Component And Composite Approach to Aspect-Oriented Programming

Applications in Software Product Line Evolution

Software Component

Software component **is unit of composition** with explicitly determined (approval) and required interfaces and dependencies. Similarly, it can be independently deployed and is **subject of composition** performed by third parties. Component has no externally observable state.

Source: SZYPERSKI, Clemens, 2002. Component Software : Beyond Object-Oriented Programming. ISBN 0-201-745572-0.

Unit of modularization exceeding ***the size of a single class*** that may be connected with other such units in different contexts.

Rather composed?

Composition Of Components

SOA

Services are composed based on
on Loose coupling

Architectural level

Aspects

Aspects are composed with the rest of the code

Implementational level

Microservices

Micro-frontend

Using orchestrator like Docker Swarm or Kubernetes

Architectural level

Composition Of Components

Aspects

Aspects are composed with the rest of the code

Implementational level

SOA

Services are composed based on
on Loose coupling

Architectural level

Microservices

Micro-frontend

Using orchestrator like Docker Swarm or Kubernetes

Architectural level

… how can these respective components be used to effectively handle variability?

Independent Pointcuts On Specific Join Points

As are in Themes/UML approach

ABSTRACT ASPECTS Use abstract pointcuts, and specify their behavior later for particular case

To reuse reuse components in different contexts

Managing internal structure on implementational level:

Exceeding exposed implantation of component … :

- Propagating shared roles to particular entities
- Preparing shared implementation without implementational details
- Independent pointcuts on specific join points

State

Accompanied with the prescription of operations that can be taken

…when sequence of operations without any flag to rely upon is on the input

Where is the state in the code?

Where are states in the code?

Something like **State machine diagrams**

Stateful Pointcuts in JAsCo

State Dependent Behaviour

SOFTWARE BEHAVIOUR

Highly interactive and state dependent

STATE MACHINES

states

state transitions

behavior dependencies on state

No explicit support of their abstractions in natural oop

PROBLEM TO DECOMPOSE THEIR DEFINITION

To Ensure Stability and Consistency

EXPLICIT STRUCTURE TO STATE MACHINES

JAsCo

Java Aspect Composition

- Research project from Brussels Vrije available online: http://ssel.vub.ac.be/jasco/
- Aspect oriented programming language tailored for component-based use
- Run time weaver
- Highly reusable aspects and composition mechanism
- Capabilities:
  - Stateful pointcuts – incorporation of state management
  - Connectors – separation of the way how aspects are connected to specific join points [Adaptive Programming]
- Unmaintained, academic project
- Harder learning curve then in AspectJ

Links

- Source from : http://ssel.vub.ac.be/jasco/lib/exe/fetch2cc0.pdf?cache=cache&media=taosd05.pdf
- Download to run: http://ssel.vub.ac.be/jasco/kernel.html
- Documentation: http://ssel.vub.ac.be/jasco/documentation_main.html

Hooks in JAsCo

Is some kind of **advice** and like **nested class**

Stored/expressed in classes as hooks

constructor

Pointcut is defined in constructor of the hook

Defining when hook is active:

method **isApplicable()**

```
class MyClass {
    boolean turnedOn = false;

    hook MyHook {
        MyHook(amethod (..args)) {
            execute(amethod);
        }
        isApplicable() { return global.turnedOn; }
        around() { System.out.println("around"); }
    }
}
```

Method will be specified

Connector in JAsCo

Defining the connector with the hook

Connects aspects with code – without them aspects are not modifying the original program

As a binding in Theme/UML

Example:

```
static connector MyConnector {
    MyClass.MyHook hook0 = new p.MyClass.MyHook(* *.*(..));
}
```

Applicable on all methods

Source: Aspekty v analýze a návrhu – Komponentový a kompozičný prístup k aspektovo-orientovanému programovaniu Poznámky k prednáškam z predmetu Aspektovo-orientovaný vývoja softvéru, Valentino Vranić, 2017

State

Accompanied with the prescription of operations that can be taken

…when sequence of operations without any flag to rely upon is on the input

Where is the state in the code?

Where are states in the code?

Something like **State machine diagrams**

Stateful Pointcuts in JAsCo

Stateful Code - Core

Used methods that should be executed

```
public class MyLogger {
    hook LogHook {
        LogHook(startmethod(..a1), logmethod(..a2), stopm(..a3)) {
            start>p1;
            p1: execute(startmethod) > p3||p2;
            p2: execute(logmethod) > p3||p2;
            p3: execute(stopm) > p1;
        }

        before() {
            System.out.println(thisJoinPoint.getName());
        } \\ ak chceme len p2(): before p2() { . . . }
    }
}
```

Type of transitions

a) Explicit start transition

b) Two possible destination transitions

c) Two start transitions:

```
start>p1 || p2;
```

d) No transition:

```
p2: execute(logmethod)
        && !cflow(stopm);
```

Connector for Stateful Code

Defining the connector with the hook

Connects aspects with code – without them aspects are not modifying the original program

As a binding in Theme/UML

```
static connector MyLoggerConnector {
    MyLogger.LogHook hook0 = new MyLogger.LogHook(
        void mypackage.Main.x(),
        void mypackage.Main.y(),
        void mypackage.Main.z()
    );
}
```

Example Program With State

Defining the methods and custom program

```
public class Main {
    public static void main(String[] args) {
        Main m = new Main();
        m.x();
        m.x();
    }
    public void x() {  y(); }
    public void y() { z(); }
    public void z() { zzz(); }
    public void zzz() { System.out.println(”out”); }
}
```

Output:

```
x
y
z
out
x
y
z
out
```

Another Example

Method does not move from the initial state

```
public static void main(String[] args) {
    Main m = new Main();
    m.z();
    m.z();
}
```

Output:

out
out

JAsCo and Adaptive Programming

- **Law of Demeter** – method can communicate only with arguments, part objects (stored or computed) and newly created objects

```
class ViolatingDemetersLaw {
  MyClass myClass;

  public void do() {
     myClass.getVariable1().doSomething(-1);
  }
}
```

Resolved with:

```
class DemetersLaw1 {
  MyClass2 myClass2;

  public void do() {
     myClass2.do (-1);
  }
}
```

```
class MyClass2 {
  MyClass3 myClass3;

  public void do(int i) {
     myClass3.doSomething(i);
  }
}
```

