Aspect Oriented Software Development

Carlos Noguera

2012-2013

Introduction to AOSD

Software Complexity

Functional Requirements

Non-functional requirements

Software Development Requirements

Software Complexity

Functional Requirements

+

Non-functional requirements

+

Software Development Requirements

COMPLEXITY

Accidental vs. Essential

[F.P. Brooks]

COMPLEXITY

Essential

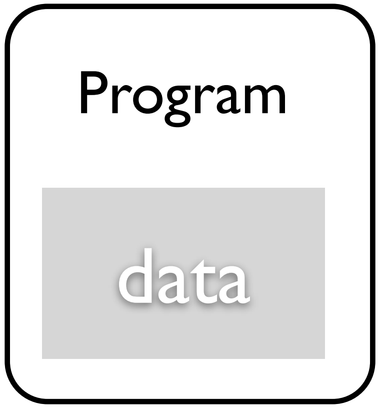
- Irreducible
- The problem is hard.

Accidental

- Reducible
- The tools/approach is bad

Need adequate software engineering techniques

(Quick) Evolution of Software Programming




```
----------------------------------------
* FUNCTION: INCH - Input character
* INPUT: none
* OUTPUT: char in acc A
* DESTROYS: acc A
* CALLS: none
* DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04   INCH    LDA A   ACIA      GET STATUS
C013 47                 ASR A             SHIFT RDRF FLAG INTO CA
C014 24 FA              BCC     INCH      RECIEVE NOT READY
C016 B6 80 05           LDA A   ACIA+1    GET CHAR
C019 84 7F              AND A   #$7F      MASK PARITY
C01B 7E C0 79           JMP     OUTCH     ECHO & RTS
```

