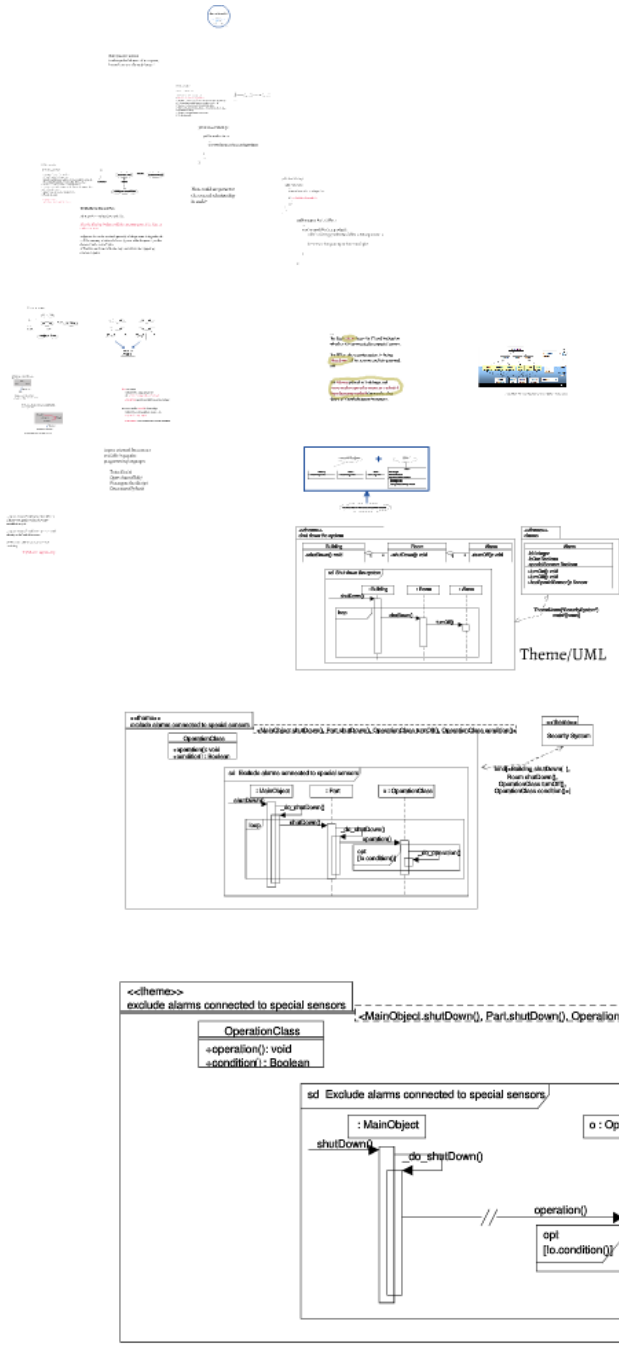



```
pointcut operations(OperationClass o):
    target(o) && call(* turnOff()) && cflow(call(* Building.shutDown(..)));
```

```
public aspect ExcludeAlarmsConnectedToSpecialSensors extends
    ConditionallyStopOperations {

    declare parents: Alarm implements OperationClass;

    public boolean Alarm.condition() {
        return hasSpecialReason();
    }

    pointcut operations(OperationClass o): target(o) && call(* turnOff(..));
}
```

Aspects Around Us

Valentino Vranić

Institute of Informatics, Information Systems,
and Software Engineering

vranic@stuba.sk fiit.sk/~vranic

25/10/2018

Have you ever needed
to change the behavior of a program,
but without actually modifying it?

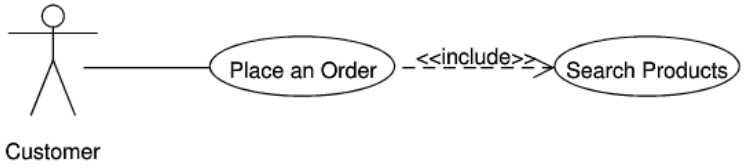
UC Place an Order

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.

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.

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

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

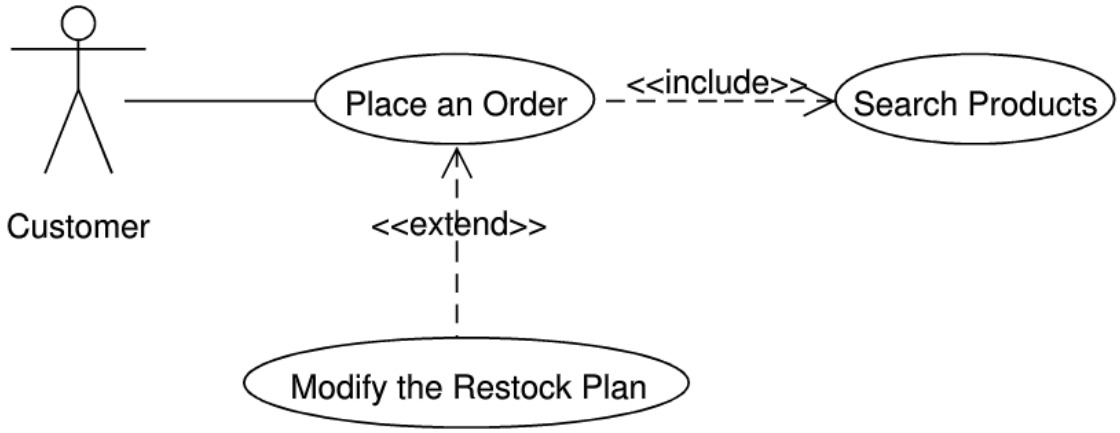
UC Place an Order

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.

How could we preserve the extend relationship in code?

