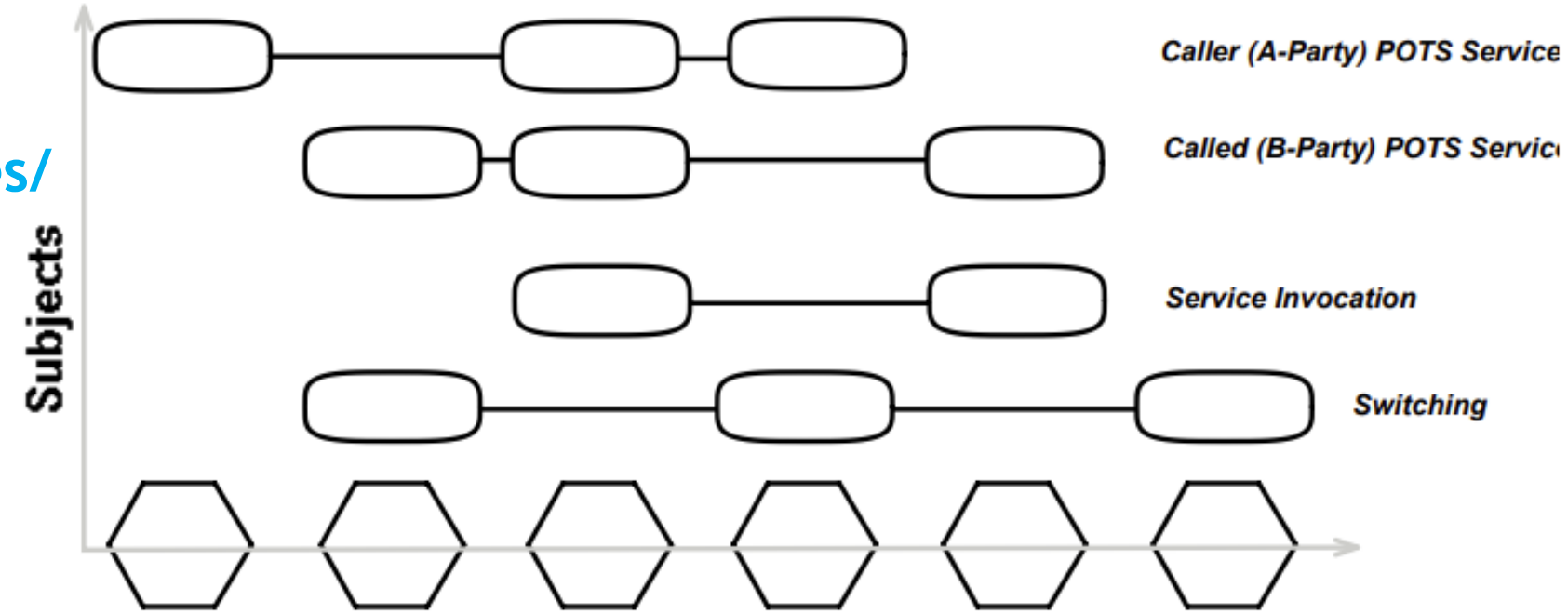
Object-Oriented Role Analysis and Modeling (OOram)

Elements in themes can be perceived as roles.

Perceiving Aspects as Role Collaborations

Concerns/Themes/

As role models



/Elements in themes

Each plays one or more roles

Figure 6. Subjects and objects

Perceiving Aspects as Role Collaborations

TRANSFORMATION FROM THEME/UML TO OORAM

Elements in themes can be perceived as roles.

View as role

Decomposing whole to units covered by specific aspect or view.

Role ***extending object functionality:***

-to be applied in other contexts

-perform corresponding functionality

-take effect into inherent structure

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems,** Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

Extensions to object-oriented programming

Role based approaches to software development

Aspect-oriented programming

Both have capability to be added, removed and replaced at runtime

Theme/UML And OOram Relations

Table 1. The corresponding notions in Theme/UML and OOram.

Theme/UML	OOram
theme	collaboration of roles
parameter class in an aspect theme	role
non-parameter class in an aspect theme	role
class	role or collaboration of roles
class fragment	role
operation	interface method
bind	two roles relationship
base theme	collaboration view diagram
aspect theme	collaboration view diagram
concept sharing	role sharing in the collaboration diagram
crosscutting	relationship between two roles
decomposition: theme creation	decomposition: role model creation
composition: composing themes	synthesis: composing role diagrams
structural diagram (class diagram)	collaboration/interface view diagram
behavioral diagram (sequence diagram)	scenario view diagram

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems**, Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