Difficult to read/write

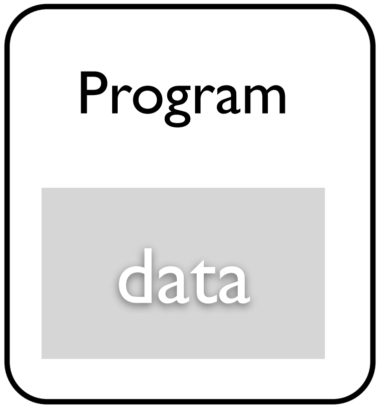
Poor evolvability

Poor maintainablity

Poor reusability

Machine-Level Programming

(Quick) Evolution of Software Programming



Easier to read/ write

Poor evolvability

Poor maintainablity

Poor reusability

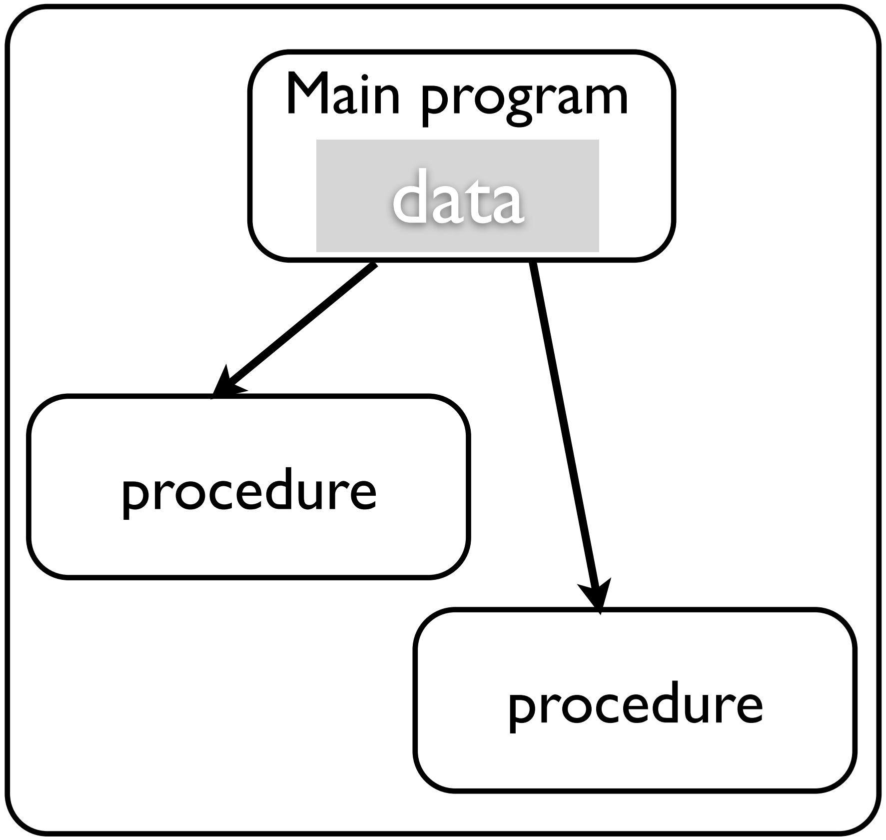
\+ Language features for common patterns

```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```

Structured Programming

(Quick) Evolution of Software Programming

- + Procedural Abstraction
- + Parameter Passing
- + Recursion



Easier to read/ write

Better evolvability

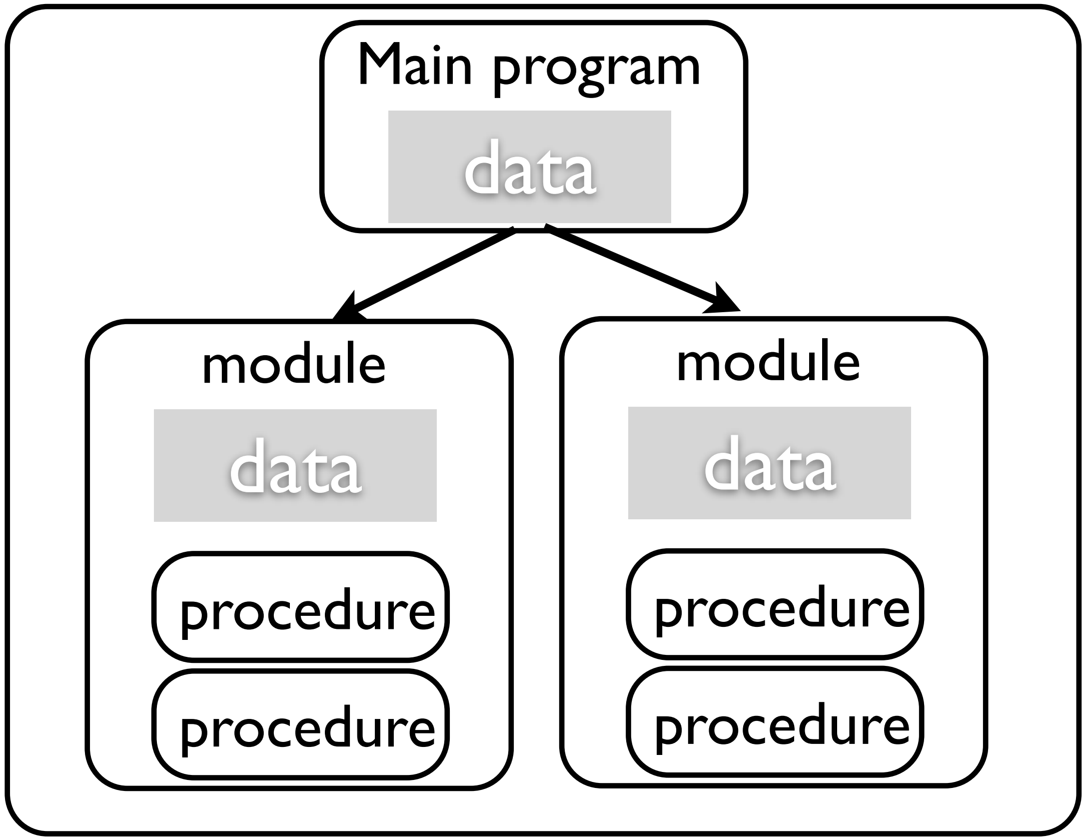
Better maintainablity

Better reusability

Procedural Programming

(Quick) Evolution of Software Programming

+ Modules



Easier to read/ write

Better evolvability

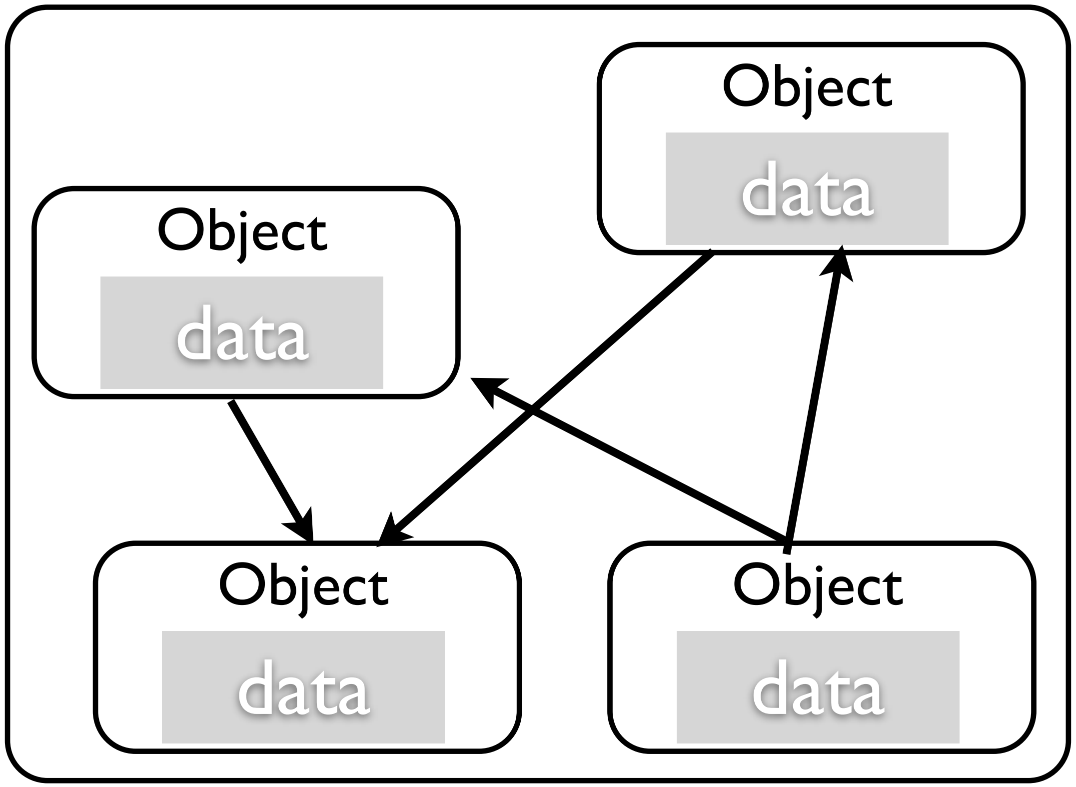
Better maintainablity

Better reusability

Modular Programming

(Quick) Evolution of Software Programming

+ Encapsulation
+ Polymorphism
+ Inheritance



Easier to read/write

Better evolvability

Better maintainablity

Better reusability

Object-Oriented Programming

(Quick) Evolution of Software Programming

Aspect Oriented Programming

Components

Model-Driven Engineering

Magic?

On keeping things Separate

So, what's the problem?

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to **study in depth an aspect of one's subject matter in isolation** for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.

On keeping things Separate

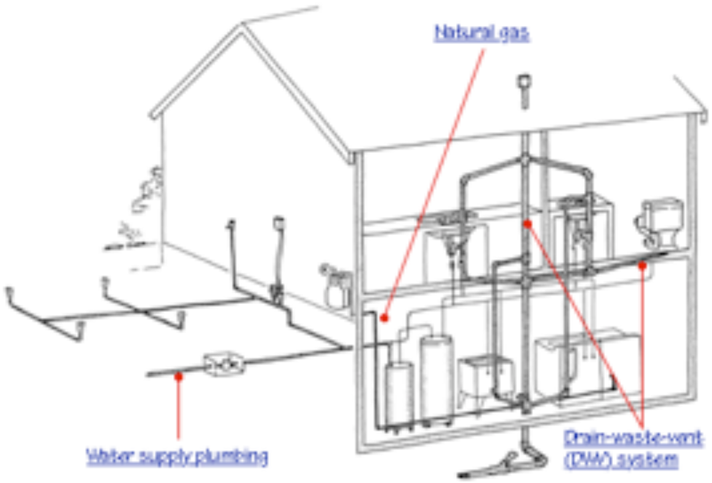
So, what's the problem?

We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called **"the separation of concerns"**

[E.W. Dijkstra]

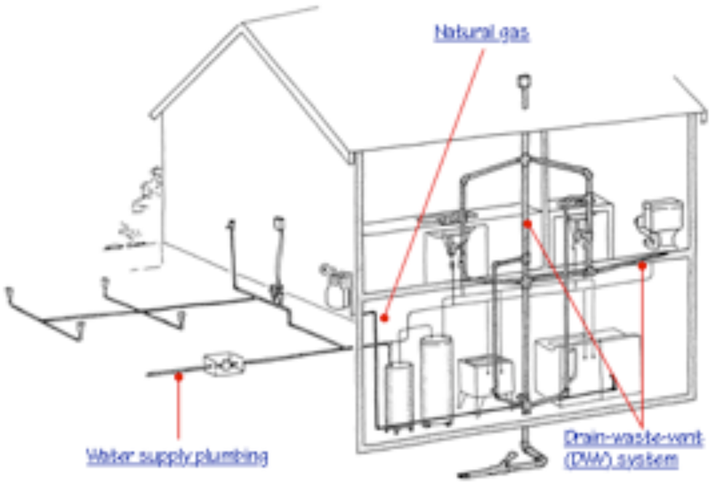
Separation of Concerns

Separation of Concerns

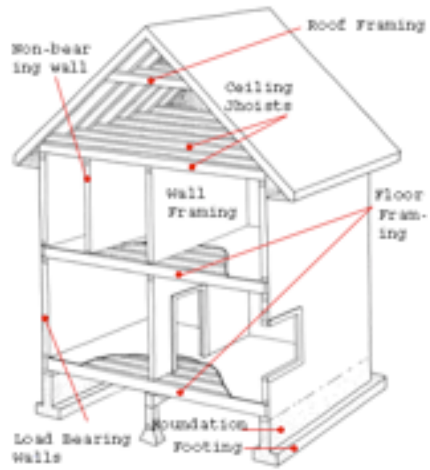


Piping

Separation of Concerns



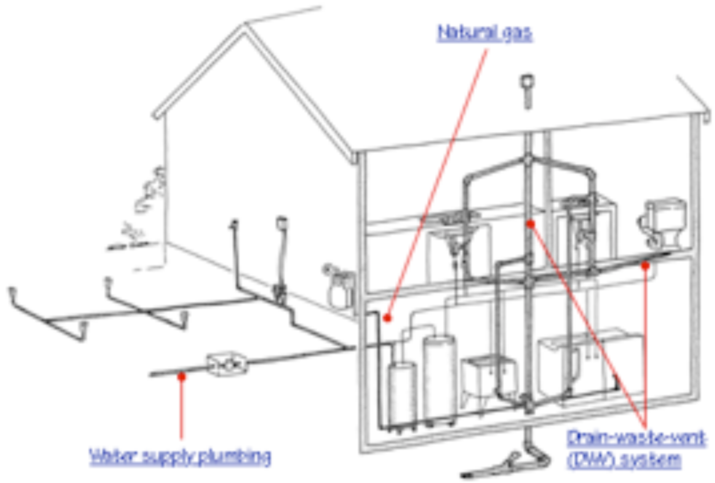
Piping



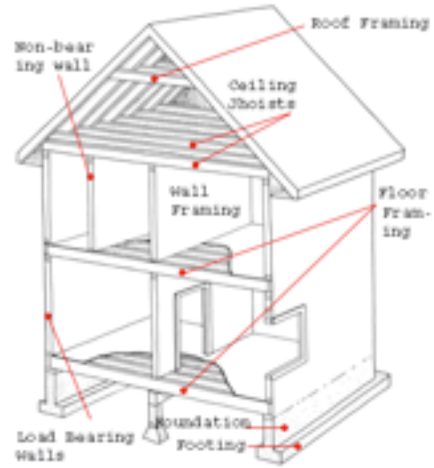
Structural

Separation of Concerns

Electrical



Piping



Structural

Separation of Concerns

con·cern /kən'sərn/

noun- Something the developer needs to care about

Separation of- to handle each concern in isolation

→ Separation of Concerns drives evolution of programming languages and paradigms

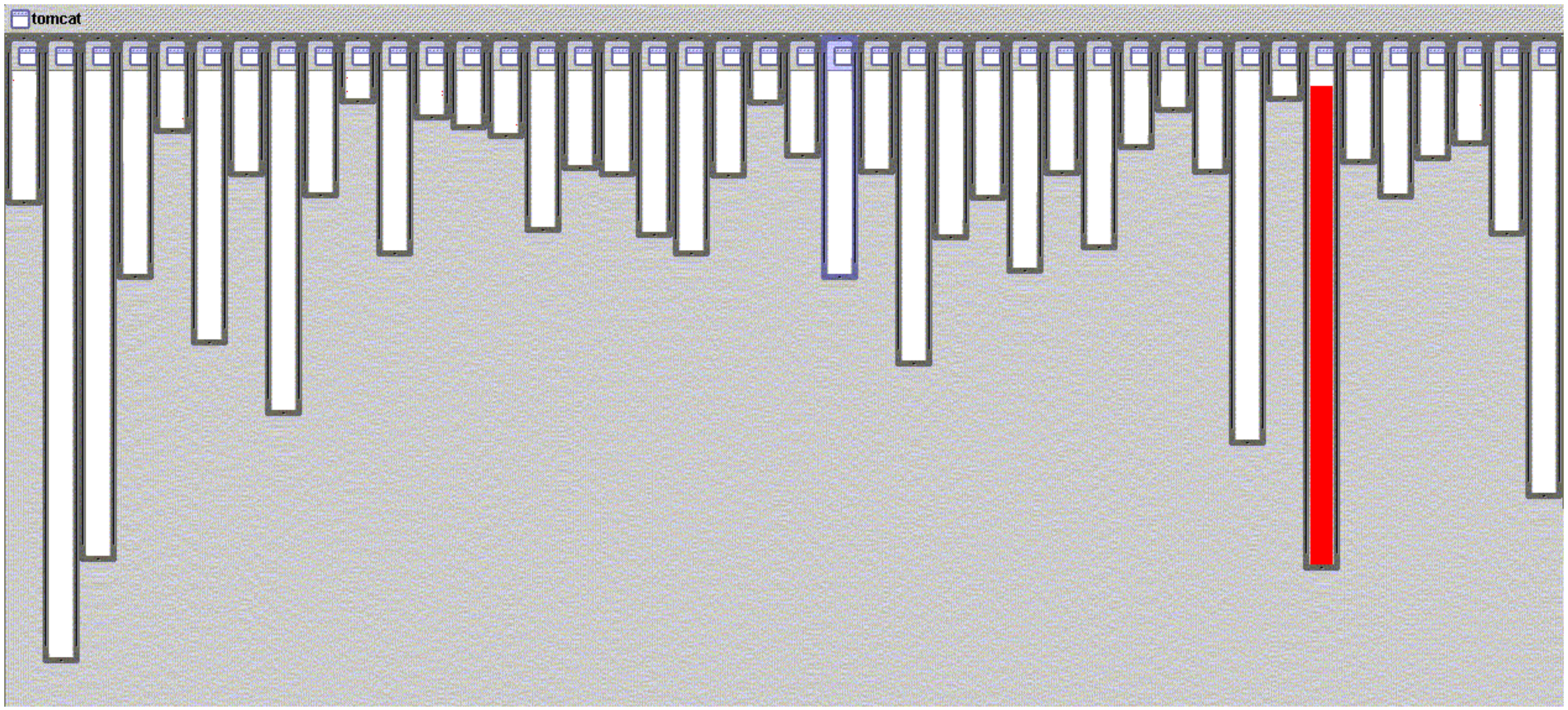
e.g., Modular programming groups code by data and functionality

SoC allows you to

- Reduce complexity
- Promote traceability across artifacts
- Limit the impact of change
- Facilitate reuse
- Simplify integration

What's the problem again?

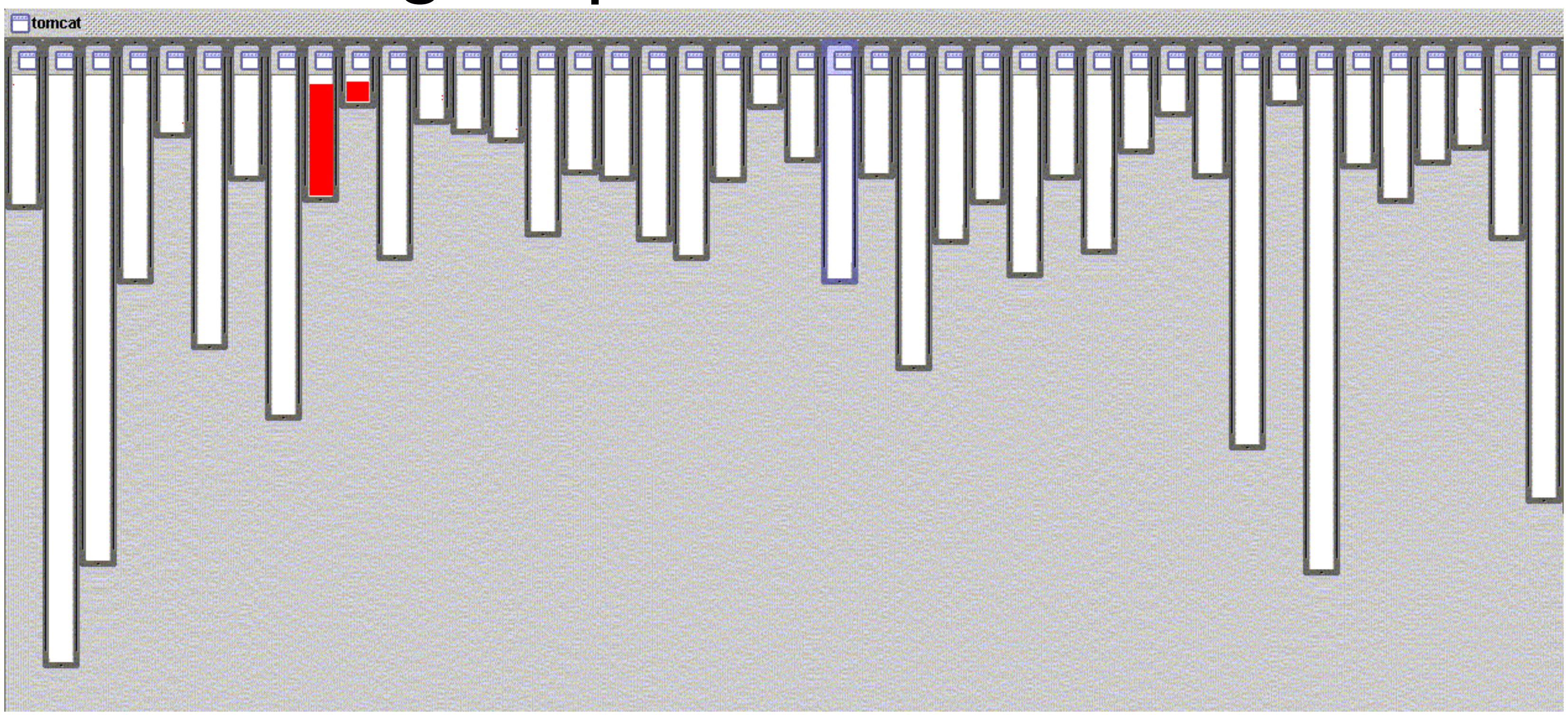
XML Parsing in Apache Tomcat



aspectj.org website

What's the problem again?

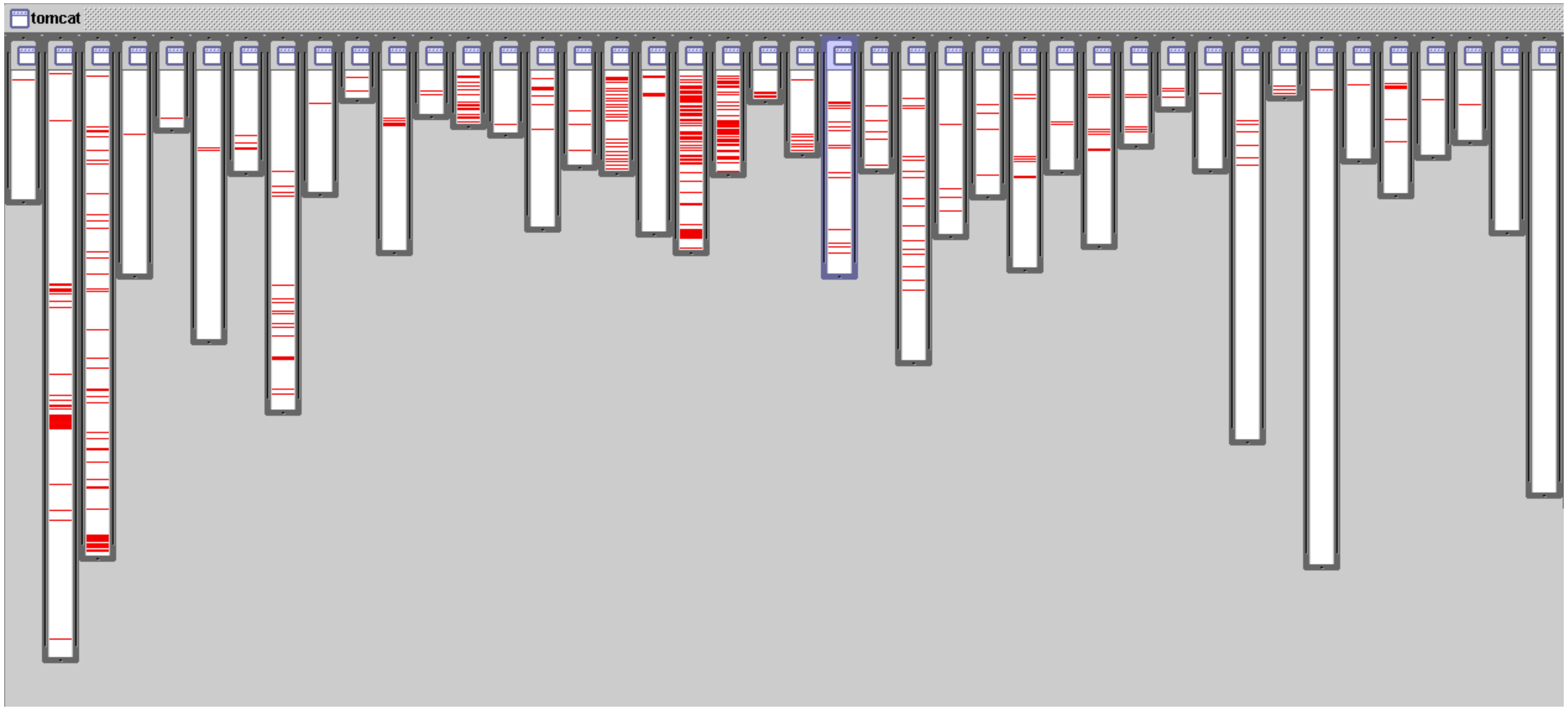
URL handling in Apache Tomcat



aspectj.org website

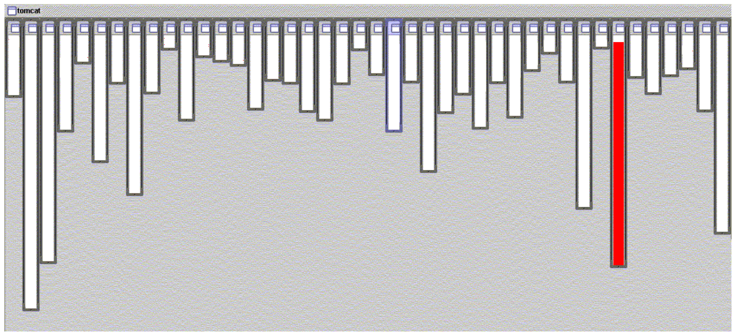
What's the problem again?

Logging in Apache Tomcat

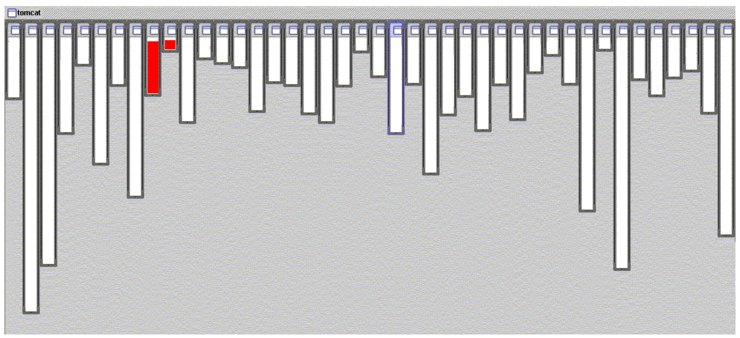


aspectj.org website

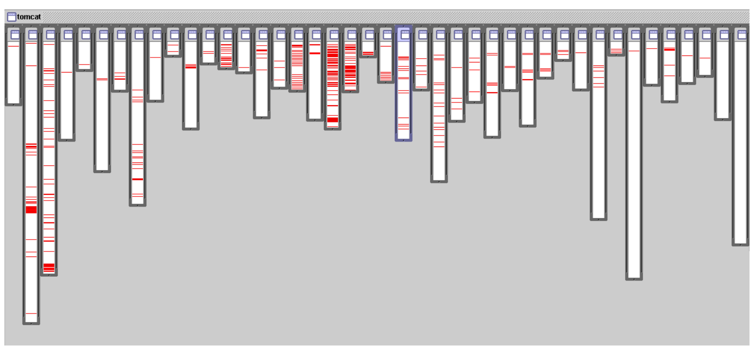
Crosscutting Concerns



XML Parsing
- Good SoC
- One class

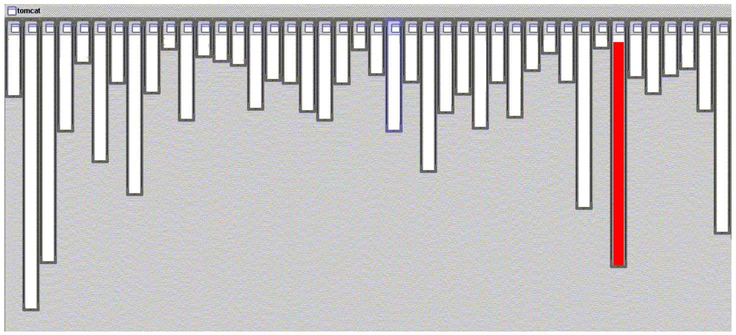


URL Handling
- Good SoC
- Two classes, related by inheritance

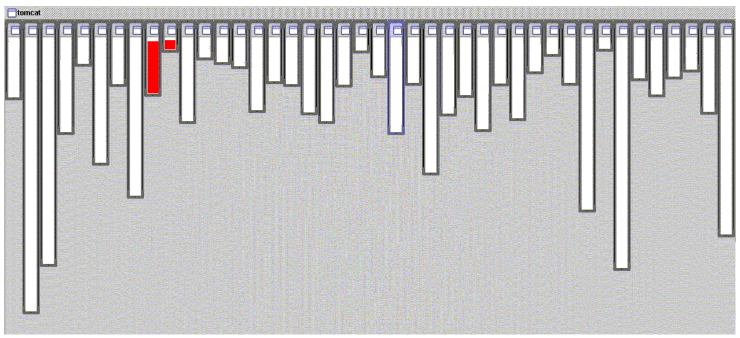


Logging
- Bad SoC
- Everywhere

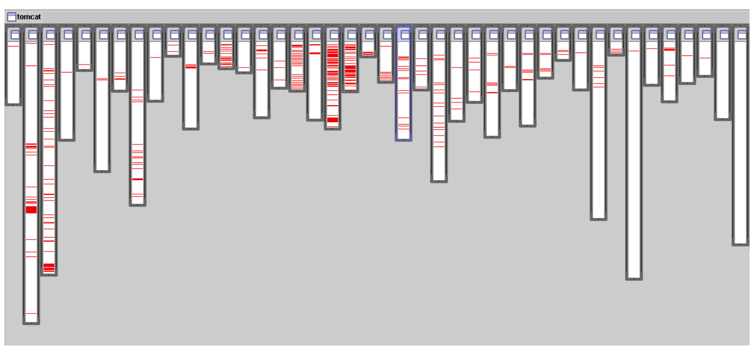
Crosscutting Concerns



XML Parsing
- Good SoC
- One class



URL Handling
- Good SoC
- Two classes, related by inheritance



Logging
- Bad SoC
- Everywhere

CROSSCUTTING!

Crosscutting Concerns

Scattering

Code addressing the concern in several places

Tangling

Code in one region addresses several concerns

Scattering and Tangling are symptoms of the same problem

The problem with Scattering and Tangling

Scattering and tangling results in code that is

- Redundant
- Hard to reason about
- Difficult to change

Where CCC?

- Logging
- Caching
- Security
    - Access control
    - Confidentiality
- Transactions
- Persistence
- ....

Where CCC?

- Logging
- Caching
- Security
  - Access control
  - Confidentiality
- Transactions
- Persistence
- ....