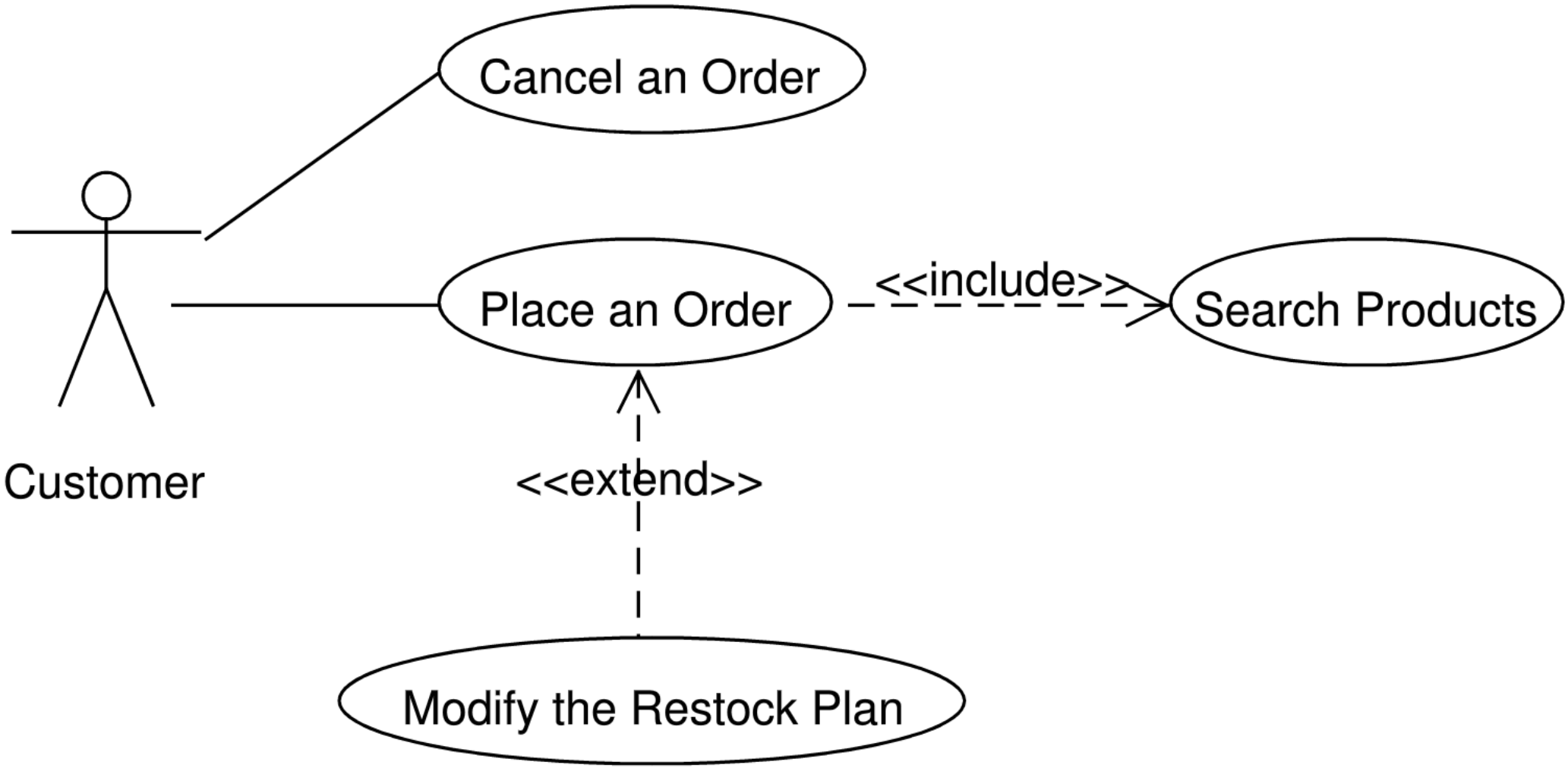
```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
        if (productAvailable(product)) {
            ...
        } else...
    }
    ...
}
```

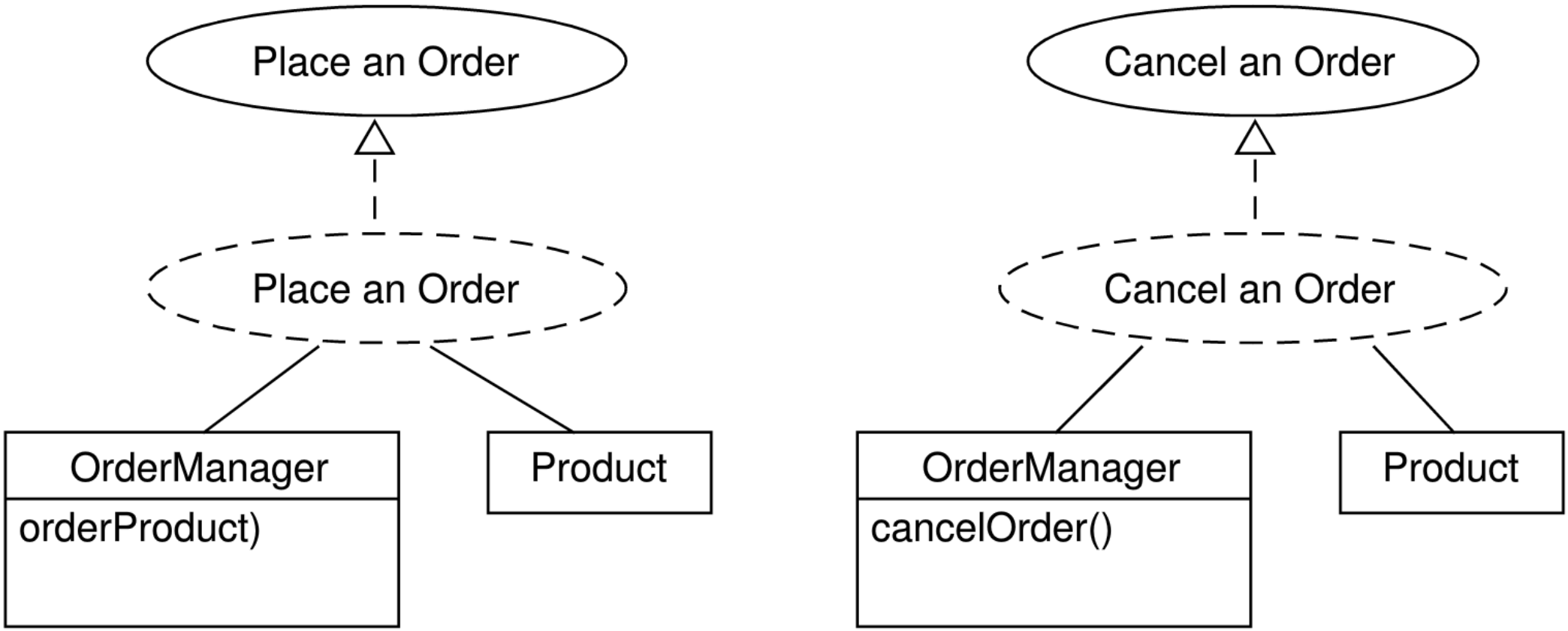
```
public class Ordering {
    ...
    public void order() {
        ...
        new ProductSearch().search(product);
        ...
        if (productAvailable(product)) {
            ...
        } else...
    }
    ...
}
```