Theme/UML

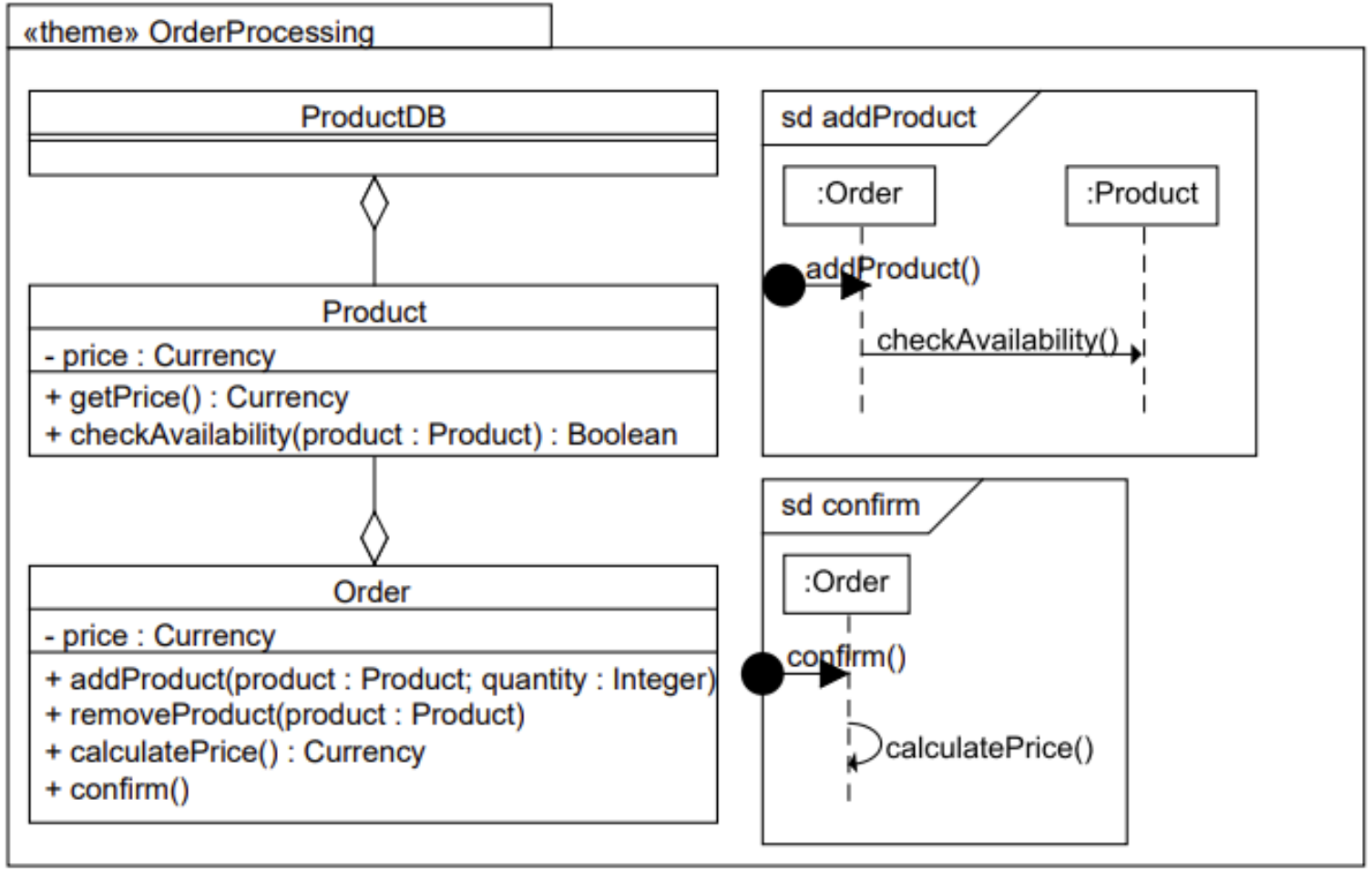


Fig. 1. A base theme.