```
public void boe(String s, Key k) {
    log("entering method boe with arguments … ");
    …
    …
    log("exiting method boe");
}
```

Where CCC?

- Logging
- Caching
- Security
  - Access control
  - Confidentiality
- Transactions
- Persistence
- ....

```
public String compute(Object input) {
   Object[] args = new Object[] {input};
   Object res = cache.fetch(“myclz.compute”,args);
   if(res!=null) //cache contains value
      return (String) res;

   …
   //store result into computedResult
   cache.store(“myclz.compute”,args,computedResult);
   return computedResult;
}
```

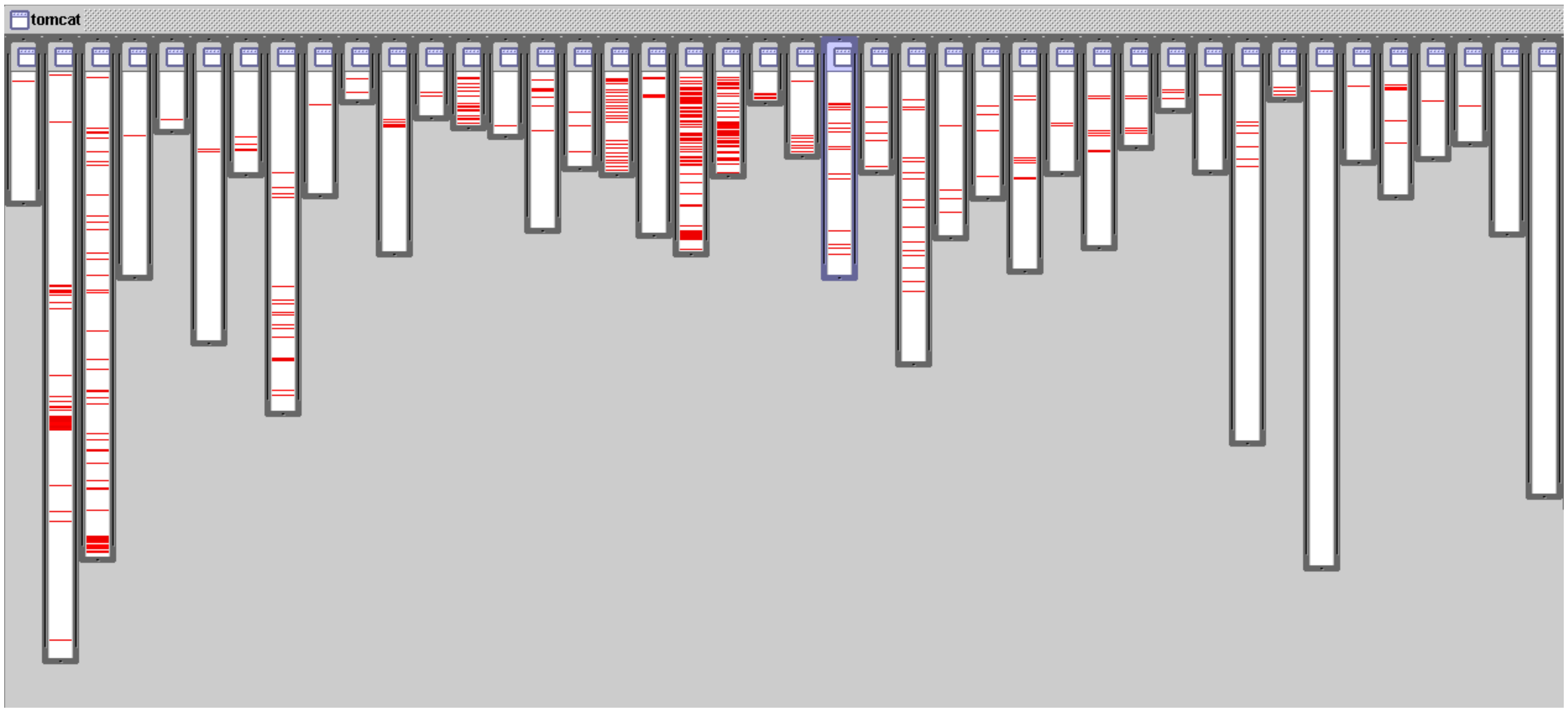
Where CCC?

- Logging
- Caching
- Security
  - Access control
  - Confidentiality
- Transactions
- Persistence
- ....

```
public void transactional(Object input) {
    Transaction t =
      transactionManager.startTransaction();
    try{
        ...