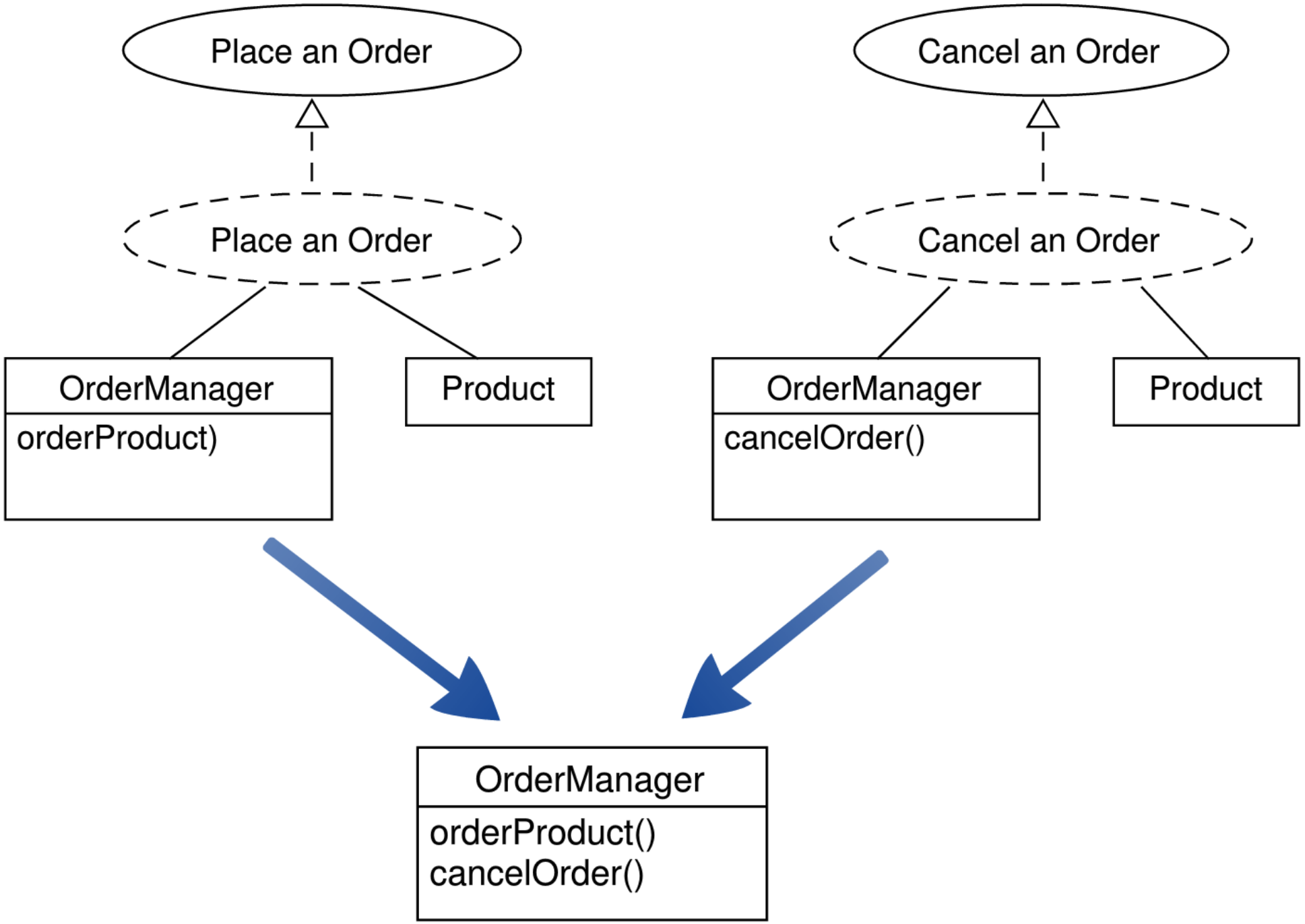
```
public aspect RestockPlan {
    ...
    void around(Product product):
        call(* Ordering.productAvailable(..) && args(tovar) {

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

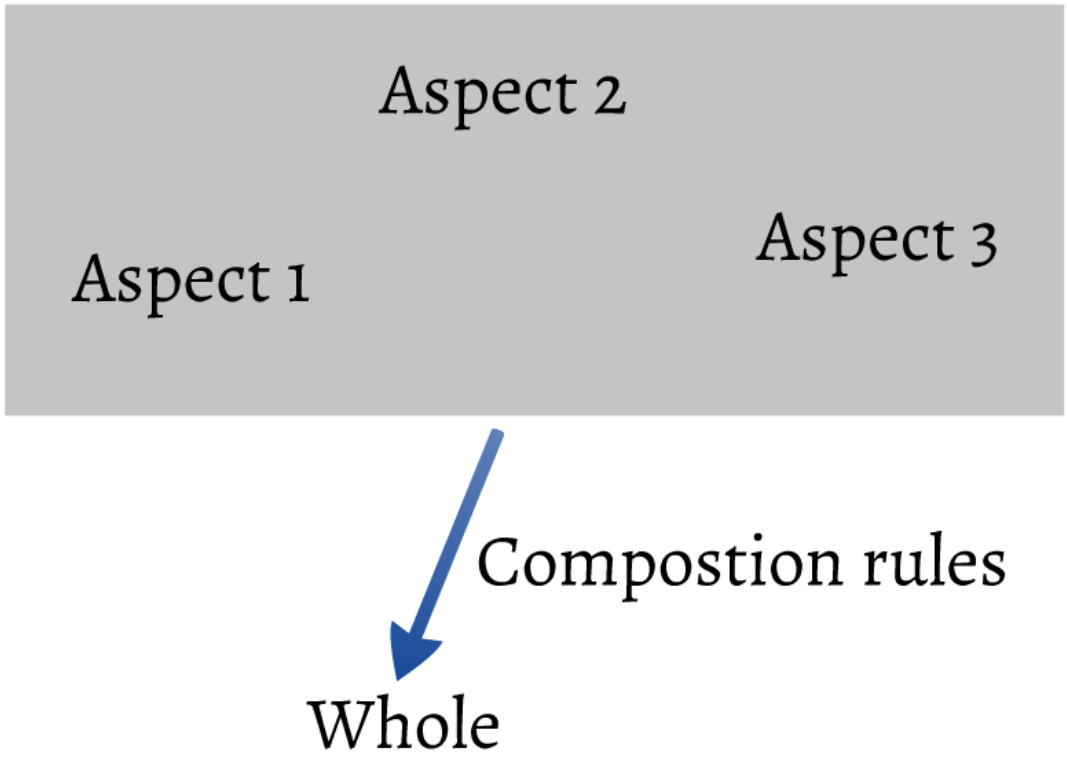
Peer use cases




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


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

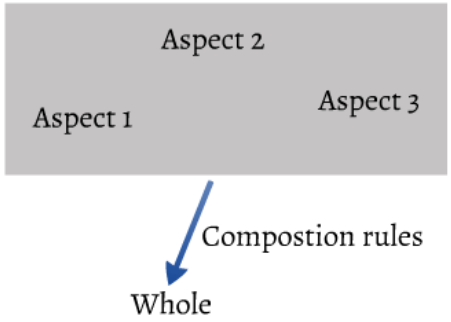
Symmetric aspect-oriented modularization (decomposition)



> Aspects as different views of a whole

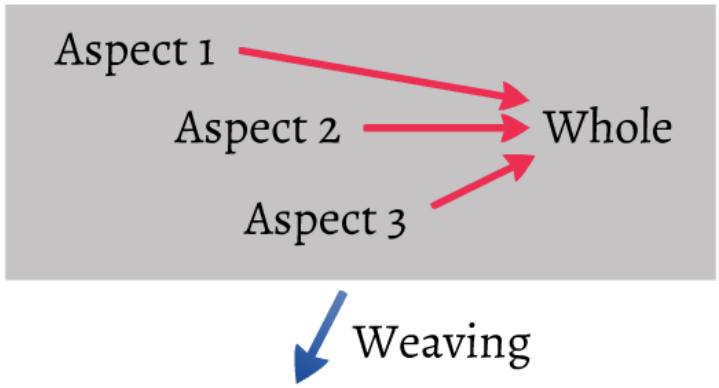
> The whole is being composed out of the elements being at the same level

Symmetric aspect-oriented modularization (decomposition)



> Aspects as different views of a whole