Theme/UML

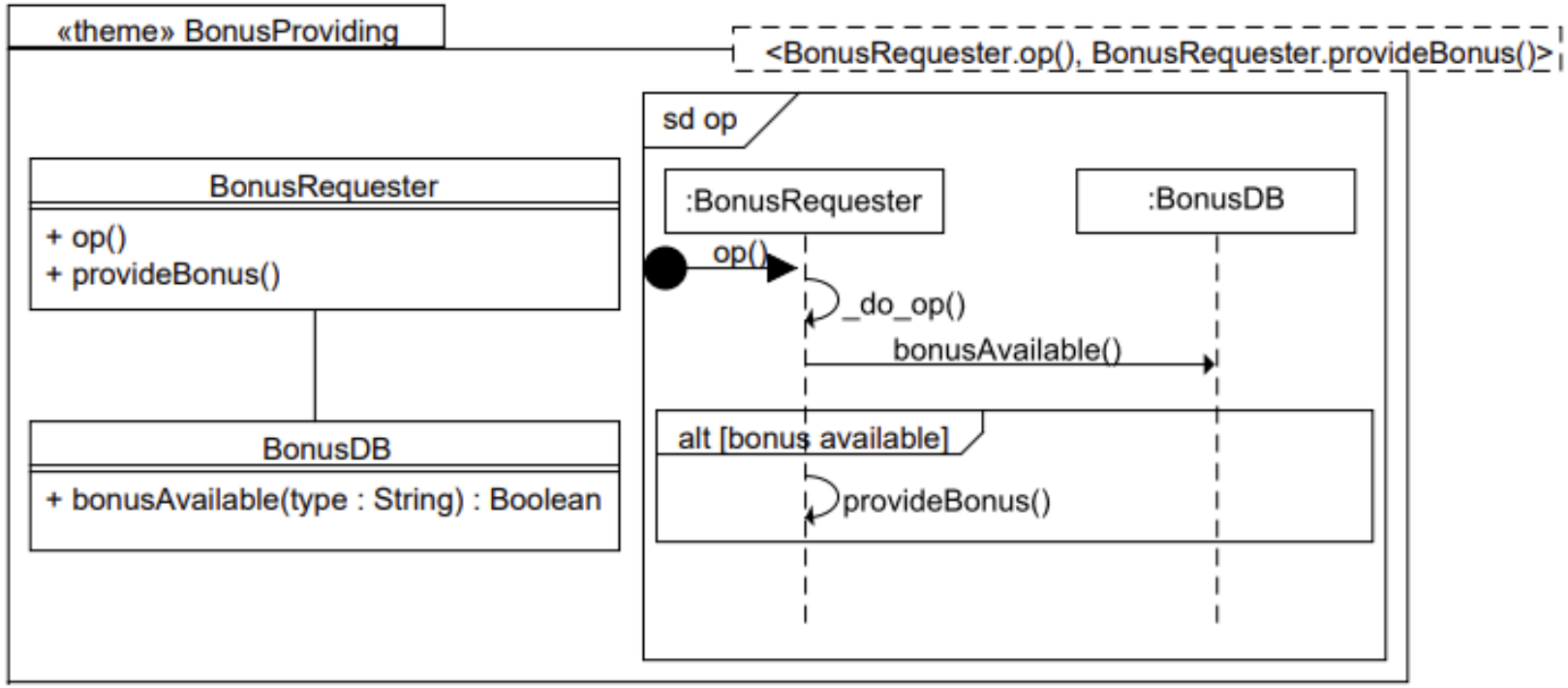


Fig. 2. An aspect theme.

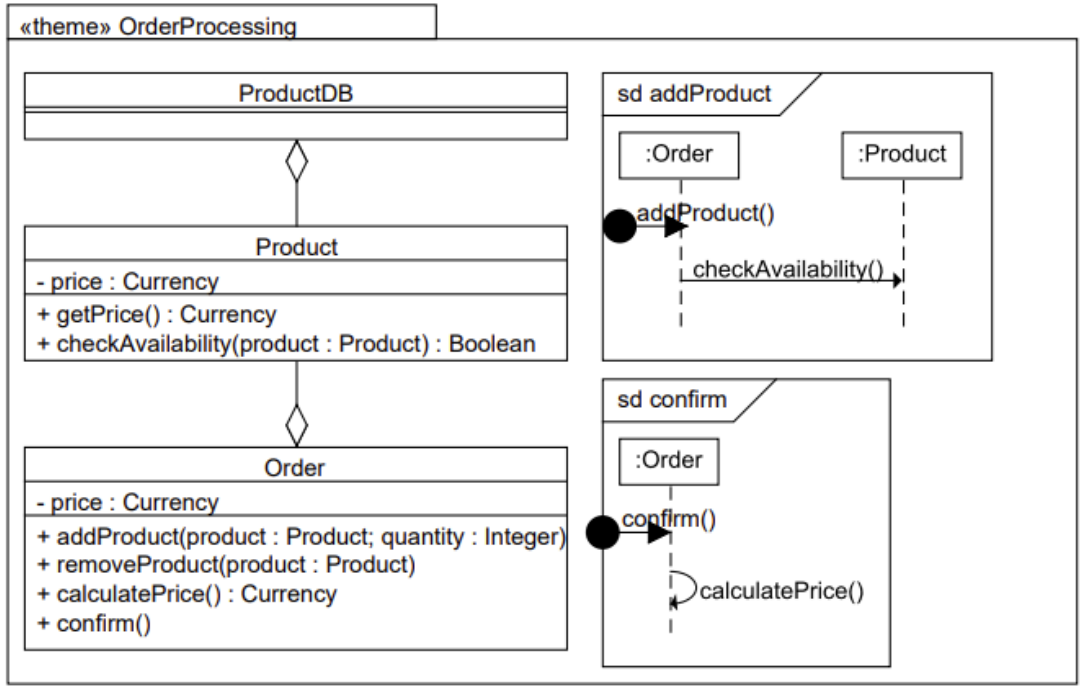


Fig. 1. A base theme.