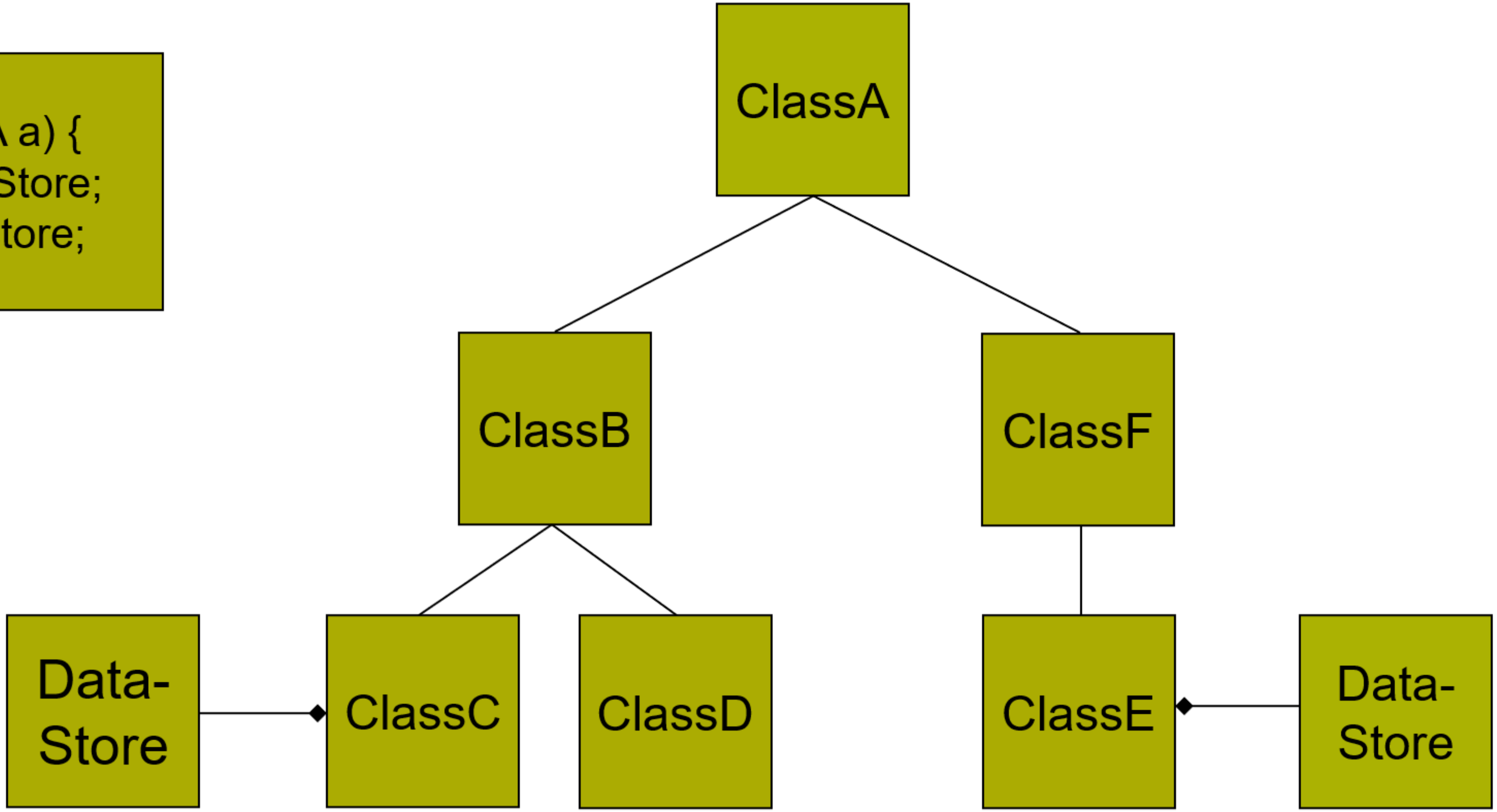
Still, Do() logic pollutes code

- is scattered in several classes

→ Save contents of DataStores to file …

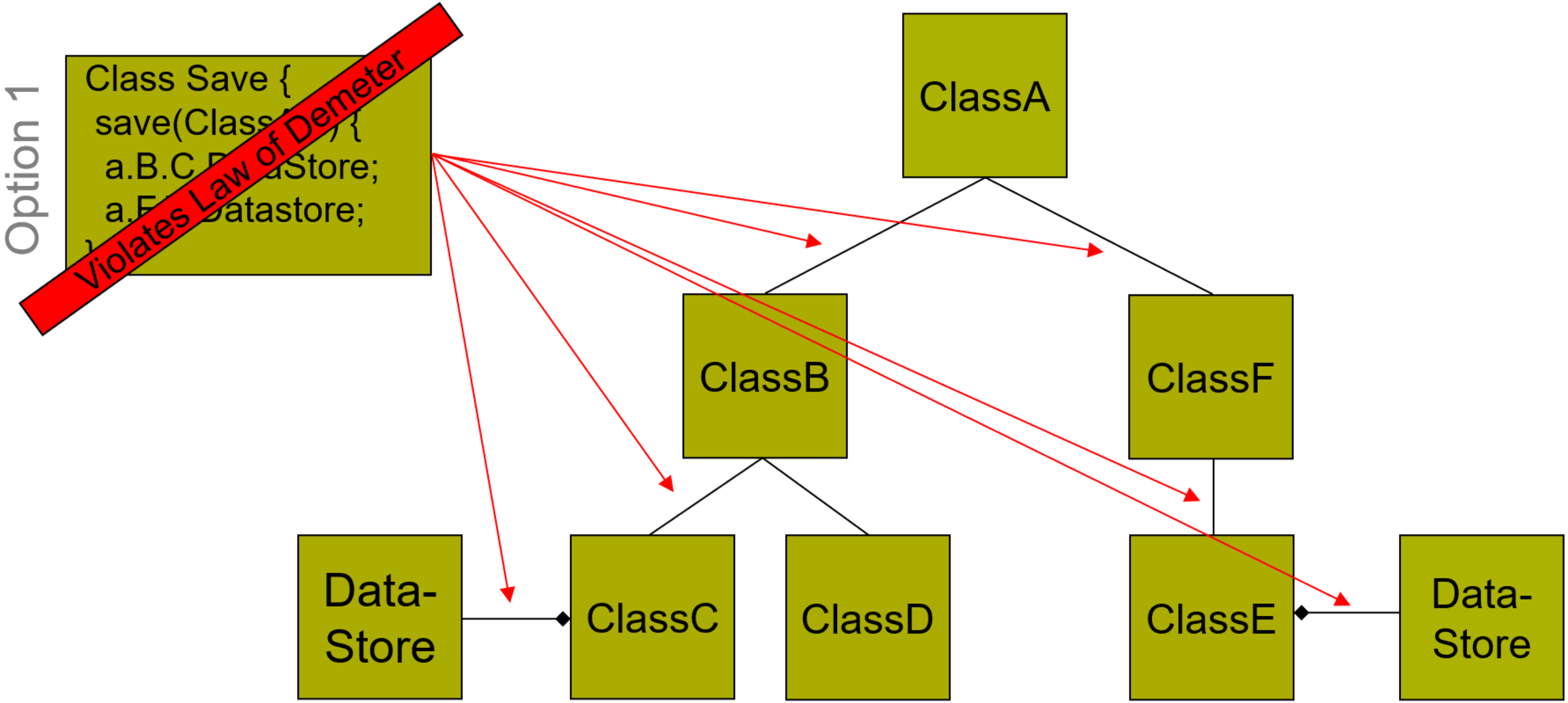
Option 1

```
Class Save {
 save(ClassA a) {
  a.B.C.DataStore;
  a.F.E.Datastore;
}
```



Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

→ Save contents of DataStores to file …

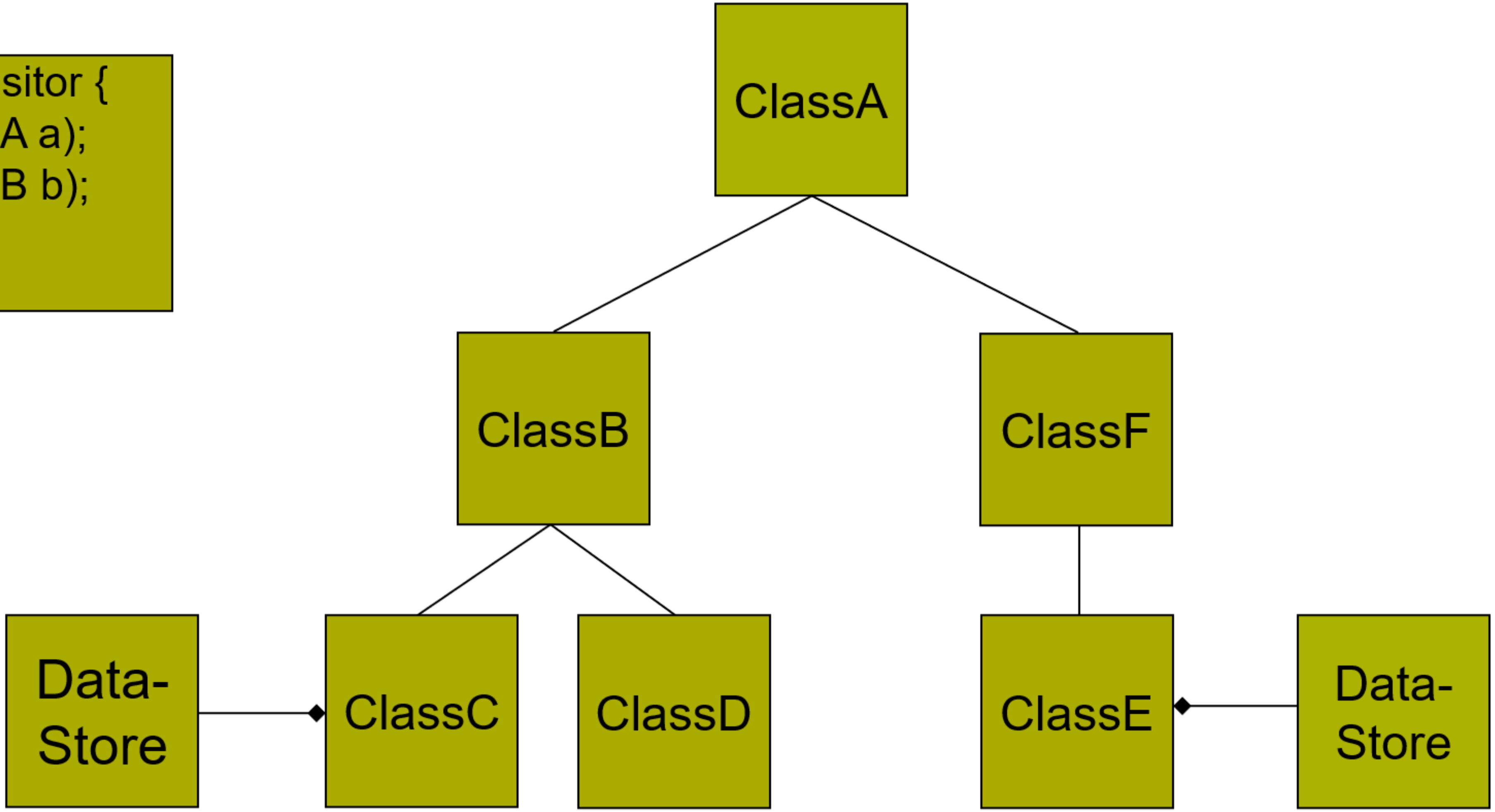


Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

→ Save contents of DataStores to file …

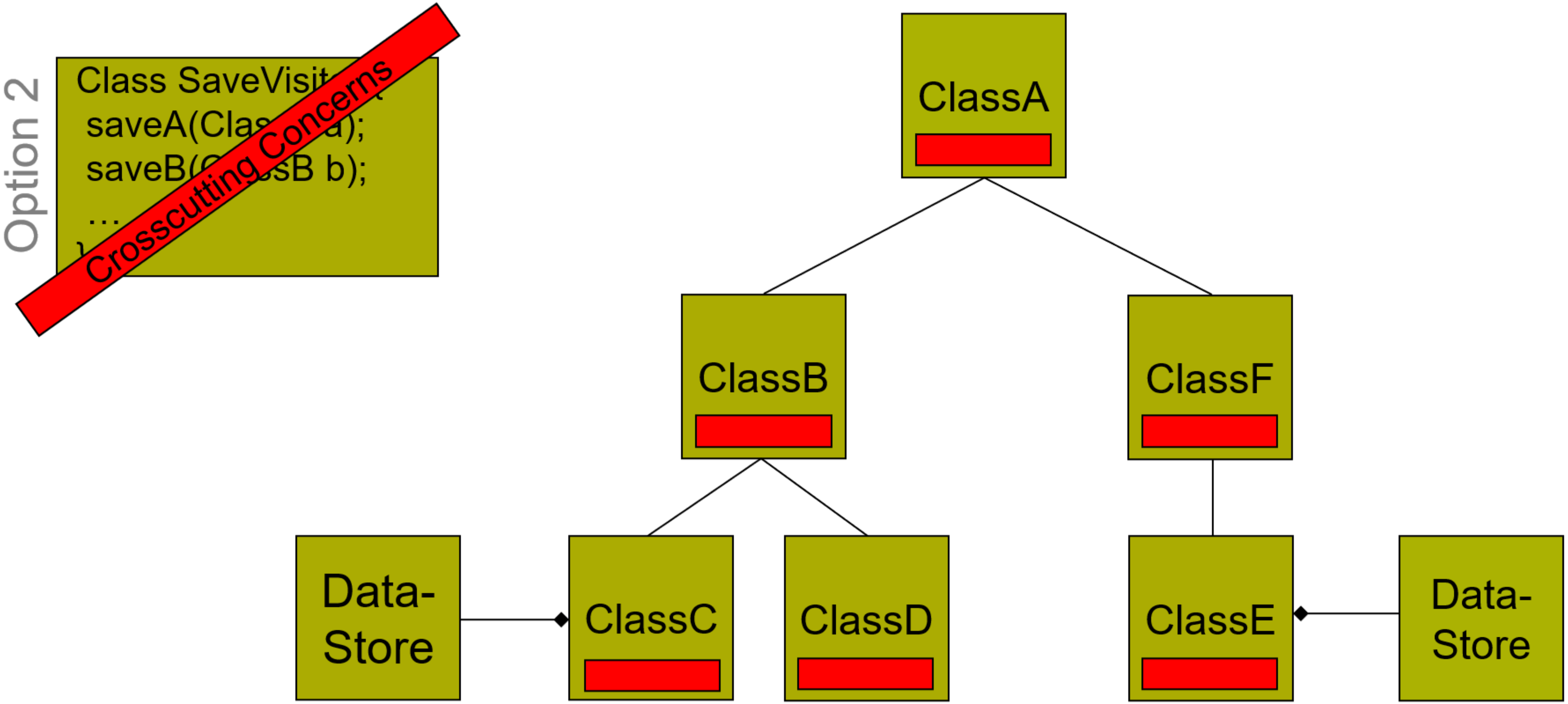
Option 2

```
Class SaveVisitor {
 saveA(ClassA a);
 saveB(ClassB b);
 …
}
```



Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

→ Save contents of DataStores to file …



Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

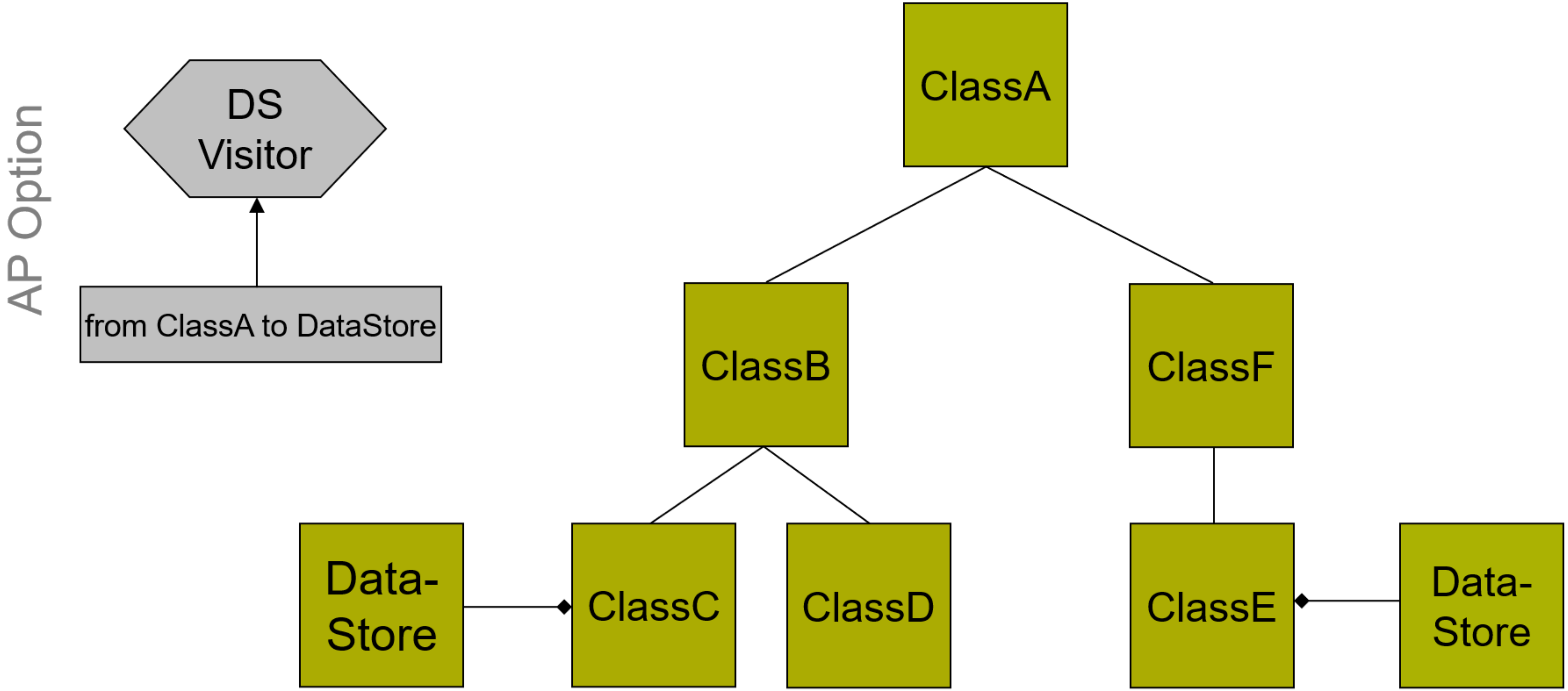
Use of Adaptive Programming to Solve Following Dilemma

Dilemma:

- Crosscutting concerns (if follow) or
- Unmaintainable code (if not follow)

Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

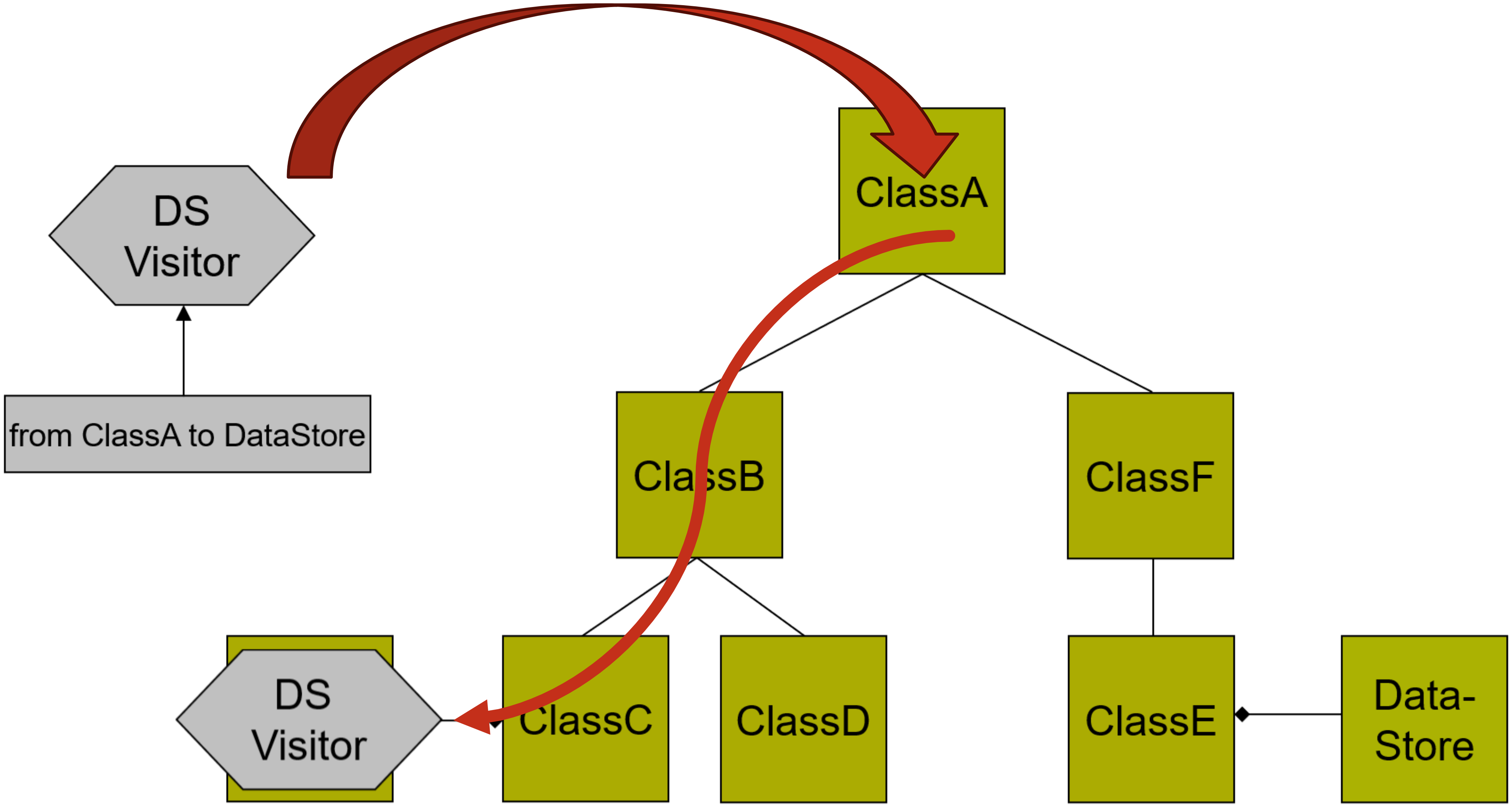
→ Save contents of DataStores to file …



Source:
http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt

→ Save contents of DataStores to file …

AP Option



Implementation in JAsCo

Transversal strategy

Traversal that is followed

Example: from ClassA to DataStore

Adaptive visitor

-traverses the specified path
-contains implementation of behaviour

JAsCo Code for Adaptive Programming

```
class DataStorePersistence extends Visitor {

   public void before(DataStore store) {
      if(changedPV(store)) {
         Writer writer = …
         writer.writeObject(store.getData());
      }
   }

   public boolean changedPV(DataStore s) {
      \\ true if changed since last visit
   }
}
```

```
void backup(ClassA a) {
   ClassGraph cg = new ClassGraph("a");
   Strategy sg = new Strategy("from ClassA to DataStore");
   TraversalGraph tg = new TraversalGraph(sg, cg);
   tg.traverse(a, new DataStorePersistence());
}
```

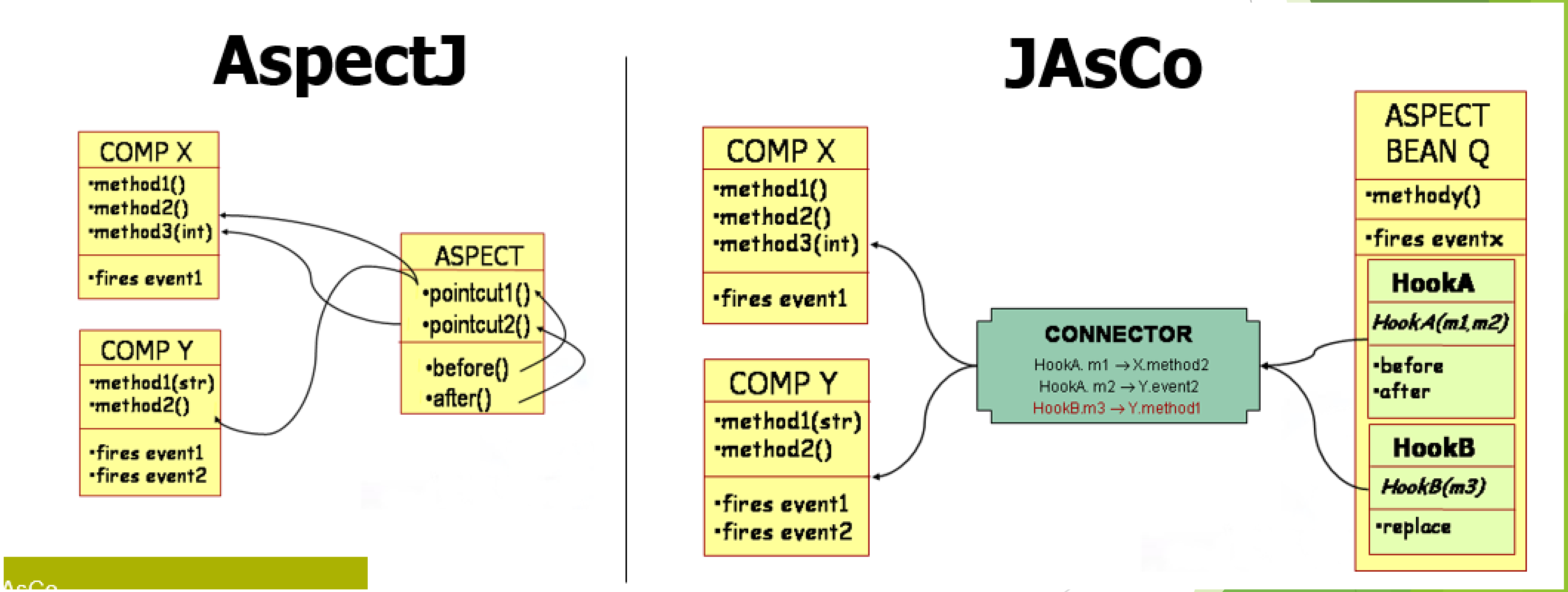
JAsCo Code for Adaptive Programming

```
class DataPersistence {

  hook Backup {

    Backup(triggeringmethod(..args)) {
      execution(triggeringmethod); }

    isApplicable() {
     // true when changed since last visit }

    before() {
      Writer writer = …
      writer.writeObject(getDataMethod());
    }

    refinable Object getDataMethod();

  }
}
```