> The whole is being composed out of the elements being at the same level

Asymetric aspect-oriented modularization (decomposition)



> Aspects affect a preexisting whole

Peer use cases:

realized by a composition of
the entities being at the same level

SYMMETRIC ASPECT-ORIENTED MODULARIZATION

Use cases in the extend relationship:

realized by affecting basic entities by
a special entity (aspect)

ASYMMETRIC ASPECT-ORIENTED MODULARIZATION

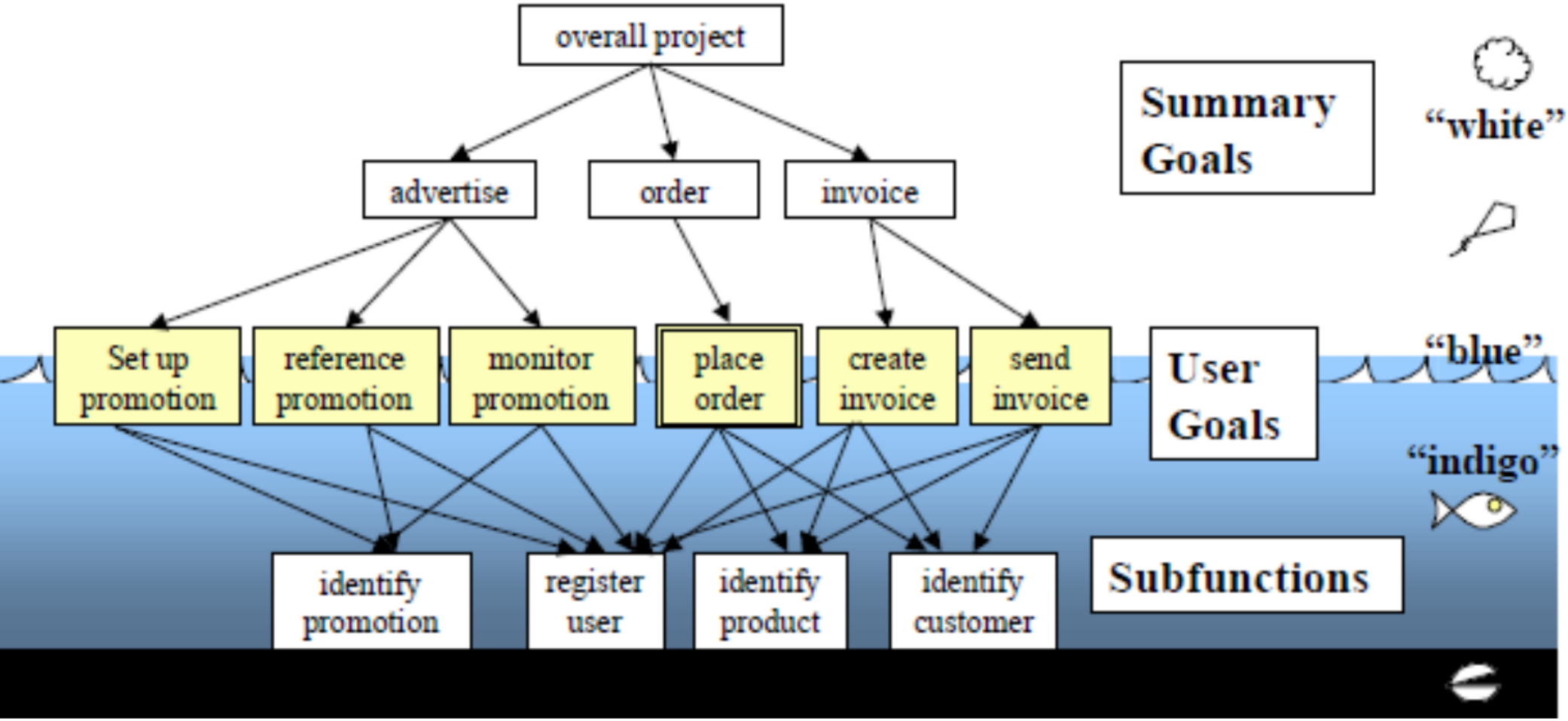
...

R3: Each alarm bears its ID and indication whether it is connected to a special sensor.

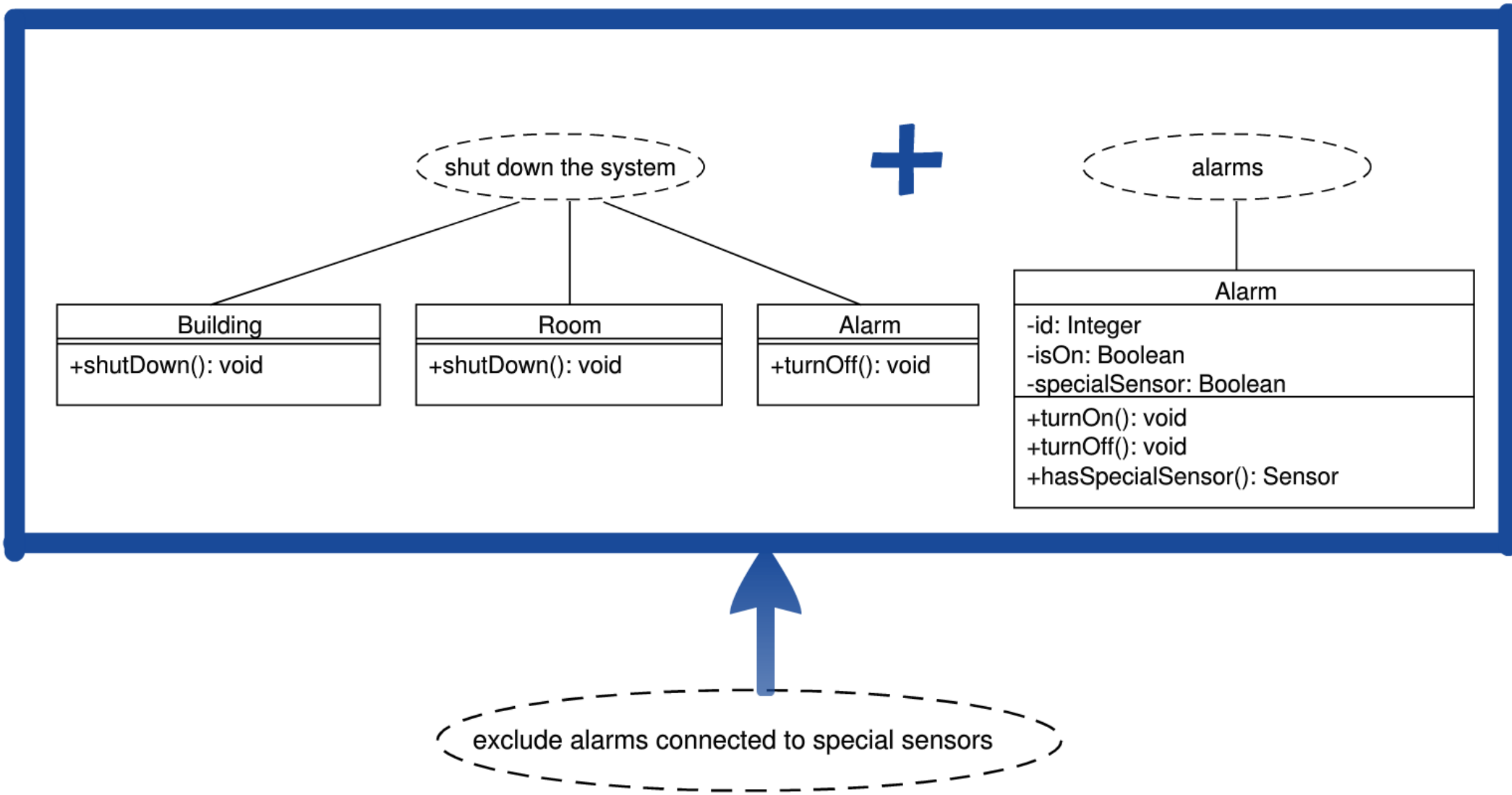
R4: When the security system is being shut down, all its sensors are being turned off.

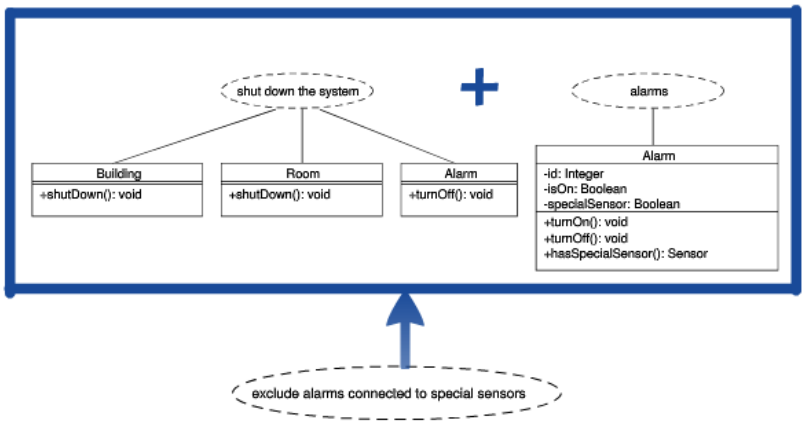
...

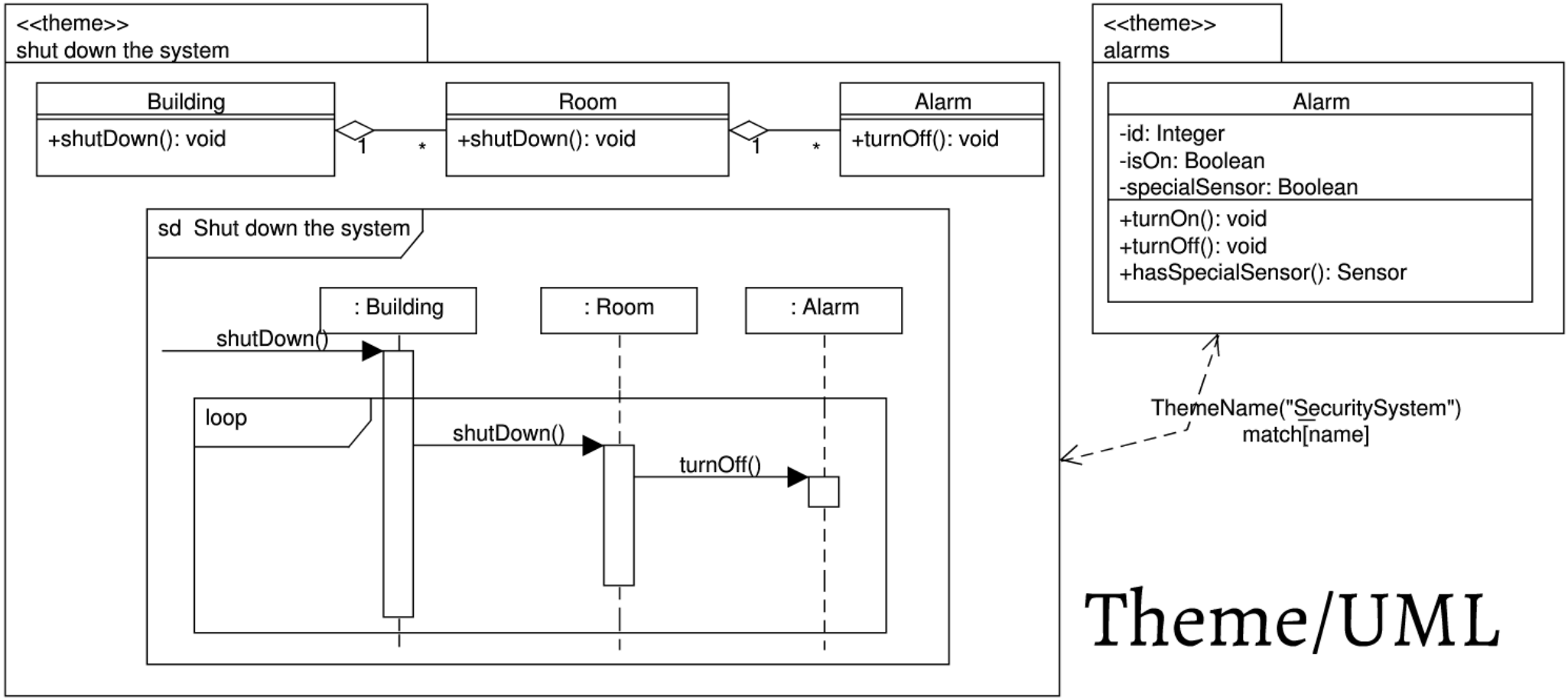