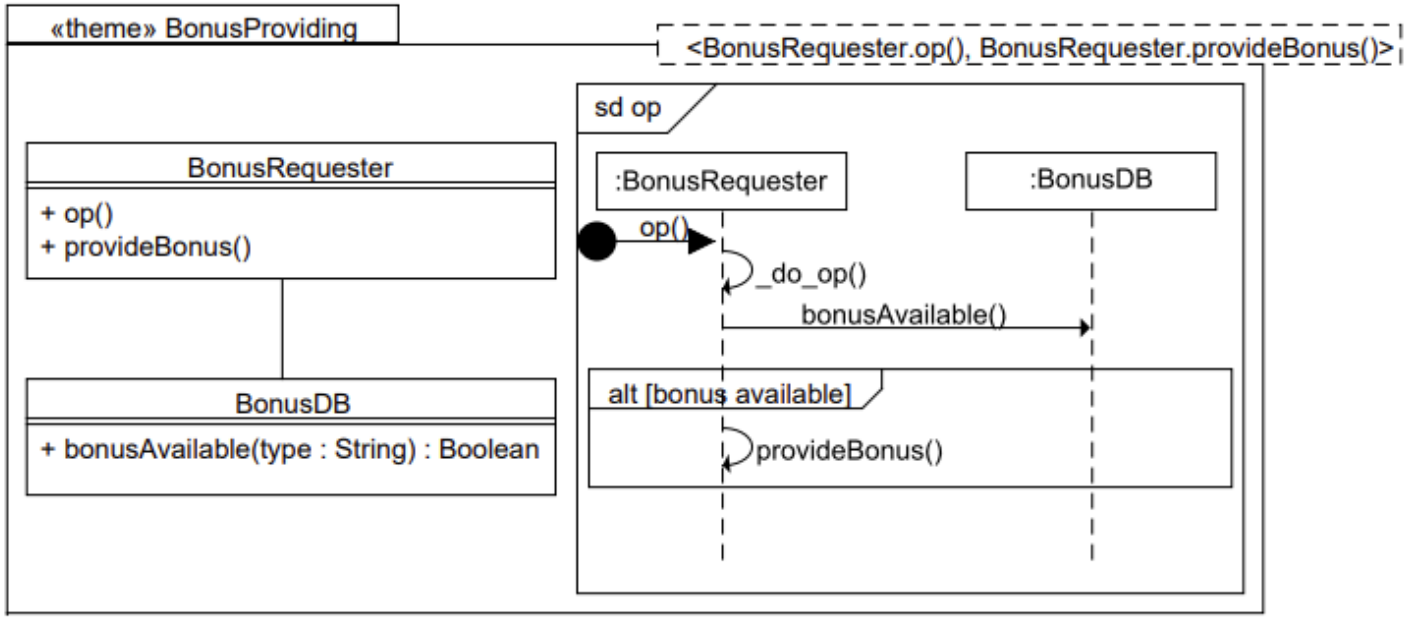


Fig. 2. An aspect theme.

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems**, Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