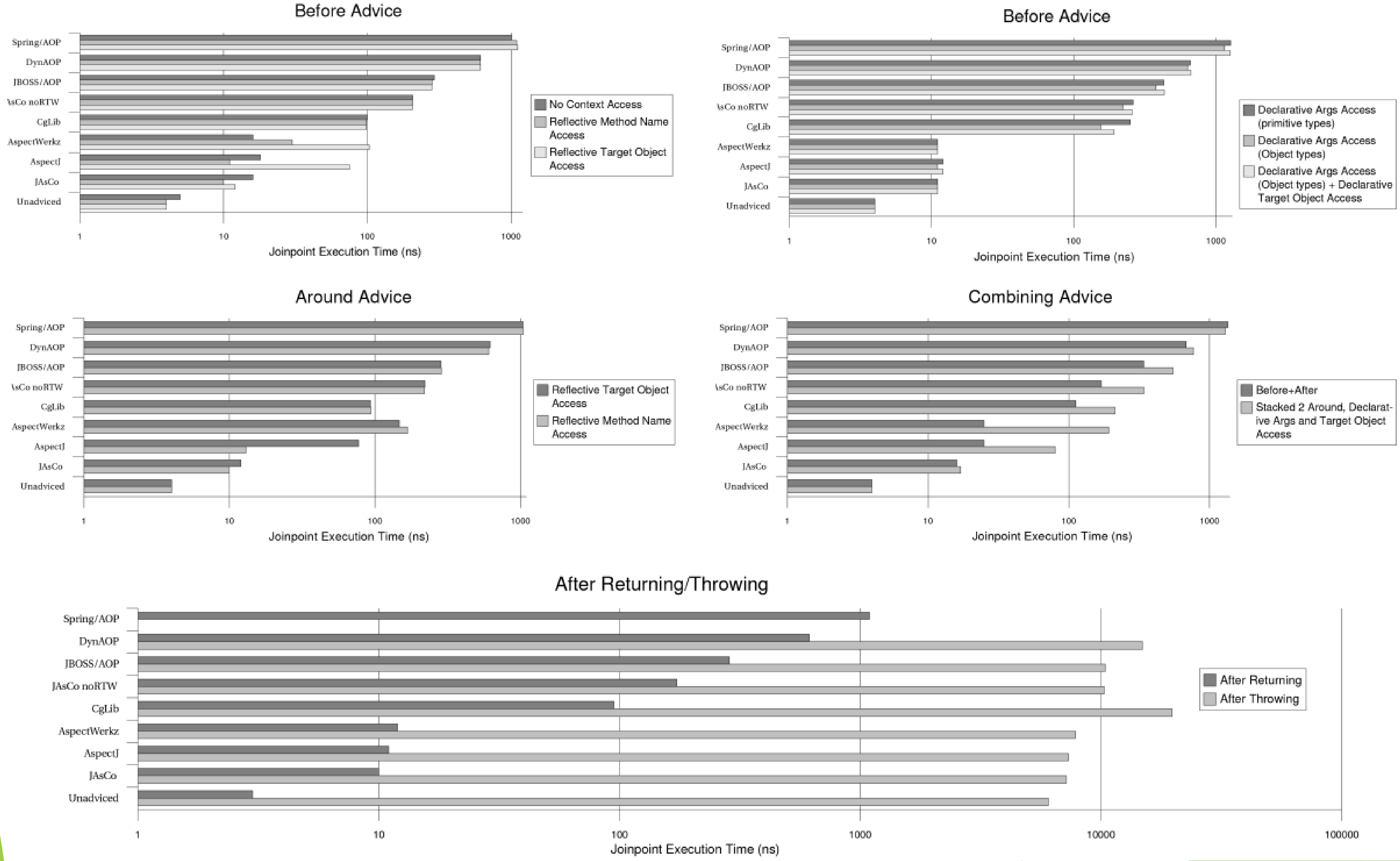
```
connector PersistenceConnector {
  DataPersistence.Backup hook = new
    DataPersistence.Backup(
      * DataStore.set*(*));
}
```

```
refining DataPersistence.Backup
  for DataStore {
    public Object getDataMethod() {
      DataStore d = thisJoinPointObject;
      return d.getData();
    }
}
```

Comparison With AspectJ



Source: http://ssel.vub.ac.be/jasco/lib/exe/fetchb6e2.php?cache=cache&media=documentation%3Ajasco-vub-session1.ppt



Source: http://ssel.vub.ac.be/jasco/lib/exe/fetche3d5.png?cache=cache&media=benchdrawing2.png

New Dimension in Software: Variability

Software product lines as complex software intensive systems

Increased complexity:

-evolution of common assets

-independent evolution of products and their instantiation

Independent on structure, behaviour, and its state

- problem with modularization

Modularizing different kinds of variability

Preserving encapsulation and communication over explicit interfaced

Feature-Oriented Decomposition

Analysis only of those modules of the features that are available in product

Design benefits

- *Independent evolution of product line features*
- *Modular introduction of new features*
- *Does not always reflect on technical commonalities between features*

Does not rely on one-to-one correspondence of features from feature model and features describing variations in software product line and implementation modules

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Modularization And Components Applied in Software Product Lines:

Creation of houses with rooms and floors

In OOP world / In AOP world

ECaesarJ

Application to evolution of software product lines

- Type-safe
- Stable decomposition of majority of software abstractions:
  - Classes
  - Methods
  - Events
  - State
    - Binding
    - Mixin Composition

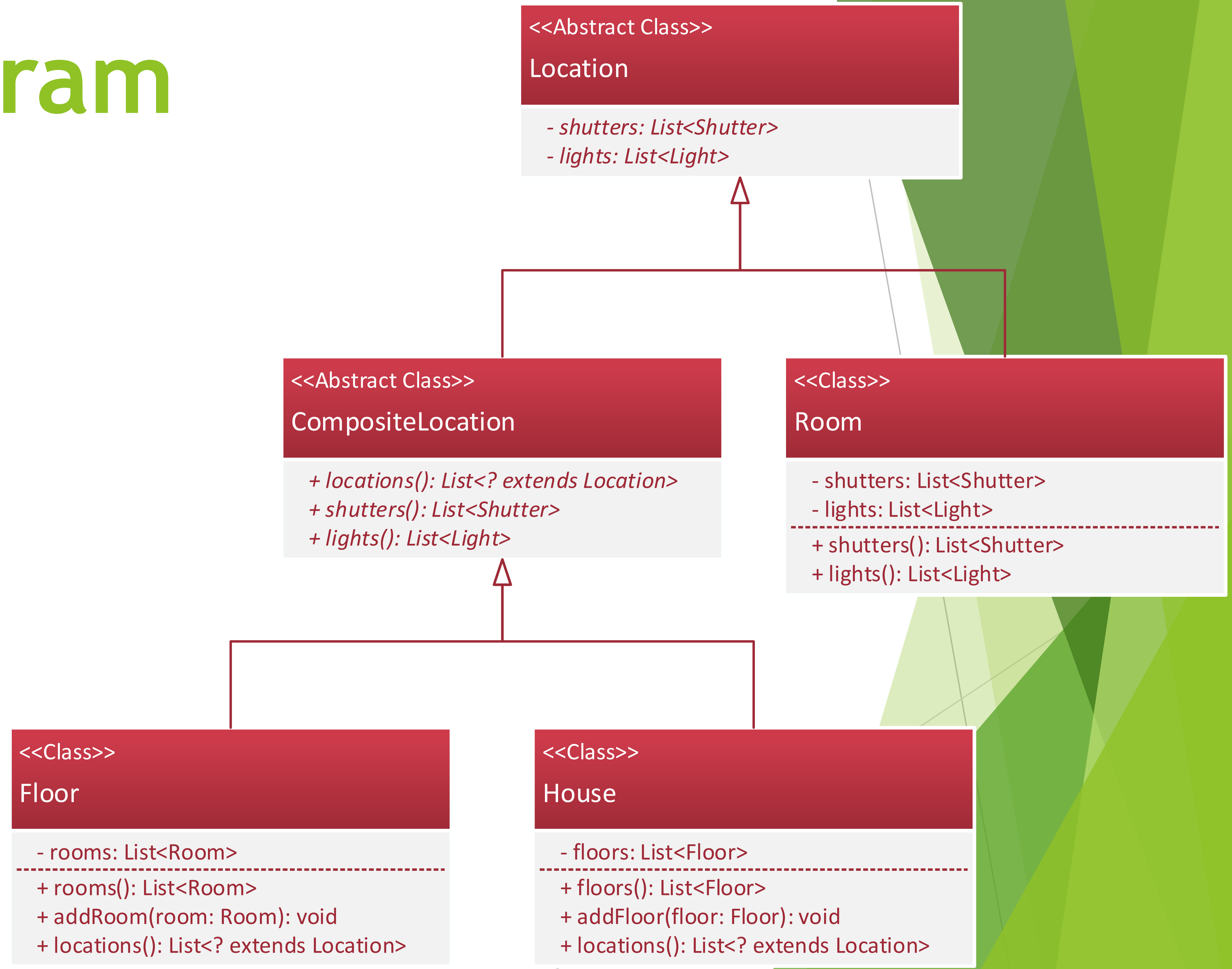
Object-Oriented Solution

```
abstract class Location {
    abstract List<Shutter> shutters();
    abstract List<Light> lights();
    ...
}

abstract class CompositeLocation extends Location {
    abstract List<? extends Location> locations();
    List<Shutter> shutters() {
        List<Shutter> shutters = new ArrayList<Shutter>();
        for (Location child : locations()) {
            shutters.addAll(child.shutters())
        }
        return shutters;
    }
    List<Light> lights() { ... }
    ...
}

class Room extends Location {
    List<Shutter> shutters;
    List<Light> lights;
    List<Shutter> shutters() { return shutters; }
    List<Light> lights() { return lights; }
    ...
}
```

```

class Floor extends CompositeLocation {
    List<Room> rooms;
    List<Room> rooms() { return rooms; }
    void addRoom(Room r) { rooms.add(r); }
    List<? extends Location> locations() { return rooms(); }
}

class House extends CompositeLocation {
    List<Floor> floors;
    List<Floor> floors() { return floors; }
    void addFloor(Floor r) { floors.add(r); }
    List<? extends Location> locations() { return floors(); }

    static House house = new House();
    static House houseInstance() { return house; }
}
```