     }catch(Throwable t) {
       transactionManager.rollback(t);
     }
     transactionManager.commit(t);
}
```

Modularizing CCCs

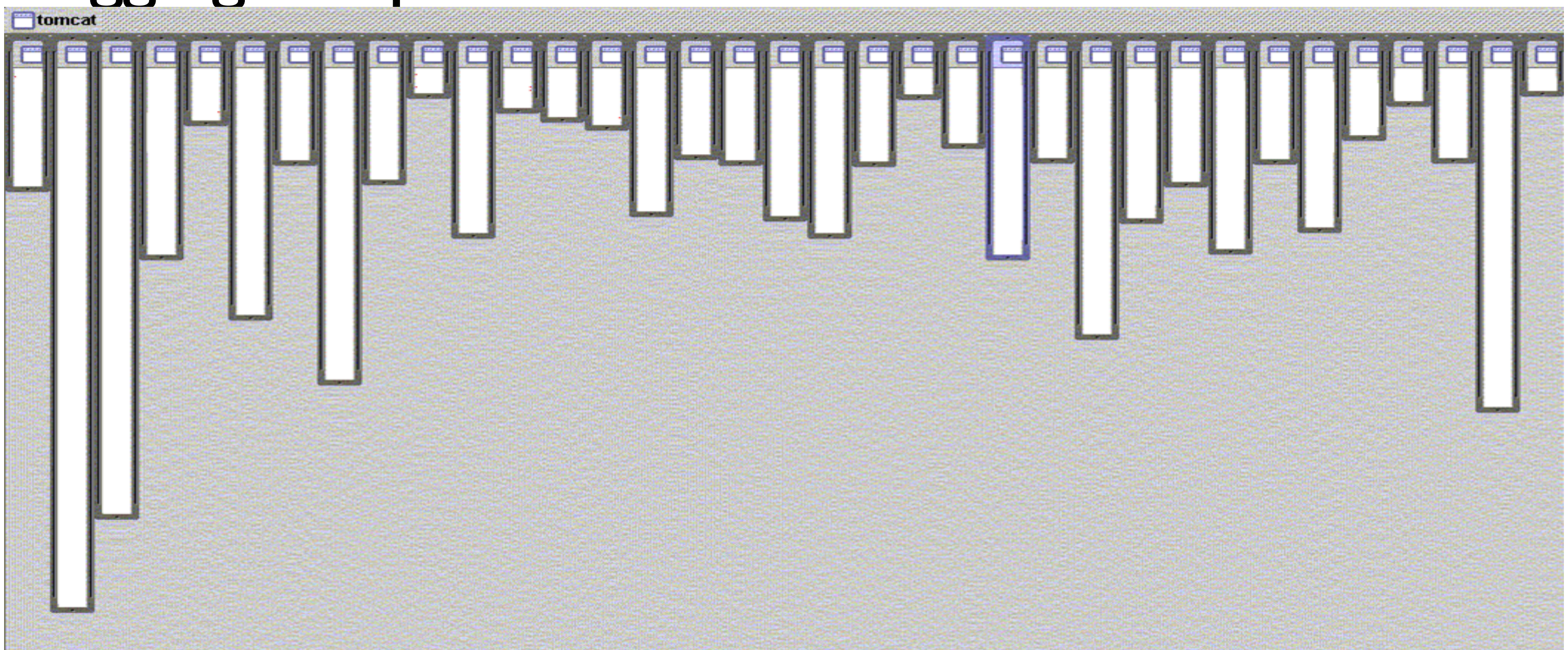
Logging in Apache Tomcat



aspectj.org website

Modularizing CCCs

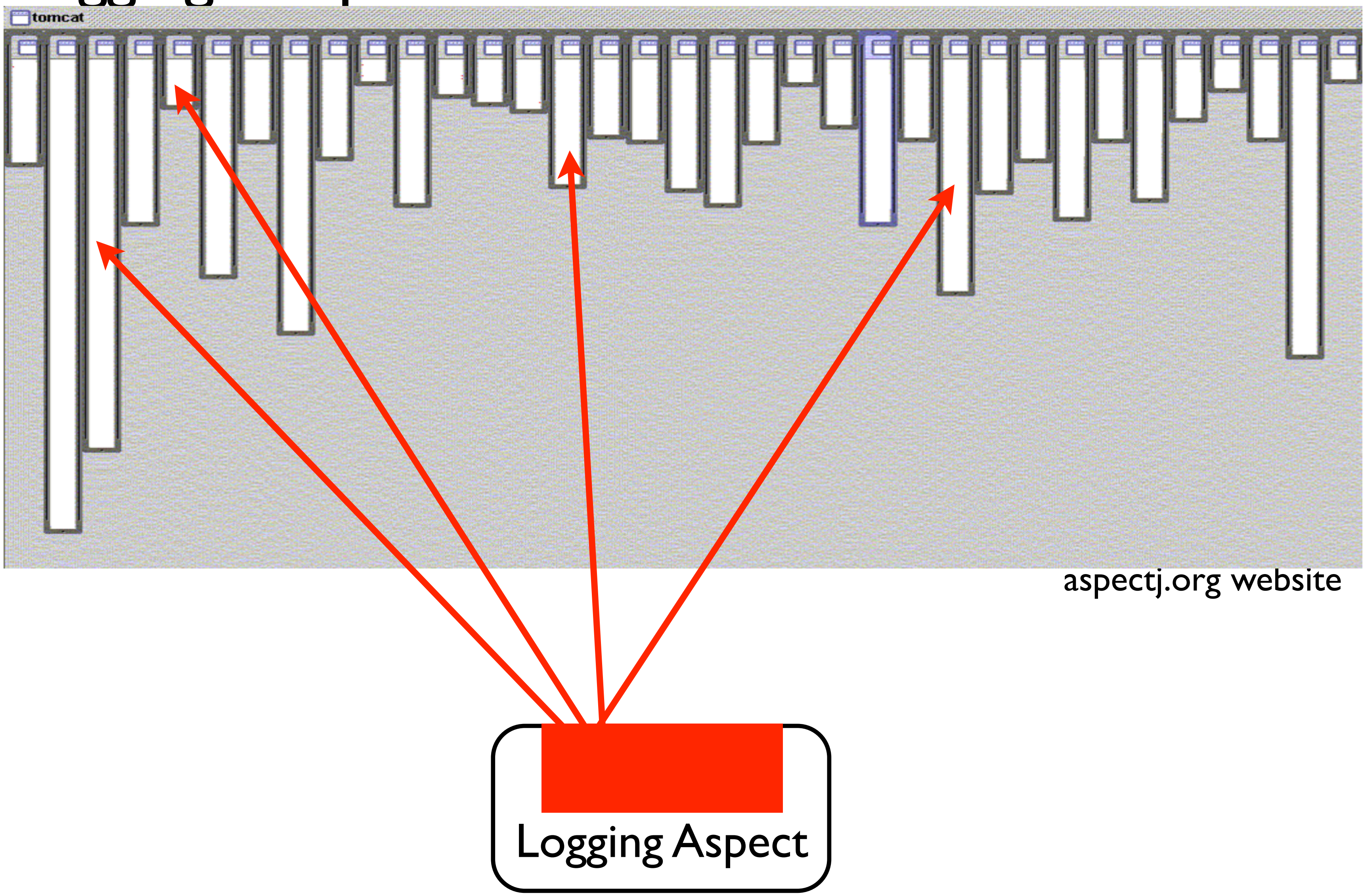
Logging in Apache Tomcat



aspectj.org website

Modularizing CCCs

Logging in Apache Tomcat



AOSD

- Crosscutting is inherent in complex systems

"The tyranny of *dominant decomposition*"

- CCCs have
  - a clear purpose
  - regular interaction points
- AOP captures CCCs
  - Modularization
  - Programming support
  - tool support

What

Where/When

Tyranny of Dominant Decomposition

Given one of multiple possible decompositions of the problem...

Tyranny of Dominant Decomposition

Given one of multiple possible decompositions of the problem...

Then, some subproblems cannot be easily modularized.

Tyranny of Dominant Decomposition

Given one of multiple possible decompositions of the problem...

Then, some subproblems cannot be easily modularized.

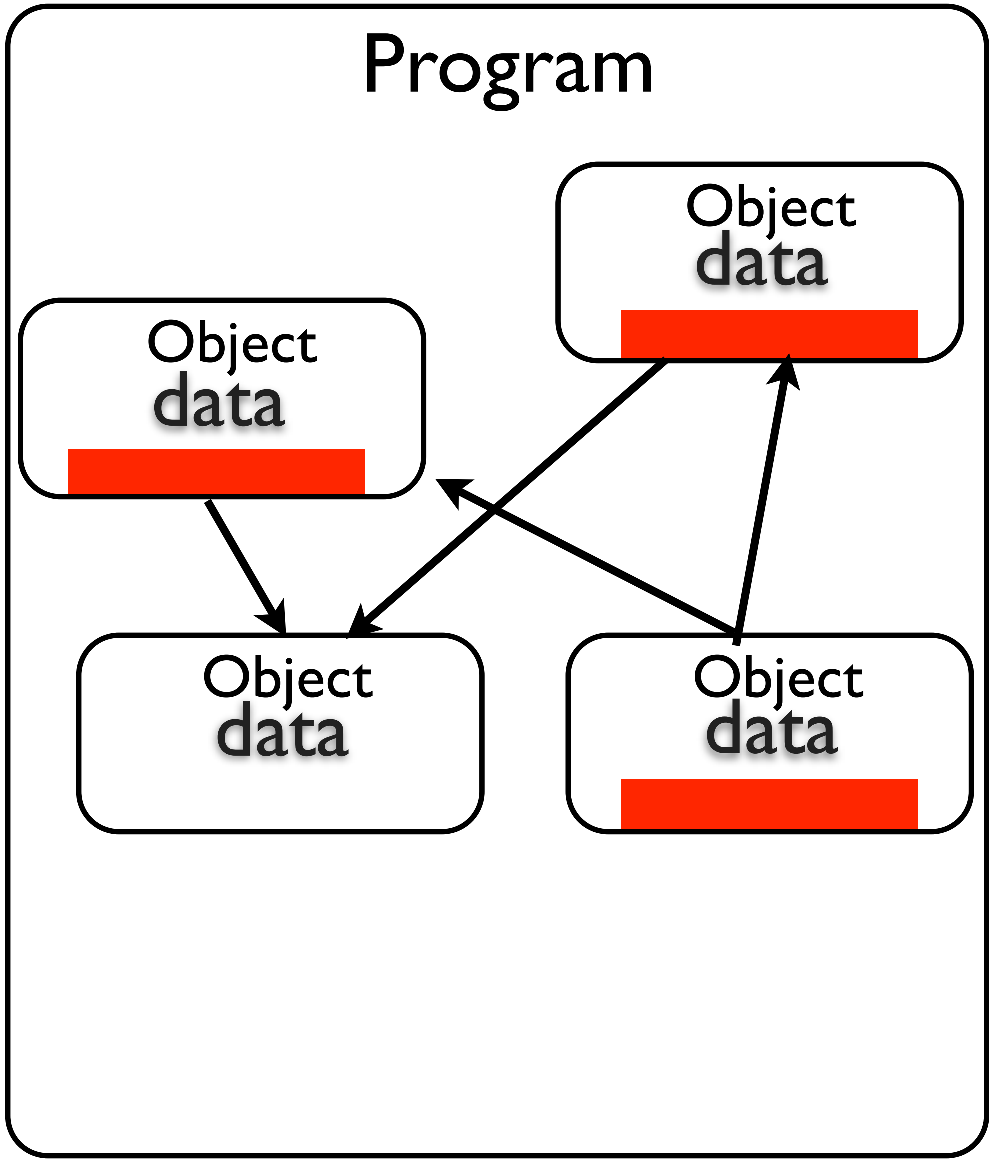
- True also for all possible decompositions
- True also for other paradigms than OO
- True also for analysis, design, etc....

Aspectual Decomposition

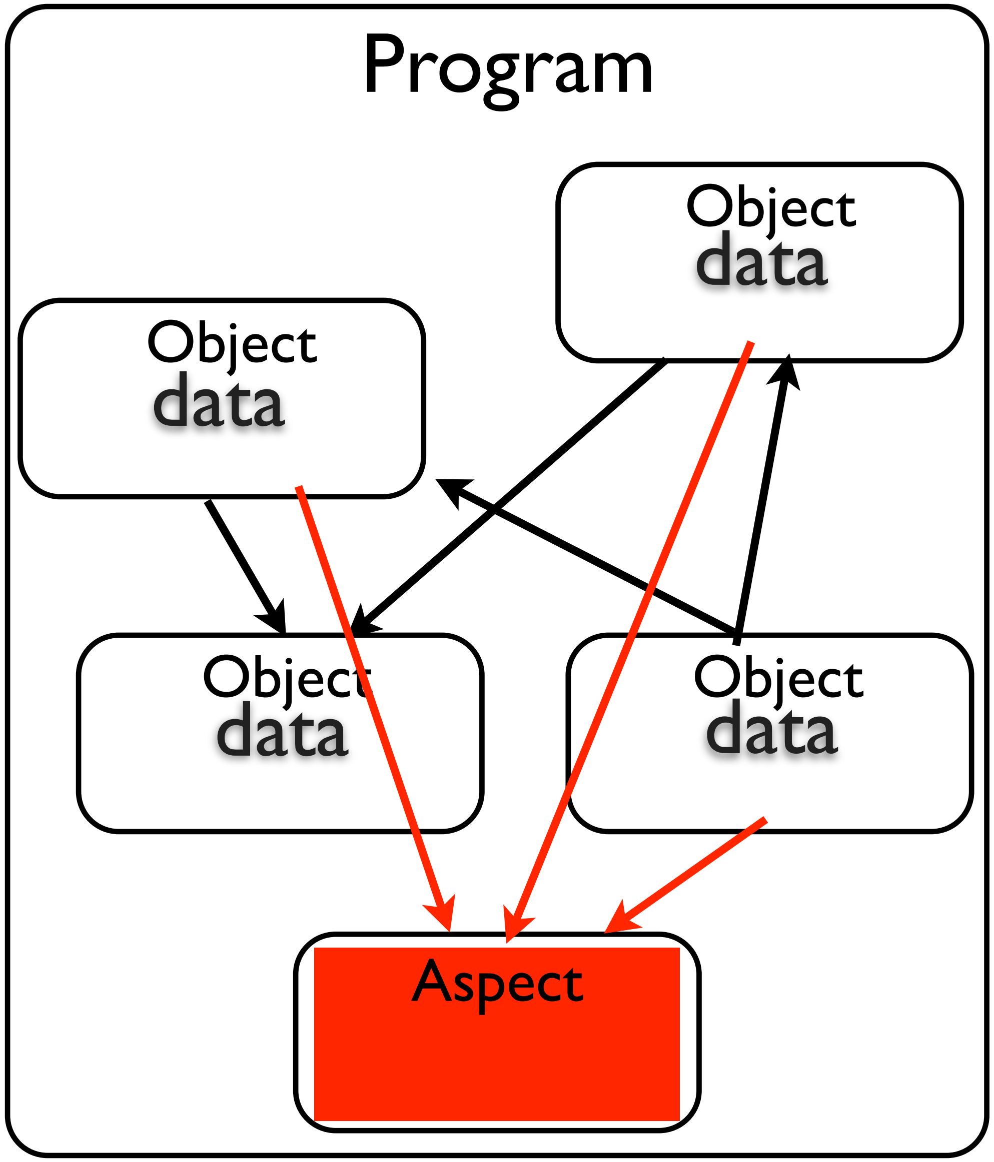