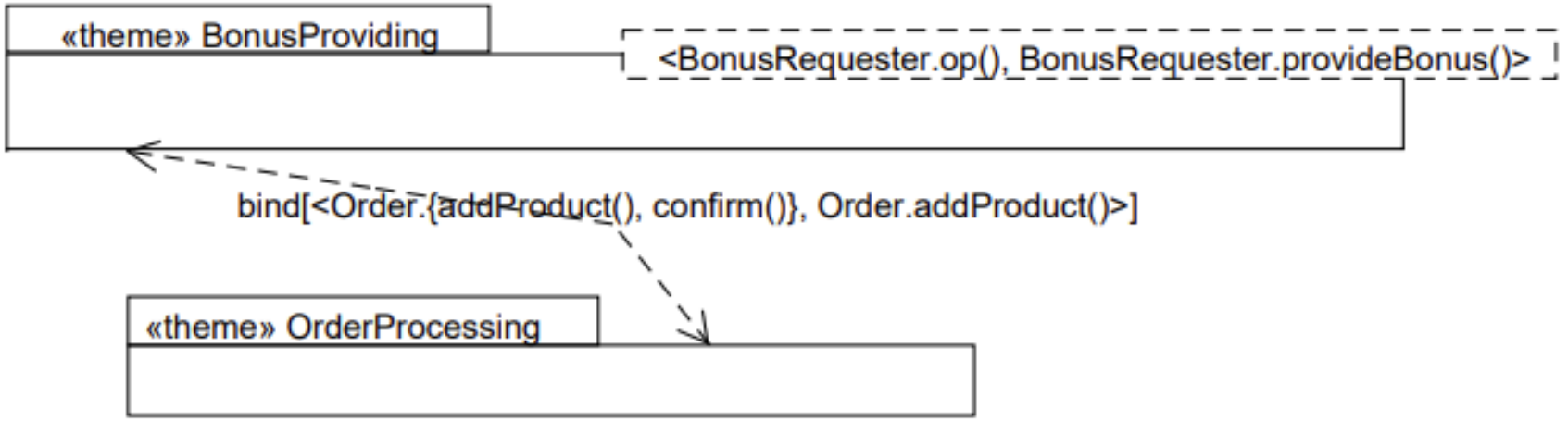


Fig. 3. A composition of an aspect theme with a base theme.

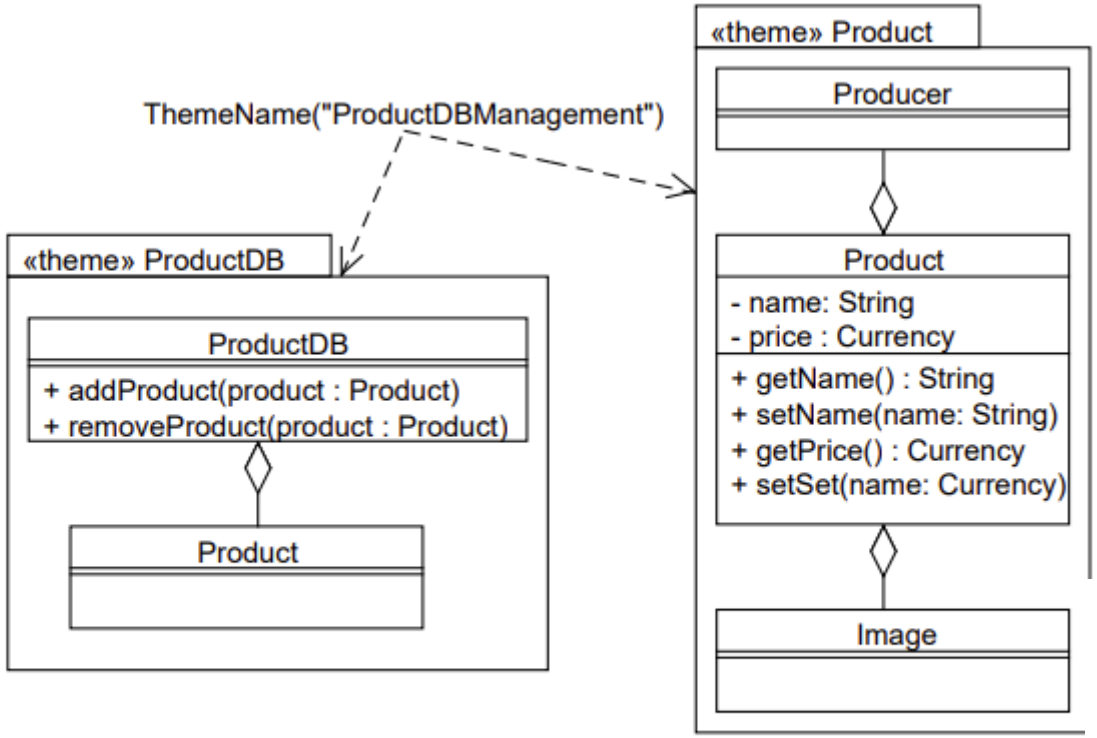


Fig. 5. A composition of two base themes.

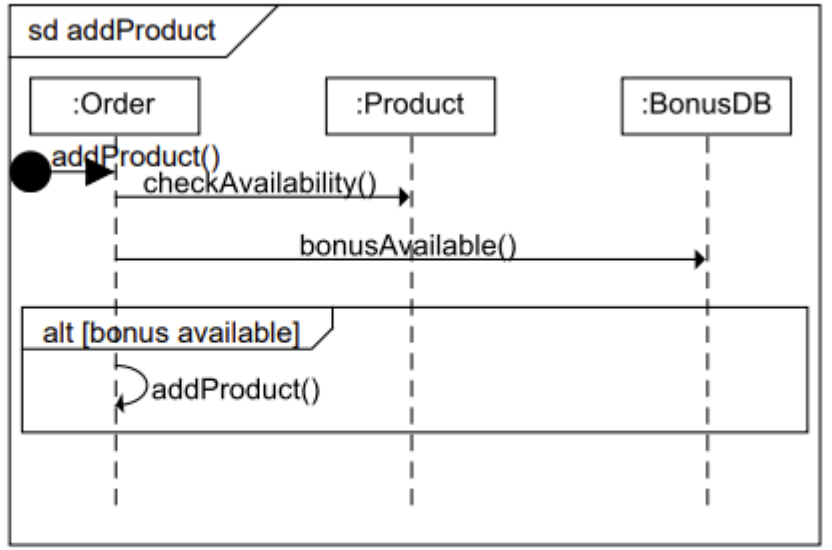


Fig. 4. The *addProduct()* operation as affected by *BonusProviding*.