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Class diagram



Attempt to extend original structure with heaters

to support heating features

Extended only individually:

Not replaced by their extensions (extended code) in their relation with other classes

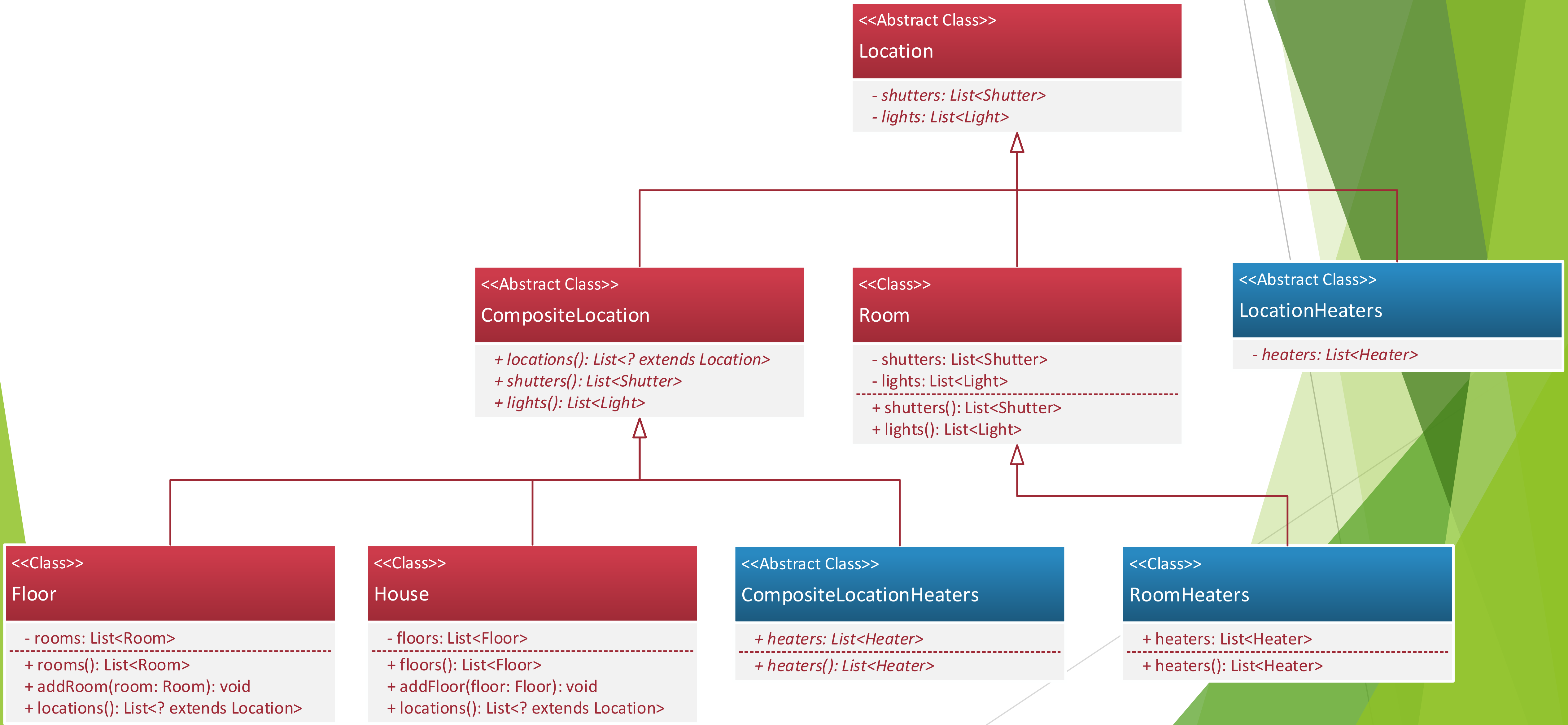
Example: Not all type of rooms will have heaters after extension is applied

```
abstract class LocationHeaters extends Location {
    abstract List<Heater> heaters();
    ...
}

abstract class CompositeLocationHeaters extends CompositeLocation {
    List<Heater> heaters() {
        List<Heater> heaters = new ArrayList<Heater>();
        for (Location child : locations()) {
            heaters.addAll(child.heaters())
        }
        return heaters;
    }
}

class RoomHeaters extends Room {
    List<Heater> heaters;
    List<Heater> heaters() { return heaters; }
}
```

Figure 6.2 An attempt to extend the house structure with heaters.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Extended class diagram



Modularization of Static Structures

1. No replacement of classes in inheritance relationship.

-redeclaration of inheritance relationship between classes

MULTIPLE INHERITANCE Impossible in Java

- introduction of additional glue code (for example in C++)

2. Incorporation of planned functionality (domain knowledge)

INVASIVE CHANGES OTHERWISE

- *If open-closed principle cannot be used = <u>implementation of core features has to be changed during introduction of product-specific feature</u>*

3. Instantiation of extended classes instead of original ones

REDEFINITION IN ALL PLACES WHERE EXTENDED CLASS IS INSTANTIATED

Here:

```
static House house = new House();
```

Implicit Events

-integrates the pointcuts of AspectJ

Not explicitly triggered

-occur as result of other event occurences

Define events in declarative way

As expressions over other events

TYPES

-possibly to specialize based on introduced conditions on parameters exposed by events

-on values computed in the scope of enclosing class

-as composition of other events using logic operators

```
abstract cclass ILightSensorFeature {
    abstract cclass ILightSensor {
        abstract event intensityChanged();
        abstract int getIntensity();
    }
}

cclass LightSensorFeature extends ILightSensorFeature requires IDeviceUpdate {
    cclass DefaultLightSensor extends ILightSensor {
        int intensity;
        event intensityChanged();
        int getIntensity() { return intensity; }

        update() => {
            int oldIntensity = intensity;
            scanLightIntensity();
            if (oldIntensity != intensity) {
                intensityChanged();
            }
        }
        ...
    }
}
```

Definition of event as class member

Definition of event as class members

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Figure 6.11 An implementation of a light sensor device driver.

Implicit Events

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

DECOUPLED EVENT FROM ITS EVENT DEFINITION

-[expr.]eventName(formalArgs) => body

```
abstract cclass IDaytimeFeature {
    abstract event void sunrise();
    abstract event void sunset();
    abstract boolean isDaytime();
}

abstract cclass ILightSensorLocatorFeature requires ILightSensorFeature {
    abstract ILightSensor lightSensor();
}

cclass SensorDaytimeFeature extends IDaytimeFeature requires ILightSensorFeature {
    final int THRESHOLD = 80;
    protected boolean daytime;

    event sunrise() = lightSensor().intensityChanged()
        && if(!daytime && lightSensor().getIntensity() > THRESHOLD);
    event sunset() = lightSensor().intensityChanged()
        && if(daytime && lightSensor().getIntensity() < THRESHOLD);

    public boolean isDaytime() { return daytime; }

    sunrise() => { daytime = true; }
    sunset() => { daytime = false; }
}
```

Figure 6.12 Two alternative implementations of daytime events.

```
abstract cclass IActivityDetection {
    abstract event activityDetected();
}

cclass AlarmControl requires IActivityDetection {
    event intrusion() = activityDetected() && if(isArmed());
    intrusion() => {
        fireAlarm();
    }
    ...
}
```

Figure 6.15 Alarm control using detection of activity in the house.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Mixin Composition Of Events

```
cclass DefaultActivityDetection extends IActivityDetection {
    event activityDetected() = empty;
}

cclass MotionActivityDetection extends DefaultActivityDetection
                                        requires IHouseMotionSensors {
    mixin event activityDetected() = super.activityDetected() ||
        some(house().motionSensors()).motionDetected();
}

cclass LightActivityDetection extends DefaultActivityDetection
                                        requires IHouseLights {
    mixin event activityDetected() = super.activityDetected() ||
        some(house().lights()).turnOn() ||
        some(house().lights()).turnOff();
}

...

cclass ActivityDetectionInMyHouse extends MotionActivityDetectio
    & DoorActivityDetection & LightActivityDetection { }
```

Figure 6.16 Variations of activity detection.