Many existing programming languages, including object-oriented languages, procedural languages and functional languages, can be seen as having a common root in that their key abstraction and composition mechanisms are all rooted in **some form of generalized procedure.**

[G. Kiczales]

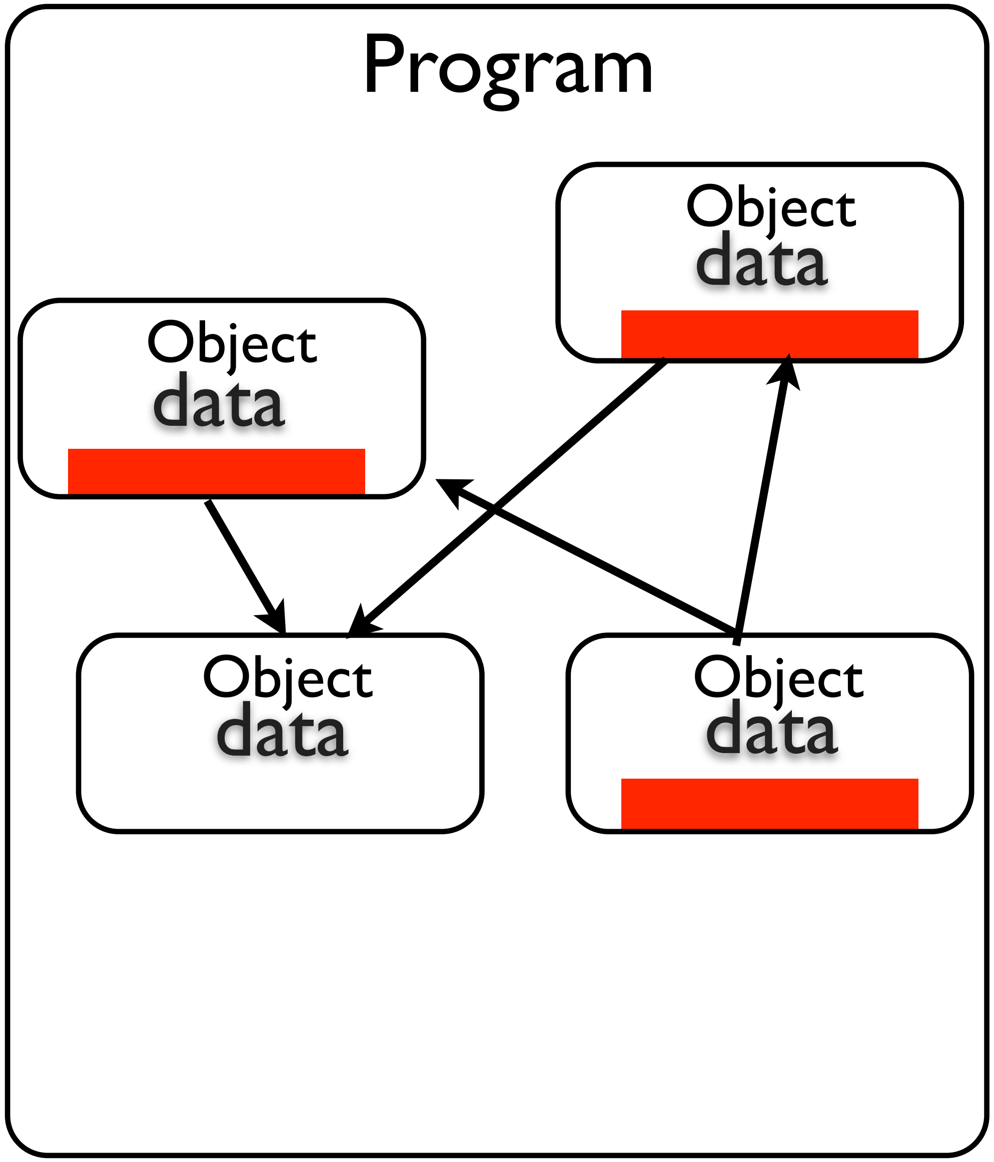
Explicit Invocation



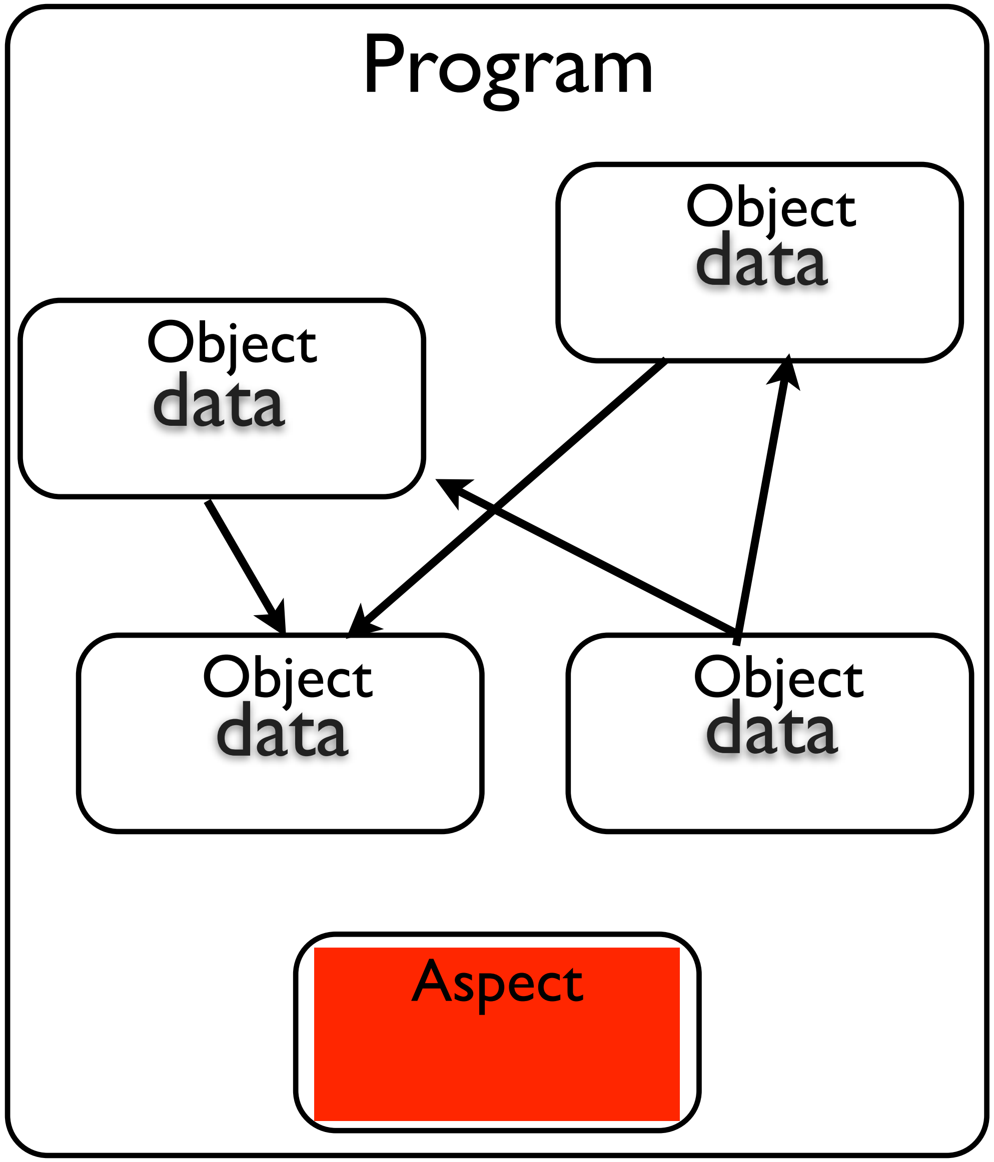
Explicit Invocation



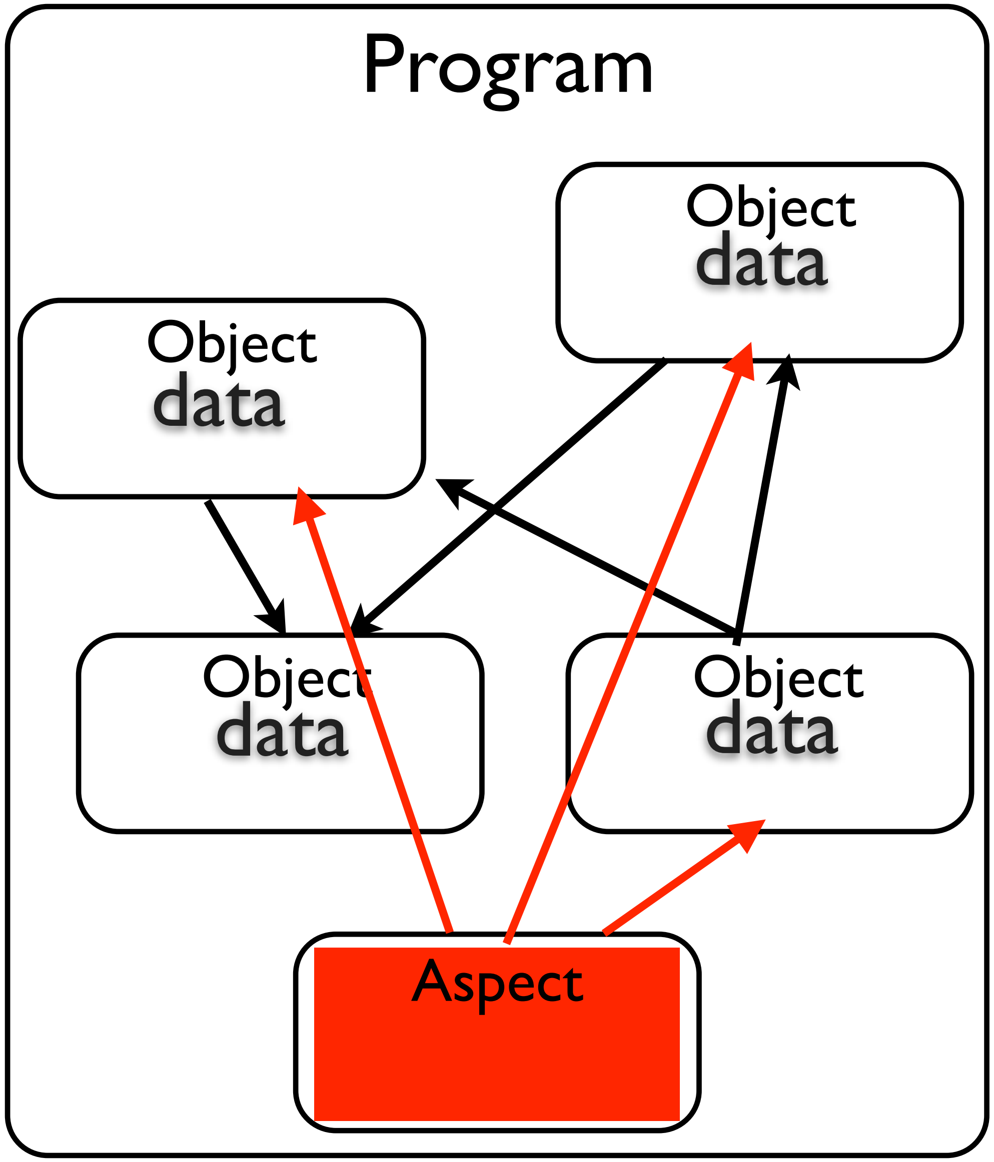
Implicit Invocation



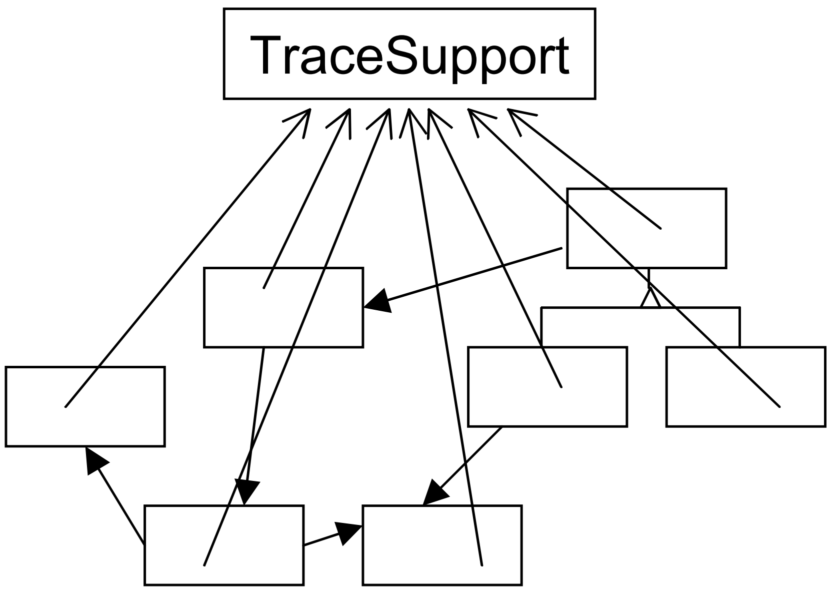
Implicit Invocation

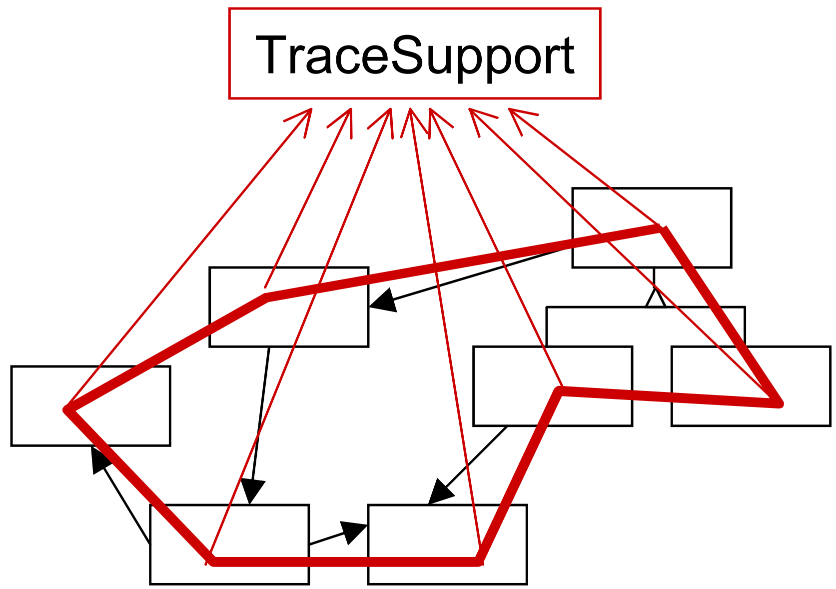


Implicit Invocation



Implicit invocation: how does it work?

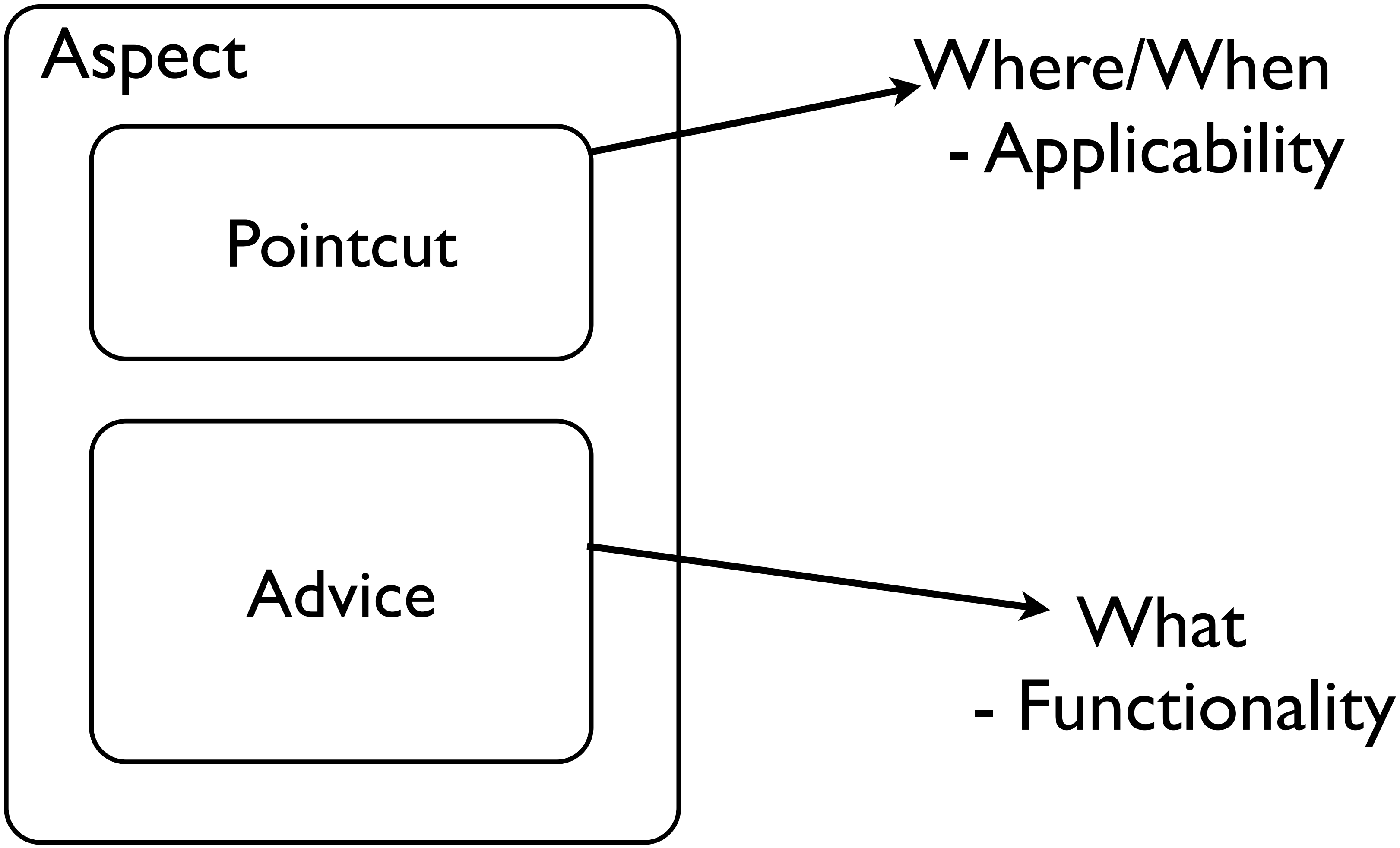




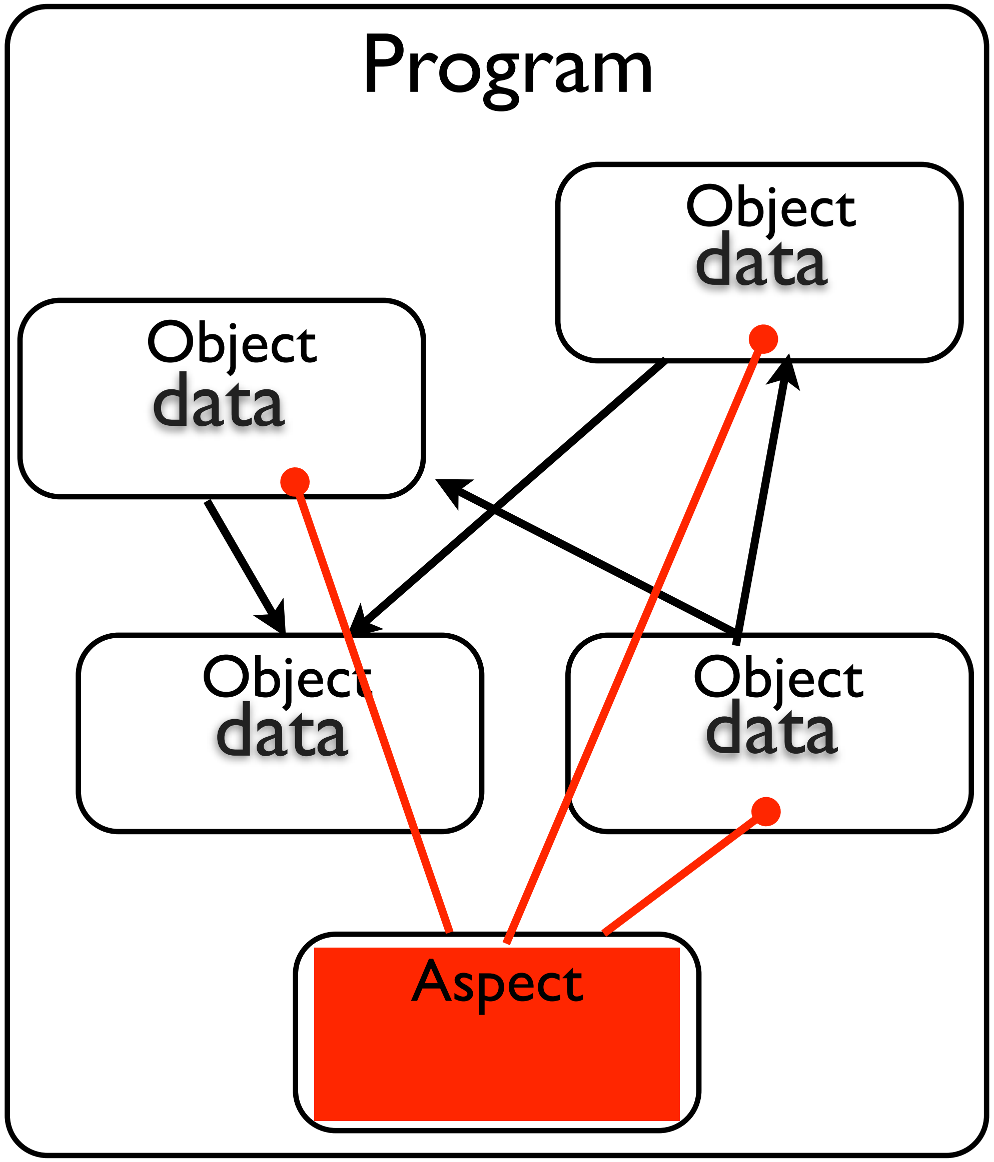
Objects invoke each other's methods

Aspects capture invocations that occur in other modules

Anatomy of Aspects



Joinpoints



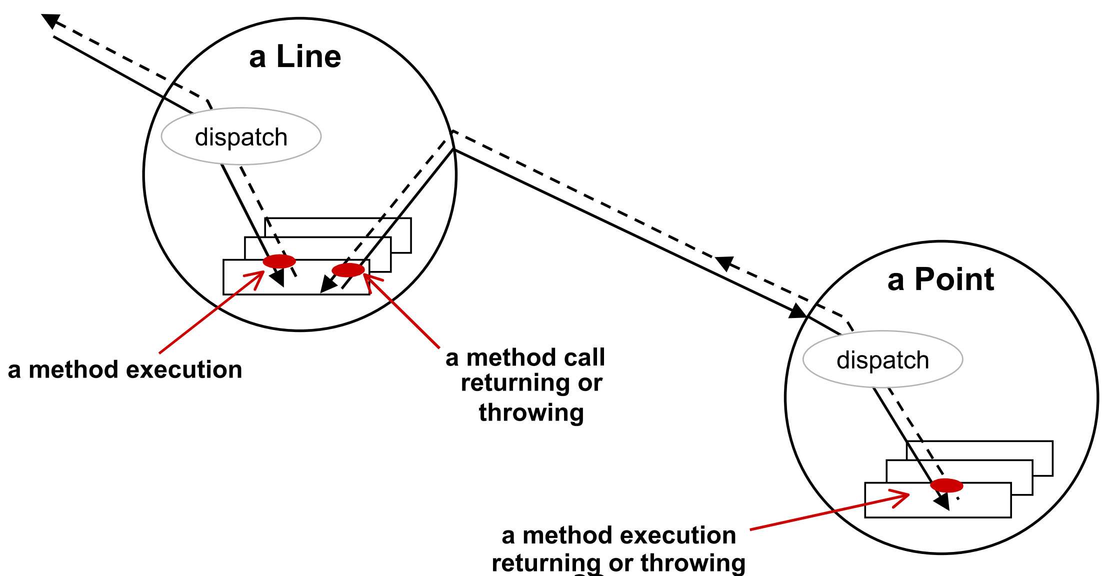
A joinpoint is a point of interest in the program where concerns may be composed

- message sends
- method execution
- error throwing
- instance creation
- ...

Joinpoint model

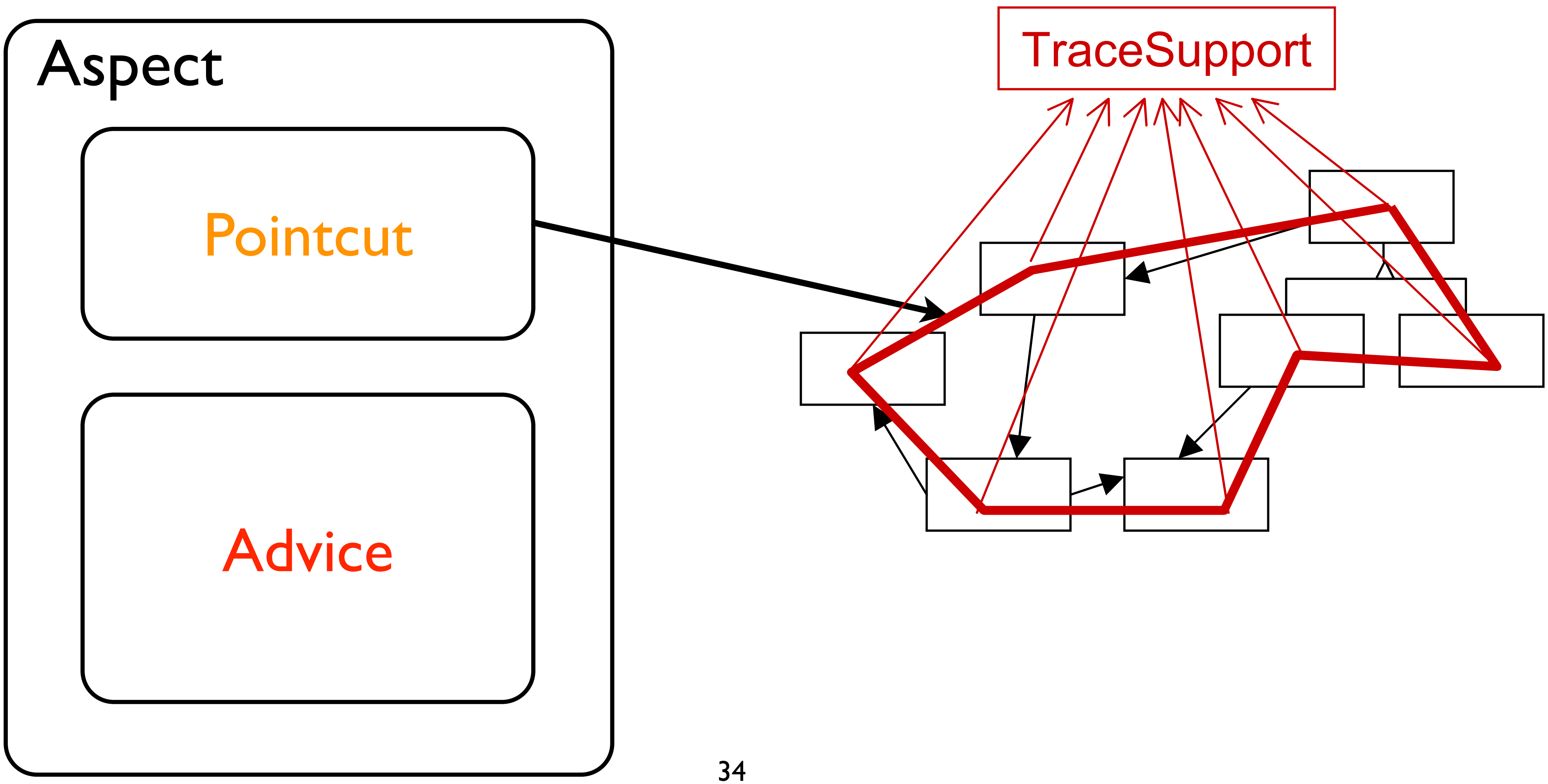
defines the kinds joinpoints available and how they are used

- Specific to aspect-oriented programming language
- e.g., key points in the dynamic call graph



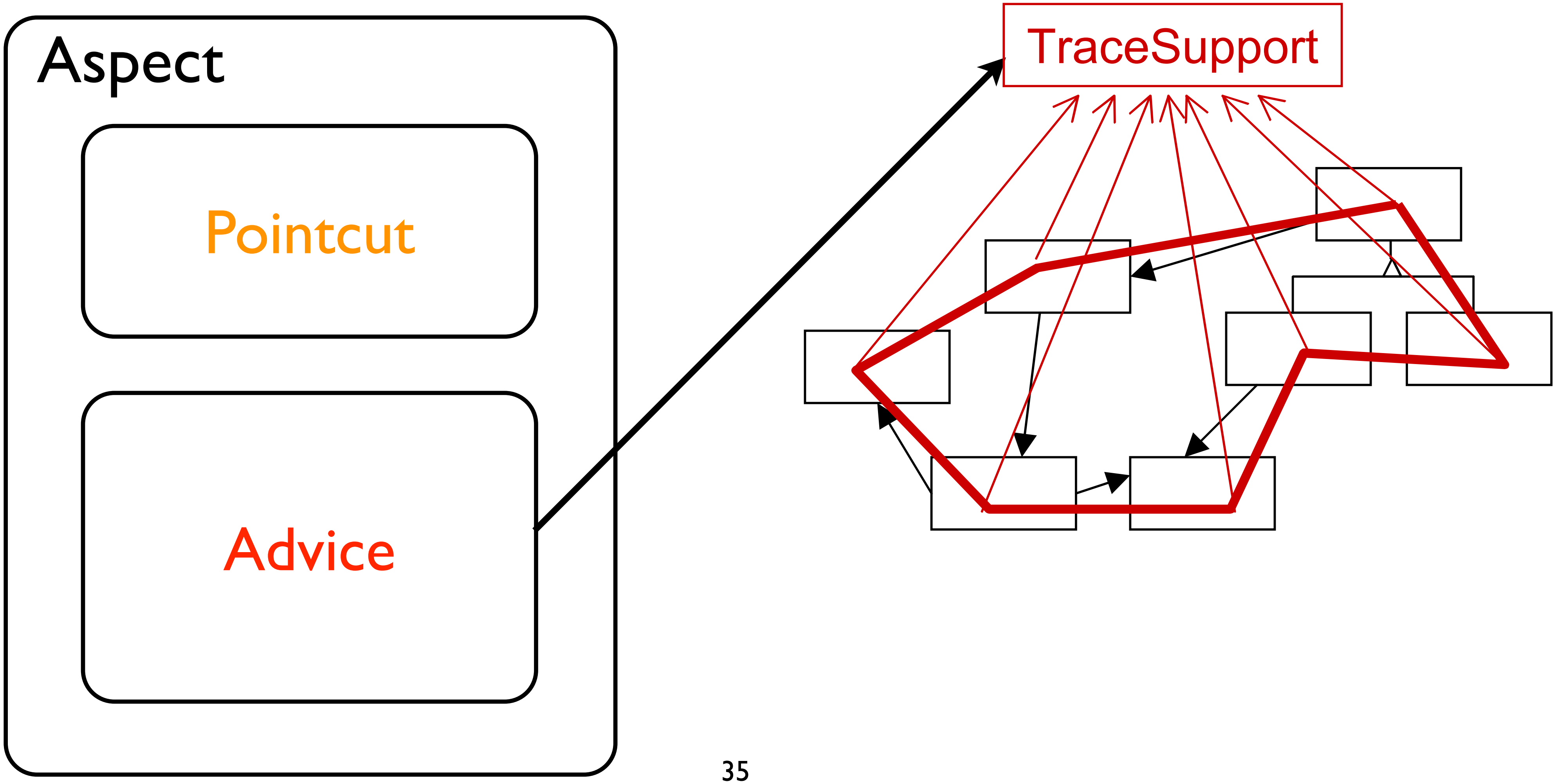
Pointcut

Predicate to select joinpoints.



Advice

behaviour to execute on the selected joinpoints



Example:
A synchronized Buffer