R8: Alarms placed in buildings and connected to special sensors are excluded from being turned off during the shut down of the whole security system.



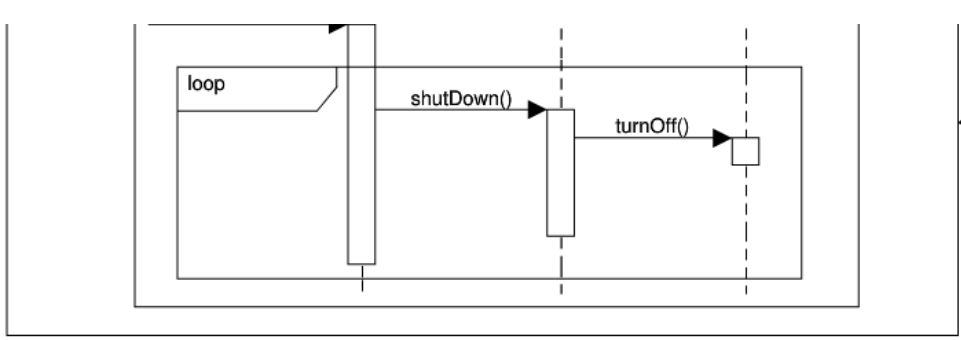
A. Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.


shut down the system
alarms
Building
+shutDown(): void
Room
+shutDown(): void
Alarm
+turnOff(): void
Alarm
-id: Integer
-isOn: Boolean
-specialSensor: Boolean
+turnOn(): void
+turnOff(): void
+hasSpecialSensor(): Sensor
exclude alarms connected to special sensors

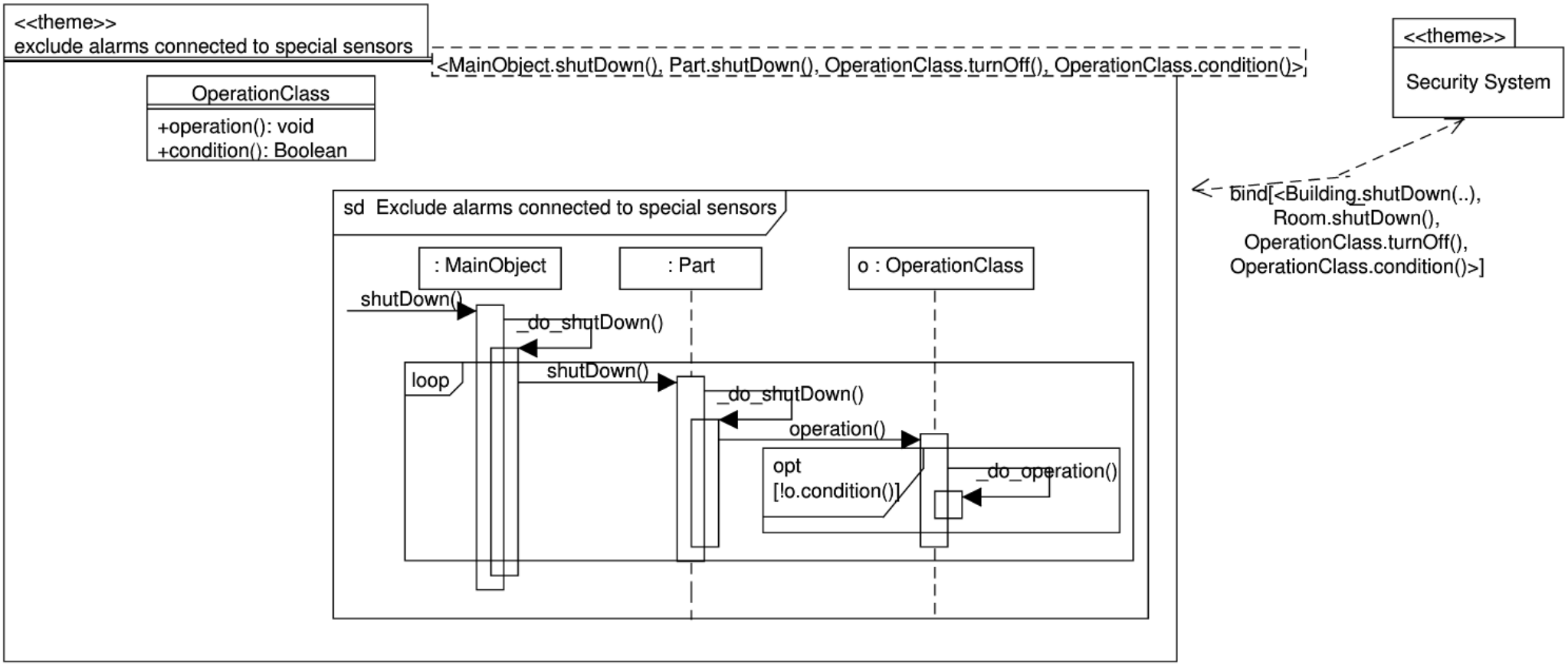

shut down the system
alarms
Building
+shutDown(): void
Room
+shutDown(): void
Alarm
+turnOff(): void
Alarm
-id: Integer
-isOn: Boolean
-specialSensor: Boolean
+turnOn(): void
+turnOff(): void
+hasSpecialSensor(): Sensor