To mark conflicting situations to resolve them automatically – use of mixin keyword

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Handling Event Using Observer Design Pattern

Modularization of behaviour

1. Much glue code - registration and notification of observers

2. Incorporation of planned functionality (domain knowledge)

INVASIVE CHANGES OTHERWISE

- *If open-closed principle cannot be used =* ***implementation of core features has to be changed during introduction of product-specific feature***

3. No support of declarative definition of events

- *Difficult to reuse for defining other kind of events*

EXPLICIT TRIGGERING IS NECESSARY

Use of ECaesarJ

Why?

Insufficient mechanisms in OOP for modularizing the features

Only **for individual objects/classes**

Multiple affected objects/classes by Features

Large-scale extension mechanism

PROVIDES LARGE-SCALE SELECTION AND COMPOSITION MECHANISMS

VIRTUAL CLASSES

-inner classes
-late-bound instantiation
-can be refined in subclasses of enclosing class
-as family members
– members of instances of the enclosing class [family objects/classes]

PROPAGATING THE MIXIN COMPOSITION

Composition propagates into virtual classes

ALL INHERITED DECLARATIONS OF VIRTUAL CLASSES WITH THE SAME NAME ARE MERGED AUTOMATICALLY

Virtual Class in ECaesarJ/CaesarJ

FURTHERBINDING

-a refinement of virtual class

-implicit inheritance from refined class

- Adding new functionality
- Overriding existing functionality

Methods, fields, and inheritance relationships

-references to virtual class are bounded to its the most specific refinement

-in family class

REFINEMENT OF VIRTUAL CLASSES

Extending functionality between features:

in subclasses -in inheritance of these virtual classes

REACHING:

Consistent extension of group of classes

defining

Features as extension of other features

modeling

Feature

In family class

Domain objects

In virtual classes

Implementation in ECaesarJ

Feature

In family class

feature-oriented modularization of software product lines

To not confuse this with Java **class**
-to use it as extension of Java

cclass

New CaesarJ dat structure

Domain objects

In virtual classes

```
cclass HouseStructure {
    abstract cclass Location { }

    abstract cclass CompositeLocation extends Location {
        abstract List<? extends Location> locations();
    }

    cclass Room extends Location { }

    cclass Floor extends CompositeLocation {
        List<Room> rooms;
        List<Room> rooms() { return rooms; }
        void addRoom(Room r) { rooms.add(r); }
        List<? extends Location> locations() { return rooms(); }
    }

    cclass House extends CompositeLocation {
        List<Floor> floors;
        List<Floor> floors() { return floors; }
        void addFloor(Floor r) { floors.add(r); }
        List<? extends Location> locations() { return floors(); }
    }

    House house = new House();
    House house() { return house; }
}
```

Figure 6.3 Implementation of a house structure as a family class.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Furtherbinding

-extending features in consistent way

```
cclass HouseShutters extends HouseStructure {
    abstract cclass Location {
        abstract List<Shutter> shutters();
    }

    abstract cclass CompositeLocation {
        List<Shutter> shutters() {
            List<Shutter> shutters = new ArrayList<Shutter>();
            for (Location child : locations()) {
                shutters.addAll(child.shutters())
            }
            return shutters;
        }
    }

    cclass Room {
        List<Shutter> shutters;
        List<Shutter> shutters() { return shutters; }
    }
}
```

Figure 6.4 Extension of house structure with shutter devices.

House structure is extended with shutters

HouseShutters inherits all virtual classes from *HouseStructure* and can ***selectively extend them*** further

Access in type-safe way

-references to virtual classes are always relative to enclosing object

Automatically updated to refer to the new definitions of these classes in family class

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Type-safety of type refinements

- In consideration are only classes from the same

family objects = compatibility

  - the types of nested virtual classes belong to

    their parent this object

Mixins compositions

ORTHOGONAL EXTENSIONS

– contains not overlapping methods – with different name

-no conflicts during their composition

```
cclass HouseLights extends HouseStructure {
    abstract cclass Location {
        abstract List<Light> lights();
    }

    abstract cclass CompositeLocation {
        List<Light> lights() { ... }
    }

    cclass Room {
        List<Light> lights;
        List<Light> lights() { return lights; }
    }
}
```

Figure 6.5 Extension with house structure with light devices.

```
cclass HouseShutters extends HouseStructure {
    abstract cclass Location {
        abstract List<Shutter> shutters();
    }

    abstract cclass CompositeLocation {
        List<Shutter> shutters() {
            List<Shutter> shutters = new ArrayList<Shutter>();
            for (Location child : locations()) {
                shutters.addAll(child.shutters())
            }
            return shutters;
        }
    }

    cclass Room {
        List<Shutter> shutters;
        List<Shutter> shutters() { return shutters; }
    }
}
```

Figure 6.4 Extension of house structure with shutter devices.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Composing Features in CaesarJ/ECaesarJ

Propagating form of mixin compositions

For automated resolution methods should be **explicitly** annotated with keyword ***mixin***

RESOLVING THEIR PRECEDENCE

Linearisation order of composed classes

Composition propagates into virtual classes

ALL INHERITED DECLARATIONS OF VIRTUAL CLASSES WITH THE SAME NAME

Conflicts are manually resolved by developers

All inherited members

Automatically merged: Virtual classes with the same name

-each will contain all unique methods amongst supertypes

For complete composition:

```
complete cclass MyHouse extends LargeHouseStructure
                                & HouseShutters & ShutterEmulator { }
```

Figure 6.10 An example of a complete feature composition.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Composition of Virtual Classes

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

Realization from right to left

<higher> → <lower> **Priority**

COMPOSITION STEP 1:

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

(**Layer2** **&** Layer1)

overwrites

COMPOSITION STEP 2:

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

cclass Composition **extends**
[**Layer3** **&** (Layer2 **&** Layer1)] { }

overwrites

Decoupling features further:

Abstract family classes

CANNOT BE INSTANTIATED!!!

Describes common abstractions
of multiple features
and interfaces between them

CAN BE USED AS INTERFACES

DESCRIBES ABSTRACTIONS OF OBJECT
FAMILIES NOT ONLY OF INDIVIDUAL OBJECTS

Possibly contains:

- Abstract methods
- Incomplete implemented virtual classes

BENEFITS IN INCORPORATION OF ABSTRACTIONS:

-to hide implementation details of features
-to abstract from alternative variations

On example: Hiding variations of House structure

THREE LEVELS OF ABSTRACTIONS over House entity:

```
abstract cclass IHouseLocations {
    abstract cclass Location { }
    abstract cclass CompositeLocation extends Location { }
    cclass Room extends Location { }
    cclass House extends CompositeLocation { }
}

abstract cclass IHouseStructure extends IHouseLocations {
    abstract cclass CompositeLocation extends Location {
        abstract List<? extends Location> locations();
    }
    abstract House house();
}

abstract cclass AbstractHouseStructure extends IHouseLocations {
    House house = new House();
    House house() { return house; }
}
```

Figure 6.6 House structure abstractions.

Extending abstract family classes

```
cclass LargeHouseStructure extends AbstractHouseStructure {
    cclass House {
        List<Floor> floors;
        List<Room> floors() { return floors; }
        List<? extends Location> locations() { return floors(); }
    }

    cclass Floor extends CompositeLocation {
        List<Room> rooms;
        List<Room> rooms() { return rooms; }
        List<? extends Location> locations() { return rooms(); }
    }
}
```

Figure 6.7 Large house structure.

LARGE HOUSES

Covers following variations

SMALL HOUSES

```
cclass SmallHouseStructure extends AbstractHouseStructure {
    cclass House {
        List<Room> rooms;
        List<Room> rooms() { return rooms; }
        void addRoom(Room r) { rooms.add(r); }
        List<? extends Location> locations() { return rooms(); }
    }
}
```

Figure 6.8 Small house structure.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

Interfaces in ECaesarJ

Keyword *required*:

- To implement the interface

Redesign to explicit interfaces:

```
abstract cclass IShutterDevices { ... }
cclass ShutterEmulator extends IShutterDevices { ... }

abstract cclass IHouseShutters extends IHouseLocations requires IShutterDevices {
    abstract cclass Location {
        abstract List<Shutter> shutters();
    }

    abstract cclass CompositeLocation {
        provided List<Shutter> shutters();
    }
}

cclass HouseShutters extends IHouseShutters requires IHouseStructure {
    abstract cclass CompositeLocation {
        List<Shutter> shutters() {
            ... locations() ...
        }
    }

    cclass Room {
        List<Shutter> shutters;
        List<Shutter> shutters() { return shutters; }
    }
}
```

Figure 6.9 Explicit interfaces for HouseShutters.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

```
cclass HouseShutters extends HouseStructure {
    abstract cclass Location {
        abstract List<Shutter> shutters();
    }

    abstract cclass CompositeLocation {
        List<Shutter> shutters() {
            List<Shutter> shutters = new ArrayList<Shutter>();
            for (Location child : locations()) {
                shutters.addAll(child.shutters())
            }
            return shutters;
        }
    }

    cclass Room {
        List<Shutter> shutters;
        List<Shutter> shutters() { return shutters; }
    }
}
```

Figure 6.4 Extension of house structure with shutter devices.

```
abstract cclass IShutterDevices { ... }
cclass ShutterEmulator extends IShutterDevices { ... }

abstract cclass IHouseShutters extends IHouseLocations requires IShutterDevices {
    abstract cclass Location {
        abstract List<Shutter> shutters();
    }

    abstract cclass CompositeLocation {
        provided List<Shutter> shutters();
    }
}

cclass HouseShutters extends IHouseShutters requires IHouseStructure {
    abstract cclass CompositeLocation {
        List<Shutter> shutters() {
            ... locations() ...
        }
    }

    cclass Room {
        List<Shutter> shutters;
        List<Shutter> shutters() { return shutters; }
    }
}
```

Figure 6.9 Explicit interfaces for HouseShutters.