```
class Buffer {
  char[] data;
  int numElem;
  Semaphore sem;

  bool isEmpty() {
    bool retV;
    sem.writeLock();
    retV = numElem ==0;
    sem.unlock();
    return retV;
  }
}
```

Concerns:

- Buffer
- Synchronization

Example: A synchronized Buffer

```
class Buffer {
  char[] data;
  int numElem;
  Semaphore sem;

  bool isEmpty() {
    bool retV;
    sem.writeLock();
    retV = numElem ==0;
    sem.unlock();
    return retV;
  }
}
```

Concerns:

Buffer

Synchronization

Example:
A synchronized Buffer

```
class Buffer {
  char[] data;
  int numElem;
  Semaphore sem;

  bool isEmpty() {
    bool retV;
    sem.writeLock();
    retV = numElem ==0;
    sem.unlock();
    return retV;
  }
}
```

Concerns:

Buffer

Synchronization

Example: A synchronized Buffer

```
class Buffer {
  char[] data;
  int numElem;
  Semaphore sem;

  bool isEmpty() {
    bool retV;
    sem.writeLock();
    retV = numElem ==0;
    sem.unlock();
    return retV;
  }
}
```

CROSSCUTTING!

Concerns:

Buffer

Synchronization

Synchronization as an Aspect

When a Buffer object receives the message isEmpty() *first make sure that the object is not being accessed by another thread via the* get *or* set

Synchronization as an Aspect

When a Buffer object receives the message isEmpty() *first make sure that the object is not being accessed by another thread via the get or set*

Pointcut: when to execute the aspect

Advice: what to do at selected joinpoints

Kind of Advice: composition of when and what

Synchronization as an Aspect

```
class Buffer {
  char[] data;
  int numElem;

  bool isEmpty() {
    bool retV;
    retV = numElem ==0;
    return retV;
  }
}
```

```
aspect Synchronization
  Semaphore sem;