exclude alarms connected to special sensors

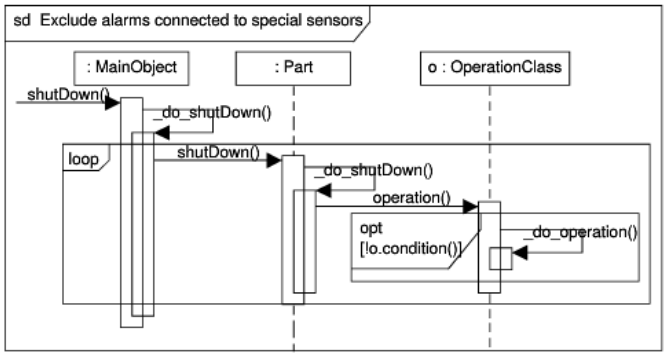

<<theme>>
shut down the system
Building
+shutDown(): void
1
*
Room
+shutDown(): void
1
*
Alarm
+turnOff(): void
sd Shut down the system
: Building
: Room
: Alarm
shutDown()
loop
shutDown()
turnOff()
<<theme>>
alarms
Alarm
-id: Integer
-isOn: Boolean
-specialSensor: Boolean
+turnOn(): void
+turnOff(): void
+hasSpecialSensor(): Sensor
ThemeName("SecuritySystem")
match[name]

Theme/UML


loop
shutDown()
turnOff()
ThemeName("SecuritySystem")
match[name]

Theme/UML


<<theme>>
exclude alarms connected to special sensors
<MainObject.shutDown(), Part.shutDown(), OperationClass.turnOff(), OperationClass.condition()>
OperationClass
+operation(): void
+condition(): Boolean