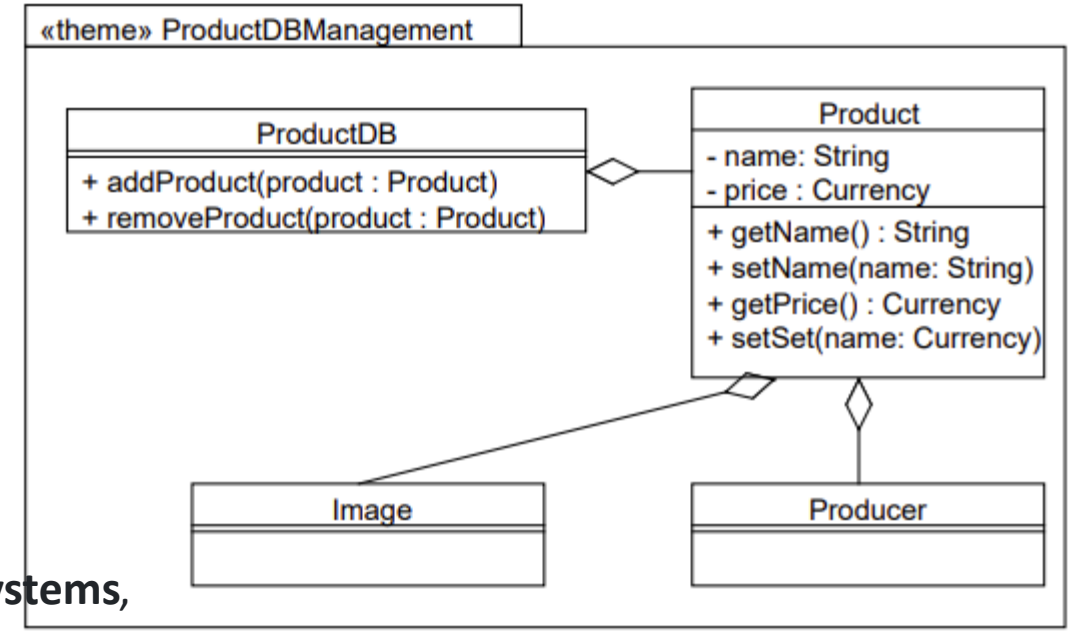


Fig. 6. A composed theme.

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems**, Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

OOram

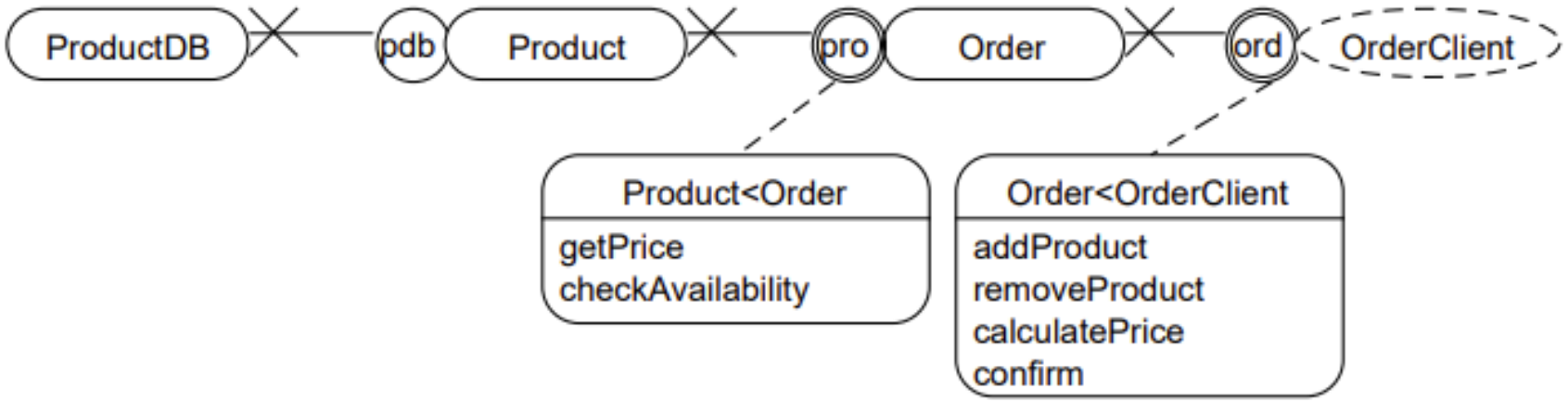


Fig. 7. An OOram interface view diagram.