  before:execution(Buffer.isEmpty())
  {
    sem.writeLock();
  }

  after:execution(Buffer.isEmpty())
  {
    sem.unlock();
  }
```

Pointcut

Kind

Advice

Advice code

- Domain-Specific Aspect Languages
  - Targeted to one kind of aspect (COOL - Synchronization, RG - Loop optimization)
  - Describes a concern, adapted joinpoint model, pointcut language
- General-Purpose Aspect Languages
  - More aspects possible with same abstractions
  - Describe croscutting

Symmetric vs Asymmetric

Asymmetric

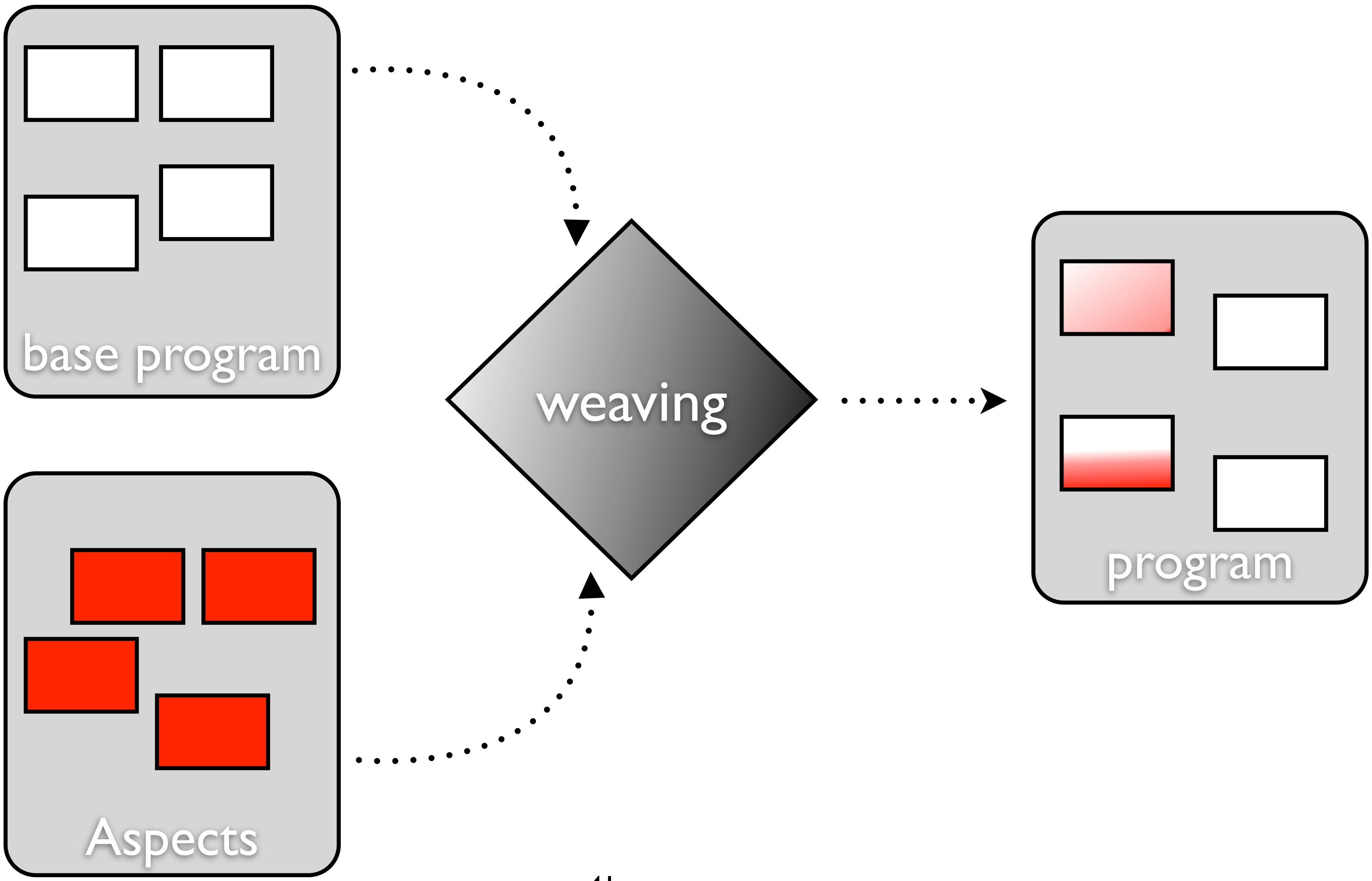
Different module kind for crosscutting concerns

Described until now

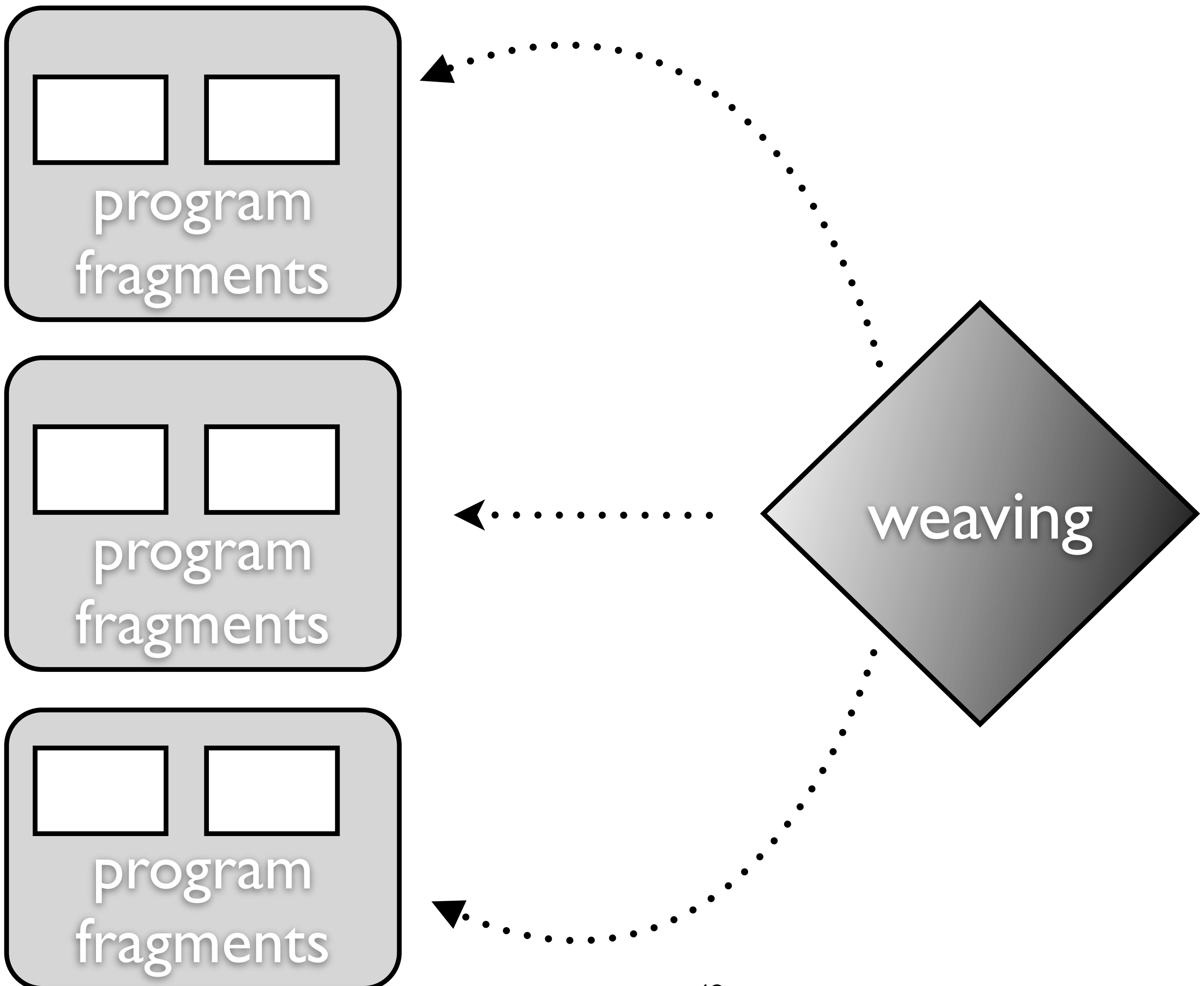
Symmetric

All concerns are modularized with the same kind of module

Asymmetric Aspects

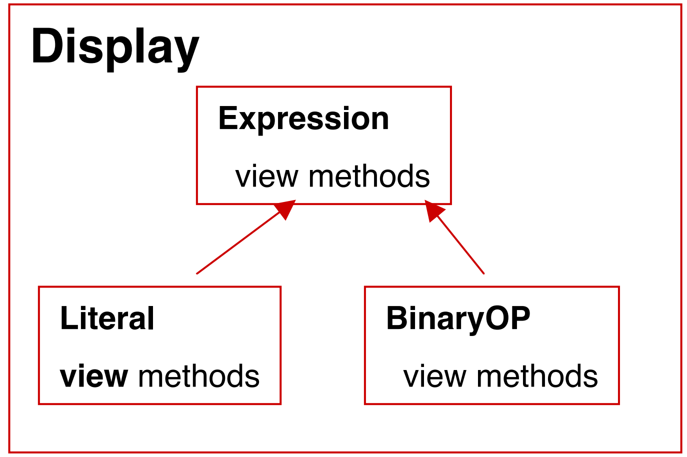


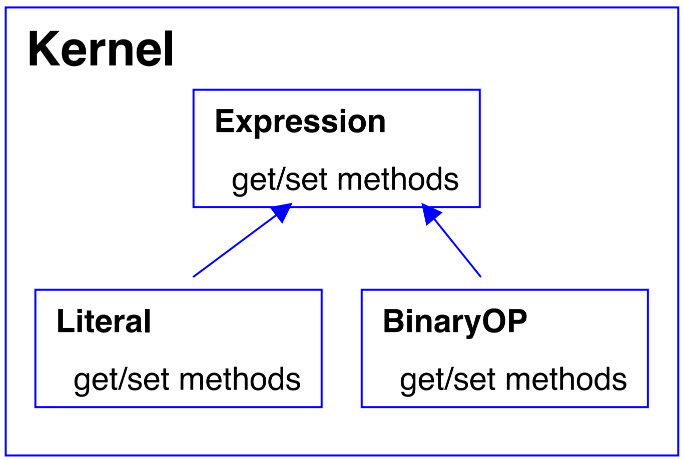
Symmetric Aspects

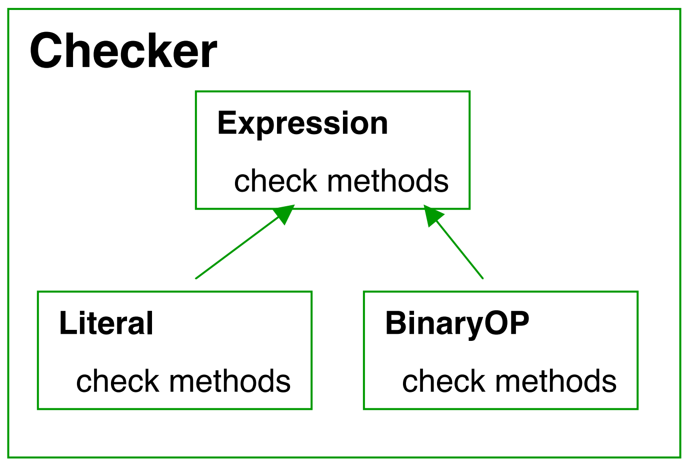


Multidimensional SoC
a symmetric approach

Each concern
is defined in
isolation

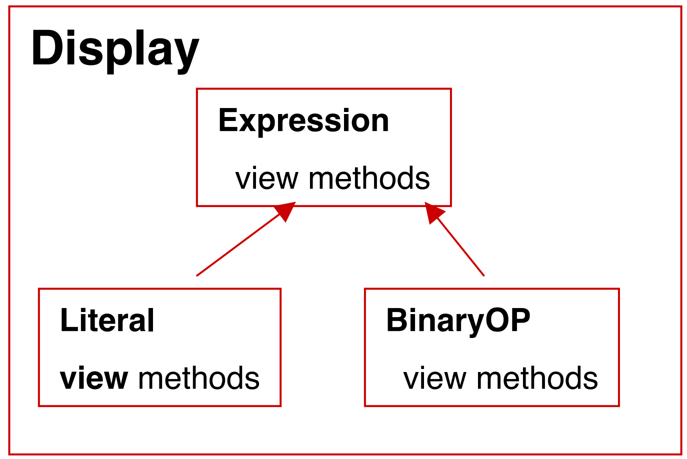


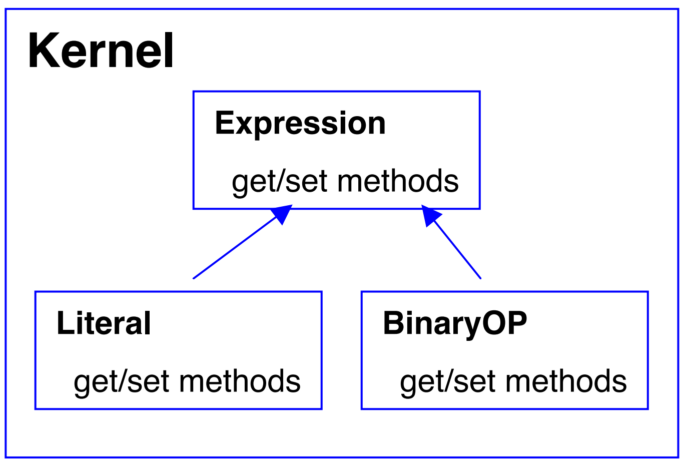


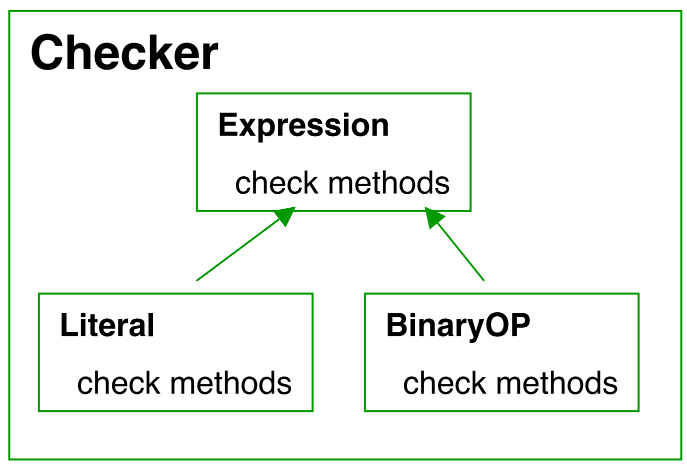


Multidimensional SoC
a symmetric approach

Composition is specified in terms of joinpoints

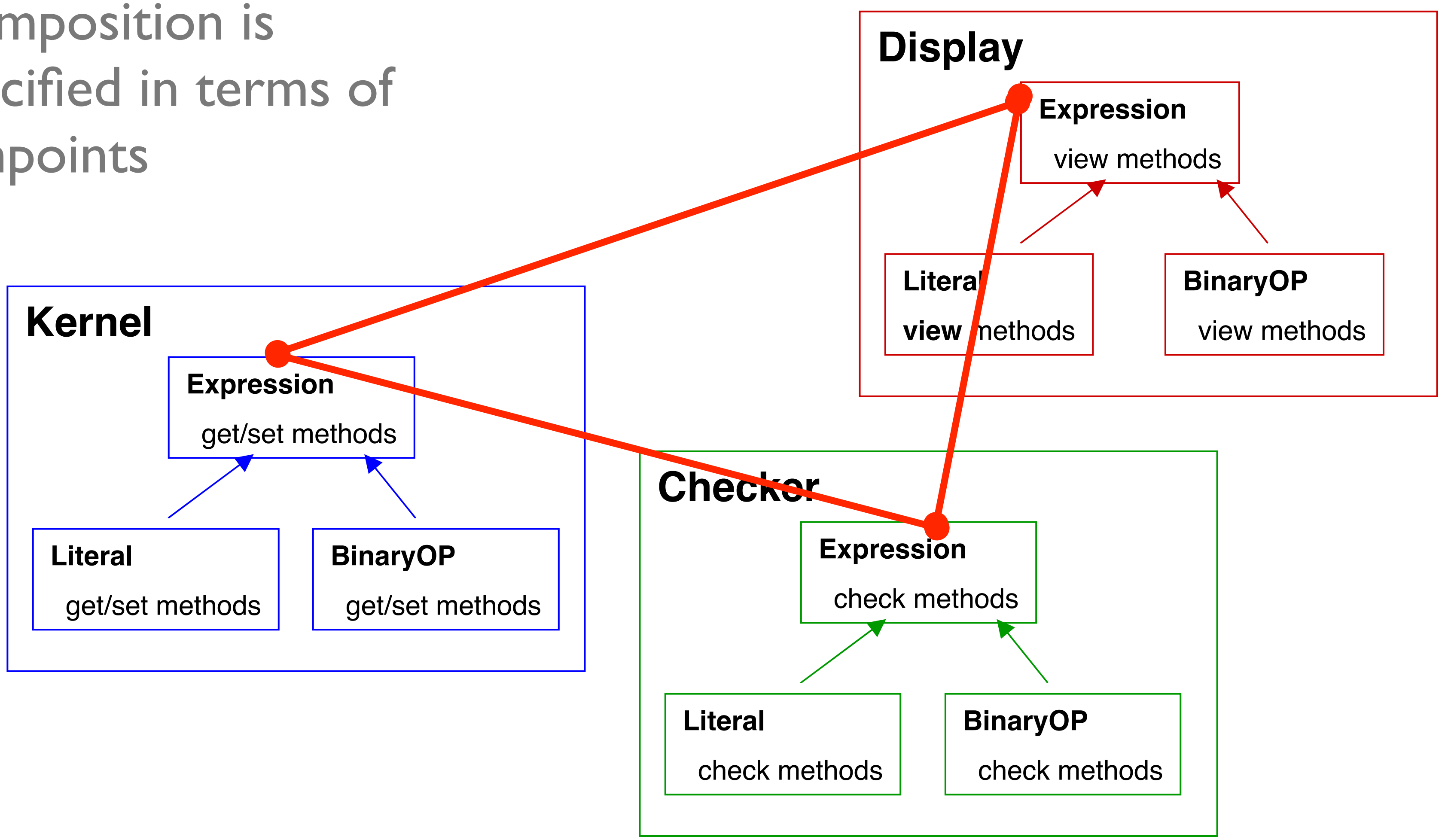






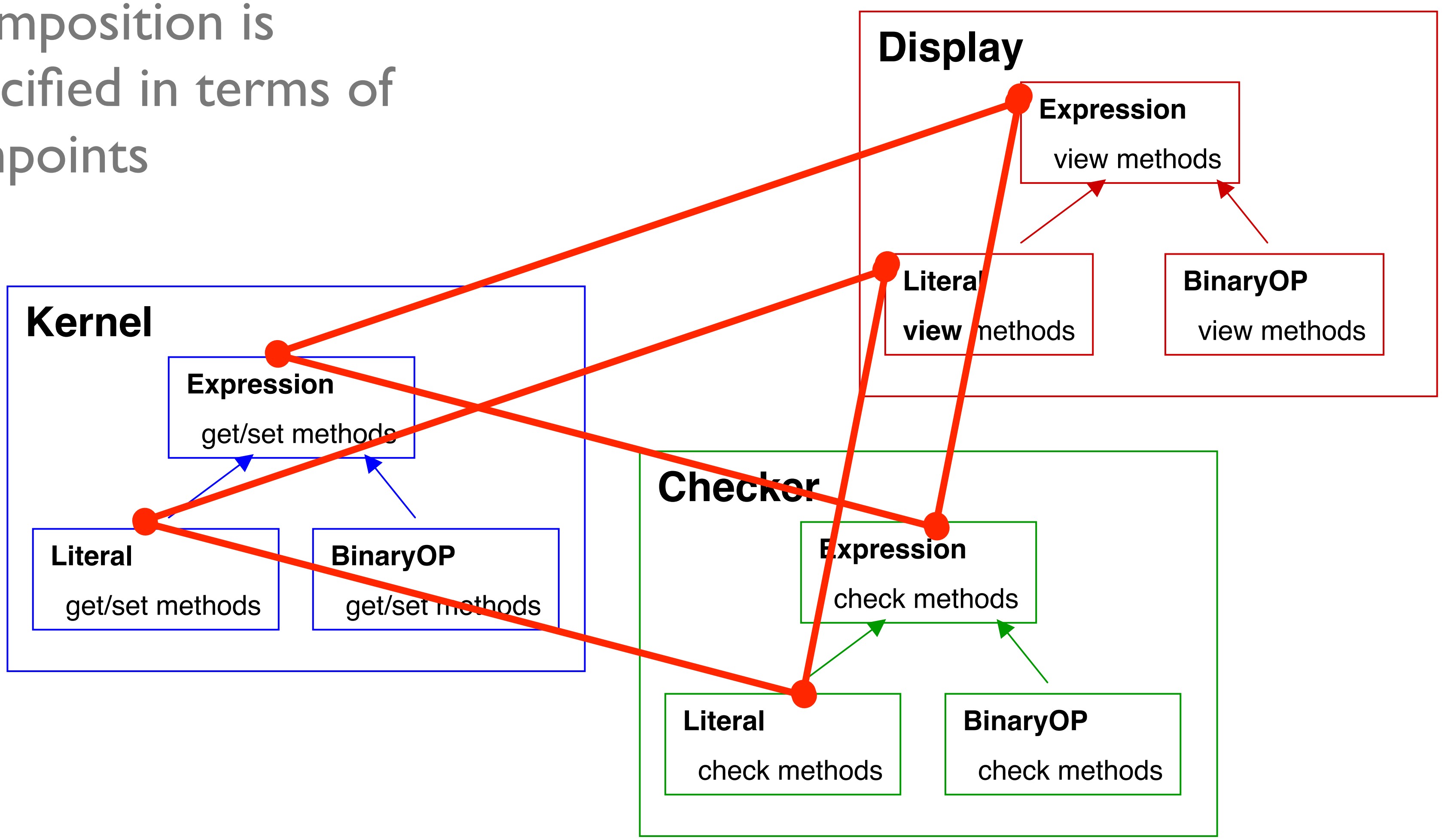
Multidimensional SoC
a symmetric approach

Composition is specified in terms of joinpoints



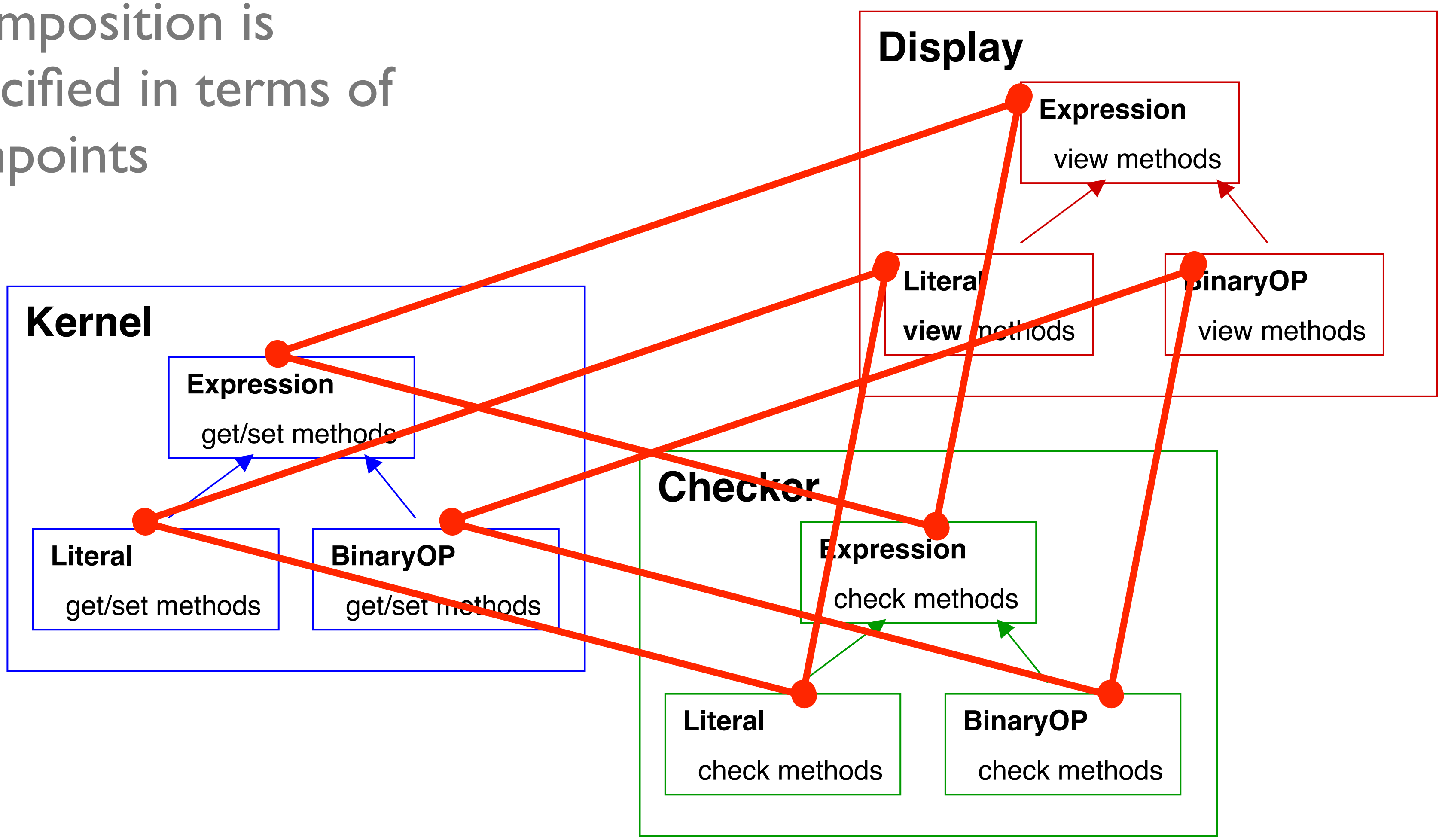
Multidimensional SoC
a symmetric approach

Composition is specified in terms of joinpoints



Multidimensional SoC a symmetric approach

Composition is specified in terms of joinpoints



Example: symmetric synchronized buffer