sd Exclude alarms connected to special sensors
: MainObject
: Part
o : OperationClass
shutDown()
_do_shutDown()
loop
shutDown()
_do_shutDown()
operation()
opt
[!o.condition()]
_do_operation()
<<theme>>
Security System
bind[<Building.shutDown(..),
Room.shutDown(),
OperationClass.turnOff(),
OperationClass.condition()>]



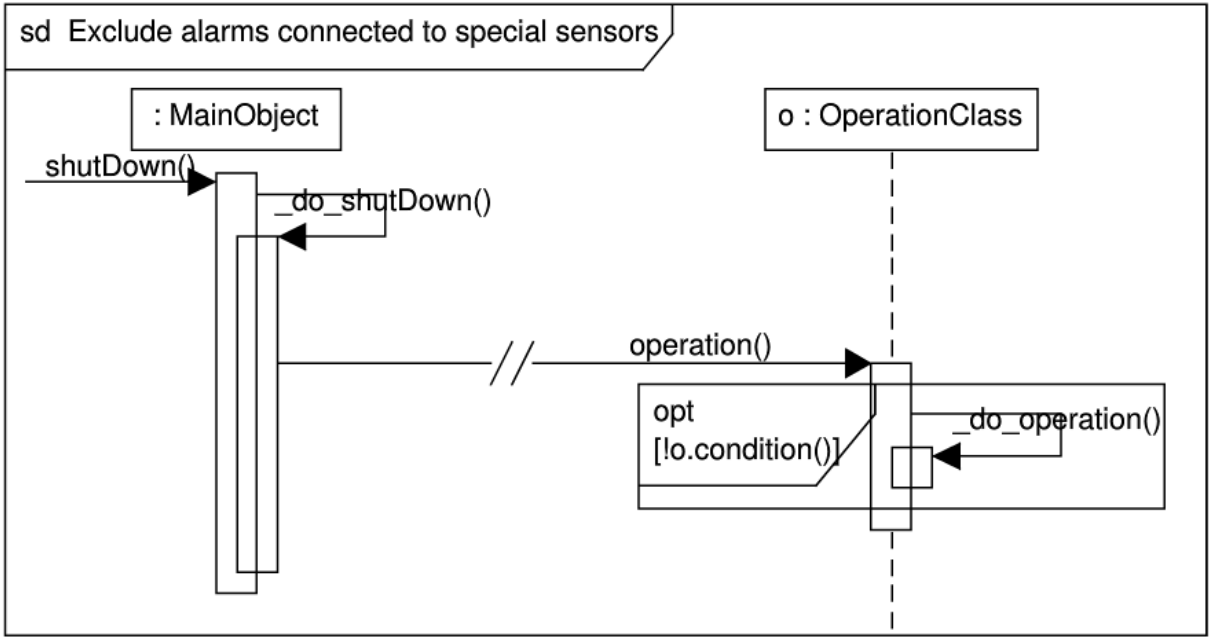
<- bind[<Building.shutDown(..),
Room.shutDown(),
OperationClass.turnOff(),
OperationClass.condition()>]

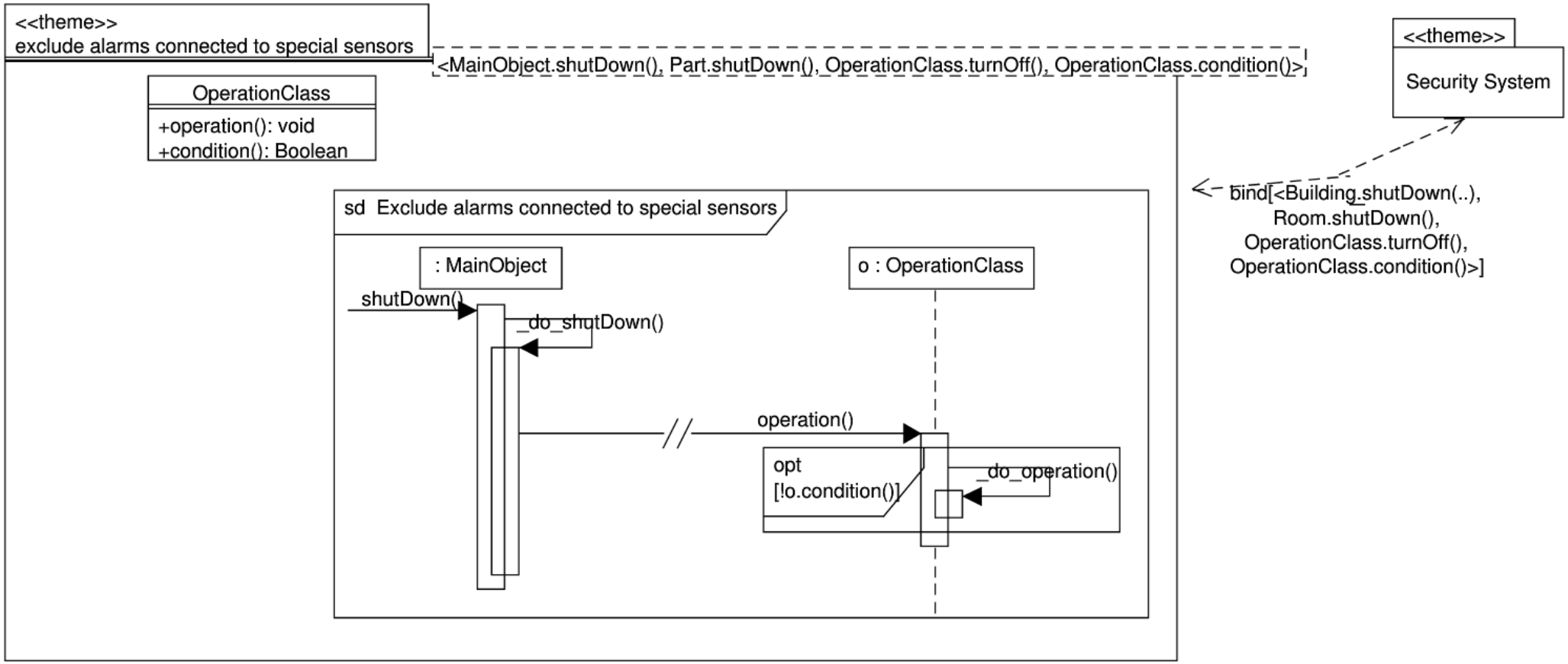
<<theme>>
exclude alarms connected to special sensors

<MainObject.shutDown(), Part.shutDown(), OperationClass.turnOff(), OperationClass.condition()>

OperationClass
+operation(): void
+condition(): Boolean

<- bind[<Building.shutD
Room.shutDow
OperationClass.tur
OperationClass.cond