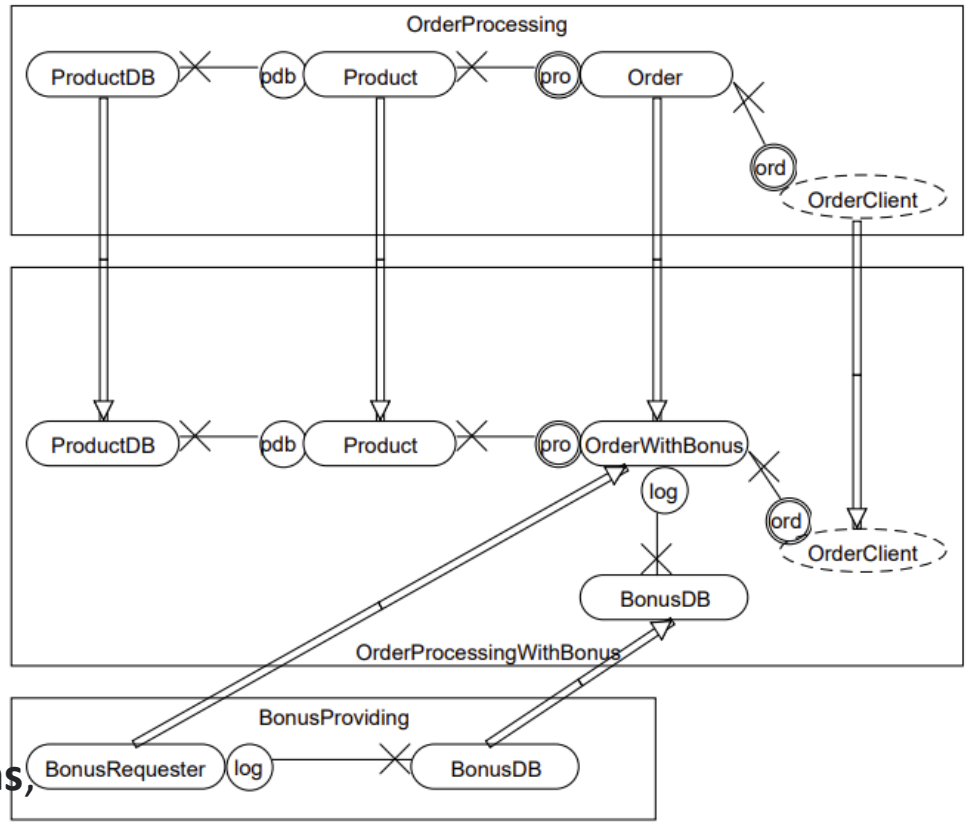


Fig. 8. Synthesis of role models.

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems,** Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