Taken from: *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*

CaesarJ

- Aspect oriented programming language based on Java
- Developed in Darmstadt and available in http://www.caesarj.org/

Composition Mechanism in CaesarJ

Binding

-such as intertype declaration in AspectJ

-between two classes

-one is wrapper and second one is wrappee

-keyword **wraps**

Composition of Virtual Classes

-such as inheritance of virtual classes

-composed using & operator

```
cclass Composition extends
        Layer3 & Layer2 & Layer1 { }
```

Realization from right to left

<higher> → <lower> **Priority**

Pointcuts and Advices

-in Caesar classes

```
public cclass AnAspect {
    public pointcut APointcut(): .... ;
    public around() : APointcut() {
        // code to insert
    }
}
```

Representing Grafs using CaesarJ

```
public cclass Graph {
	public cclass Edge { Node start, end; }
	public cclass UEdge extends Edge {…}
	public cclass Node {…}
}
```

```
public cclass WeightedGraph extends Graph {
	public cclass Edge {
		float cost;
	}
	public cclass Node {
		float cost;
	}
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Preparing for Electrical Model Scheme Specialization

```
public class Wire {
    public Pin getStartPin() { … }
    public Pin getEndPin() { … }
    …
}
```

```
public class Component {
    public int getPinCount() { … }
    public Pin getPinByNr(int nr) { … }
    …
}
```

```
public class Pin {
    public Component getOwner() { … }
    public Iterator getAttachedWires() { … }
    …
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Graph Representation Using Caesar Classes

```
public cclass Graph {
    . . .
    public cclass Edge {
        public Node getStartNode() { … }
        public Node getEndNode() { … }
    }
    public cclass Node {
        public Iterator getEdges() { … }
    }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Binding Creation

```
cclass ElectricGraph extends Graph {
    cclass Edge wraps Wire { … }
    cclass Node wraps Component { … }
}
```

WRAPPER WRAPPEE

```
cclass ElectricGraph extends Graph {
```

WRAPPING/BINDING

```
    cclass Edge wraps Wire {
        public Node getStartNode() {
            return Node(wrappee.getStartPin().getOwner());
        }
    …
    }
    …
}
```

Returnins the wrapper of Pin (Node)

ACCESSING THE INSTANCE OF THE WRAPPEE CLASS (WIRE)

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Creating Instances of Wrapped Classes

Expression returning CeasarJ class object where is wrapper defined (**this** in this case)

wrapper class

(Node)

Reference to subtype of wrapper class

(wrappee.getStartPin()
.getOwner())

this.Node(**wrappee**.getStartPin().getOwner())

Node(**wrappee**.getStartPin().getOwner())

Composition of Virtual Classes

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

Realization from right to left

<higher> → <lower> **Priority**

COMPOSITION STEP 1:

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

(**Layer2** **&** Layer1)

overwrites

COMPOSITION STEP 2:

cclass Composition **extends**
Layer3 **&** Layer2 **&** Layer1 { }

cclass Composition **extends**
[**Layer3** **&** (Layer2 **&** Layer1)] { }

overwrites

Cuckoo's Egg In CaesarJ

Symmetrically using Composition

EMPTY IMPLEMENTATION OF GENERAL CAESAR CLASS:

```
public cclass GeneralEgg {
    public cclass Egg { }
}
```

```
public cclass UncompromisedNest extends GeneralEgg {
    public cclass Egg {
        public void exec() {
            System.out.println(”a real egg.”);
        }
    }
    public cclass Nest {
        public Nest() { }
        public void exec() {
            System.out.print(”The nest contains ”);
            new Egg().exec();
        }
    }
    public void app() {
        new Nest().exec();
    }
}
```

Overiden method during composition

Overiden method during composition

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Composition Mechanism:

```
public cclass CuckoosNest
        extends CuckoosEgg & UncompromisedNest { }
```

Overwrites UncompromisedNest during composition

Cuckoo's Egg:

```
public cclass CuckoosEgg extends GeneralEgg {
    public cclass Egg {
        public void exec() {
            System.out.println(”a cuckoo’s egg.”);
        }
    };
}
```

PRECEDENCE TO THIS METHOD during overiding

Eggs instantiation:

```
public class Main {
    public static void main(String[] s) {
        new UncompromisedNest().app();
        new CuckoosNest().app();
    }
}
```

The nest contains a real egg.

The nest contains a cuckoo’s egg.

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Abstract Observer Pattern in CaesarJ

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

Declaring Subject and Observer roles

-with all their supported methods

```
abstract public cclass ObserverProtocol {
    abstract public cclass Subject {
        abstract public void addObserver(Observer o);
        abstract public void removeObserver(Observer o);
        abstract public void changed();
        abstract public String getState();
    }
    abstract public cclass Observer {
        abstract public void notify(Subject s);
    }
}
```

Observer in CaesarJ

```
public cclass ObserverProtocolImpl extends ObserverProtocol {
    abstract public cclass Subject {
        private List observers = new LinkedList();
        public void addObserver(Observer o) {
            observers.add(o);
        }
        public void removeObserver(Observer o) {
            observers.remove(o);
        }
        public void changed() {
            Iterator it = observers.iterator();
            while (it.hasNext()) { ((Observer) it.next()).notify(this); }
        }
    }
}
```

Adding observer

Implementing abstract subject role

Removing observer

Notifying observers when state changes

List of observers

Specializing Observer in CaesarJ

```
abstract public cclass ColorObserver extends ObserverProtocol {
    public cclass PointSubject extends Subject wraps Point {
        public String getState() {
            return ”Point colored ” + wrappee.getColor();
        }
    }
    public cclass LineSubject extends Subject wraps Line {
        public String getState() {
            return ”Line colored ” + wrappee.getColor();
        }
    }
    public cclass ScreenObserver extends Observer wraps Screen {
        public void notify(Subject s) {
            wrappee.display(”Color changed: ” + s.getState());
        }
    }
}
```

Binding **Subject Role** with its Implementation on **Line** and **Point**

Binding **Observer Role** with its Implementation on **Screen**

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

```
    after(Point p): (call(void Point.setColor(Color)) && target(p)) {
        PointSubject(p).changed();
    }
    after(Line l): (call(void Line.setColor(Color)) && target(l)) {
        LineSubject(l).changed();
    }
}
```

Implementing change when color is set

Defining observed event + its processing

Instantiation:

```
public cclass Test {
    private deployed static final CO co = new ColorObserverImpl();
    public void test() {
        Line l = new Line();
        Point p = new Point();
        Screen s1 = new Screen();
        Screen s2 = new Screen();
        co.LineSubject(l).addObserver(co.ScreenObserver(s1));
        co.PointSubject(p).addObserver(co.ScreenObserver(s1));
        co.LineSubject(l).addObserver(co.ScreenObserver(s2));
        l.setColor(new Color("red"));
        p.setColor(new Color("blue"));
    }
}
```

Composition:

```
public cclass ColorObserverImpl extends
    ObserverProtocolImpl & ColorObserver { }
```

Abstract pattern implementation + **Specific implementation /application**

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf

References

- **Product Line Implementation with ECaesarJ:** *Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (09 2011)*
- **Description and application of JaSCo:** http://ssel.vub.ac.be/jasco/
- **Description and application of CaesarJ:** https://caesarj.org/
- **Capabilities, Basics, Advanced topics and Application of AspectJ:** The AspectJ in Action *Laddad, Ramnivas, 2003. AspectJ in action: practical aspect-oriented programming. Greenwich, CT: Manning. ISBN 978-1-930110-93-9.*
- ***Component and composite approach notes:*** http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/komp.pdf