```
pointcut operations(OperationClass o):
    target(o) && call(* turnOff()) && cflow(call(* Building.shutDown(..)));
```

```
public abstract aspect ConditionallySkeepOperations {

    interface OperationClass {
        boolean condition();
    }

    abstract pointcut operations(OperationClass o);

    void around(OperationClass o): operations(o) {
        if (!o.condition())
            proceed(o);
        }
}
```

```
public abstract aspect ConditionallySkeepOperations {

    interface OperationClass {
        boolean condition();
    }

    abstract pointcut operations(OperationClass o);

    void around(OperationClass o): operations(o) {
        if (!o.condition())
            proceed(o);
        }
}
```

```
public aspect ExcludeAlarmsConnectedToSpecialSensors extends
    ConditionallySkeepOperations {

    declare parents: Alarm implements OperationClass;

    public boolean Alarm.condition() {
        return hasSpecialSensor();
    }

    pointcut operations(OperationClass o): target(o) && call(* turnOff(..));
}
```

Aspect-oriented features are available in popular programming languages

- Traits (Scala)
- Open classes (Ruby)
- Prototypes (JavaScript)
- Decorators (Python)

> Aspect-oriented programming enables to affect existing code without having to actually change it

> Aspect-oriented modularization is natural already at the level of use cases

> UML could embrace aspect-oriented modeling

tinyurl.com/aspects-sing