OOram

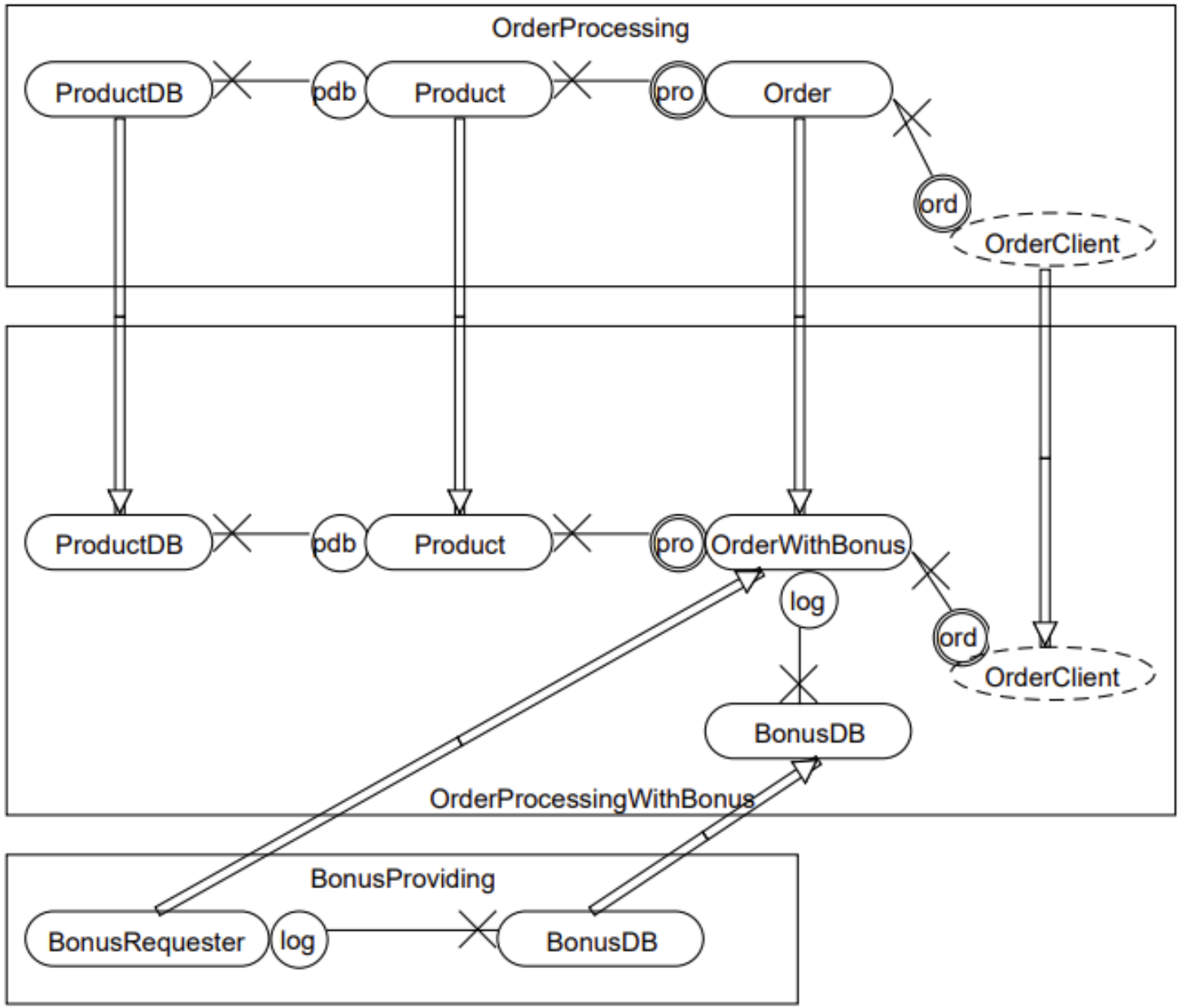


Fig. 8. Synthesis of role models.

Source: **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems**,
Vol. 13, No. 1, 199–216. (2016),
https://doi.org/10.2298/CSIS151207065V

Extension points are introduced explicitly.

Does this break the obliviousness of aspects on the side of the affected code?

Source: http://www2.fiit.stuba.sk/~vranic/

Extension points are introduced explicitly.

Does this break the obliviousness of aspects on the side of the affected code?

Source: http://www2.fiit.stuba.sk/~vranic/

Are preserved use cases in code enough?
Is traceability enough?

References

- **OOram:** Reenskaug, Trygve; P. Wold; O. A. Lehne (1996). Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall.
- **Use Cases:** Cockburn, A.. *Writing Effective Use Cases*.Addison-Wesley, 2001
- **Abstractions analogies in patterns:** J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161-173.
- **Application of Themes approach:** Bc. Pavol Michalco: PRÍPADY POUŽITIA A TÉMY V PRÍSTUPE THEME/DOC
- **D. Stein.** **Join Point Designation Diagrams**: A Visual Design Notation for Join Point Selections in Aspect-Oriented Software Development. PhD. thesis, Universität Duisburg-Essen, 2010.
- **E. Baniassad and S. Clarke,** **"Theme: an approach for aspect-oriented analysis and design,"** Proceedings. 26th International Conference on Software Engineering, Edinburgh, UK, 2004, pp. 158-167, doi: 10.1109/ICSE.2004.1317438.
- **Vranić, V., Laslop, M.: Aspects and Roles in Software Modeling: A Composition Based Comparison. Computer Science and Information Systems**, Vol. 13, No. 1, 199–216. (2016), https://doi.org/10.2298/CSIS151207065V

References

- Bystrický, M., Vranic, V.: Preserving use case flows in source code. In: Proceedings of 4th East- ´ ern European Regional Conference on the Engineering of Computer Based Systems, ECBSEERC 2015. IEEE, Brno, Czech Republic (2015)
- Hanenberg, S., Stein, D., Unland, R.: Roles from an aspect-oriented perspective. In: Proceedings of VAR'05: Views, Aspects and Roles Workshop, ECOOP 2005. Glasgow, UK (2005)
- Hanenberg, S., Unland, R.: Roles and aspects: Similarities, differencies, and synergetic potential. In: Proceedings of 8th International Conference on Object-Oriented Information Systems, OOIS 2002. Montpellier, France (Sep 2002)
- Reenskaug, T., Coplien, J.O.: The DCI architecture: A new vision of object-oriented programming. http://www.artima.com/articles/dci_vision.html (3 2009)

Aspects And Roles

a) Unification of aspects and roles

b) Modeling aspects with roles

c) Role systems as special kind of aspect-oriented systems

d) Similarities between role based and aspect-oriented system composition/decomposition

Theme/UML and OOram -> COMPOSITION PATTERNS

Symmetries

AspectJ - asymmetric

a) Element symmetry

b) Relationship asymmetry