```
class Buffer {
  char[] data;
  int numElem;

  bool isEmpty() {
    bool retV;
    retV = numElem ==0;
    return retV;
  }
}
```

```
class Synchronization{
  Semaphore sem;

  bool lock(){
    return sem.writeLock();
  }

  bool unlock(){
    return sem.unlock();
  }
```

Example: symmetric synchronized buffer

```
class Buffer {
  char[] data;
  int numElem;

  bool isEmpty() {
    bool retV;
    retV = numElem ==0;
    return retV;
  }
}
```

```
class Synchronization{
  Semaphore sem;

  bool lock(){
   return sem.writeLock();
  }

  bool unlock(){
   return sem.unlock();
  }
```

Vrije Universiteit Brussel

Design Patterns & AOSD

Agenda

- Design Patterns
  - Introduction
  - GoF Patterns
- Design Patterns and AOP
  - Observer
  - Composite
  - Flyweight
  - Singleton
  - ...

Design Patterns

- Collect and Characterize recurring architectures
  - Provide solution to a problem
  - Common language
  - Tend to be small (large ones exist)
- No immediate implementation
  - Partial implementation
  - Smaller than a Framework
  - Architectural counterpart to Programming Idiom

Elements of a Design Pattern

- Name
- Problem
  - Conditions of applicability
- Solution
  - Elements (classes, objects), Roles, Responsibilities
  - No concrete design, implementation
- Consequences
  - Tradeoffs
  - Implementation issues

Sorts of Design Patterns

Creational Patterns:
are concerned with the process of object creation

Structural Patterns:
are concerned with how classes and objects are composed to form larger structures

purpose

Behavioural Patterns:
are concerned with algorithms and the assignment of responsi-bilities between objects

Class Patterns deal with static relationships between classes and subclasses

scope

Object Patterns deal with object relationships which can be changed at run time

Vrije Universiteit Brussel

Overview

Creational Patterns

- Singleton
- Abstract factory
- Factory Method
- Prototype
- Builder

Structural Patterns

- Composite
- Façade
- Proxy
- Flyweight
- Adapter
- Bridge
- Decorator

Behavioral Patterns

- Chain of Respons.
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Implementing Design Patterns

- Design patterns offer **flexible** solutions to common software development problems
  - Sample implementations are geared towards "state-of-the-art" OO languages
- Implementation language affects DP implementation
- Roll-your-own vs. Pattern Library

Challenges with Design Pattern Implementations

- Patterns influence the system structure and vice versa
  - Patterns implementations "disappear in the code" and lose their modularity, *pattern code is scattered and tangled* with system code
  - Adding or removing a pattern is invasive, difficult to reverse change
- Pattern composition/overlay
  - Systems are difficult to reason about when multiple patterns are used and involve the same classes

Agenda

- Design Patterns
  - Introduction
  - GoF Patterns
- **Design Patterns and AOP**
  - Observer
  - Composite
  - Flyweight
  - Singleton
  - ...

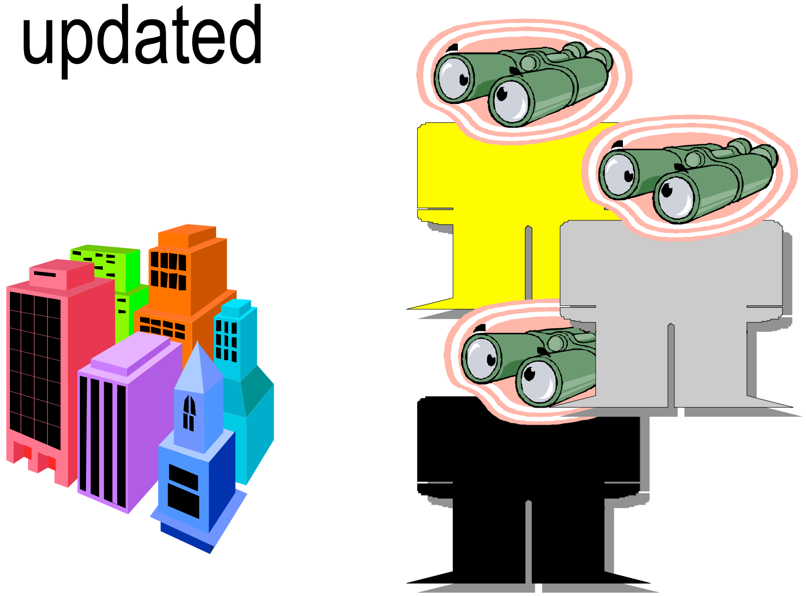
GoF design patterns in AspectJ (Hannemann & Kiczales)

- Develop and compare Java and AspectJ implementations of the 23 GoF patterns
  - Only solution structure and solution implementations can change
  - Not about discovery of new (AOP) patterns
- Results:
  - 17 can be modularized
  - For 12 of these the modularization enables a core part of the implementation to be abstracted into reusable code
  - For 14 transparent composition of pattern instances is possible

OBSERVER

The Observer Pattern: The Problem

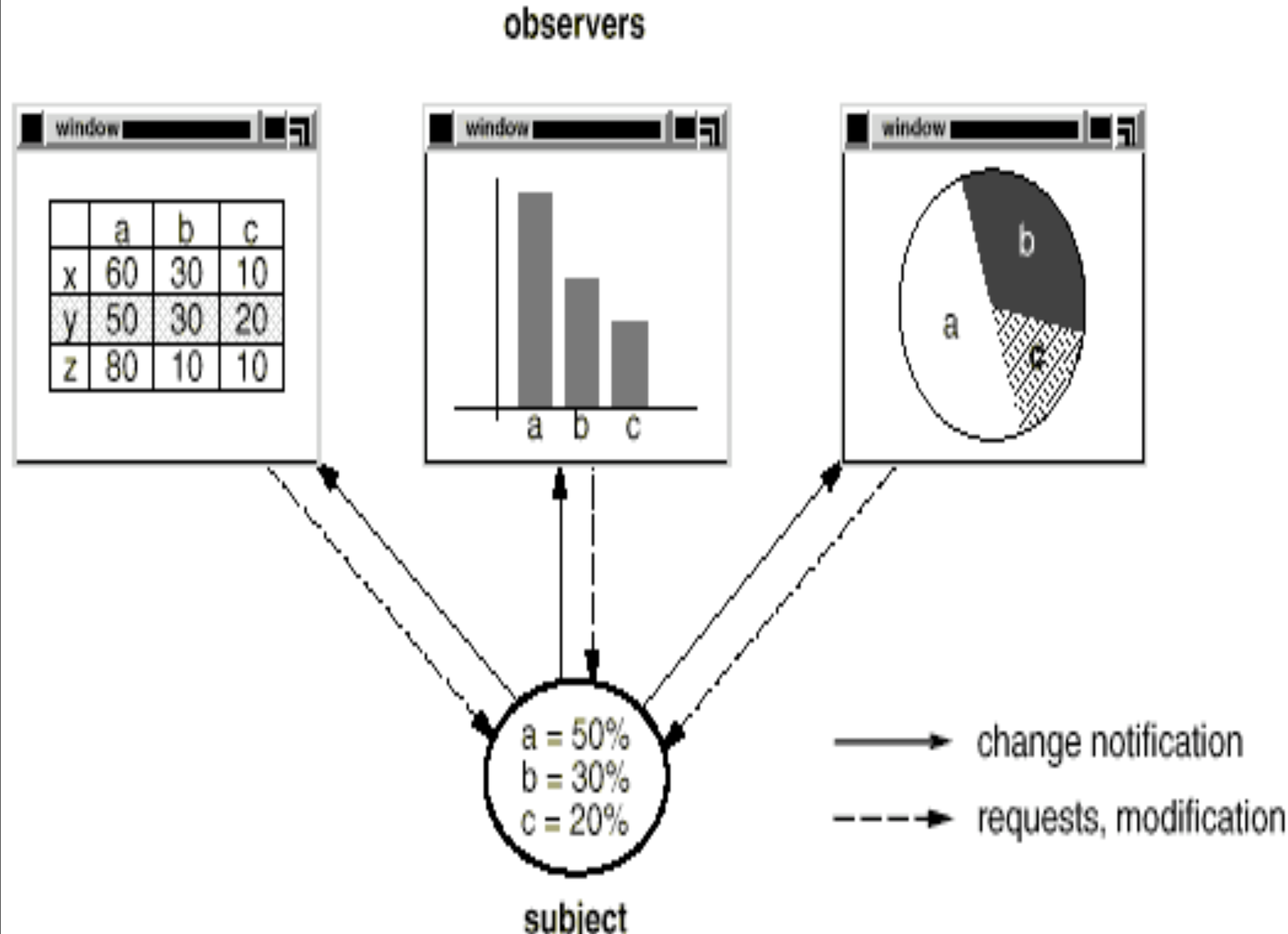
Assume a one to many relationship between objects, when one changes the dependents must be updated



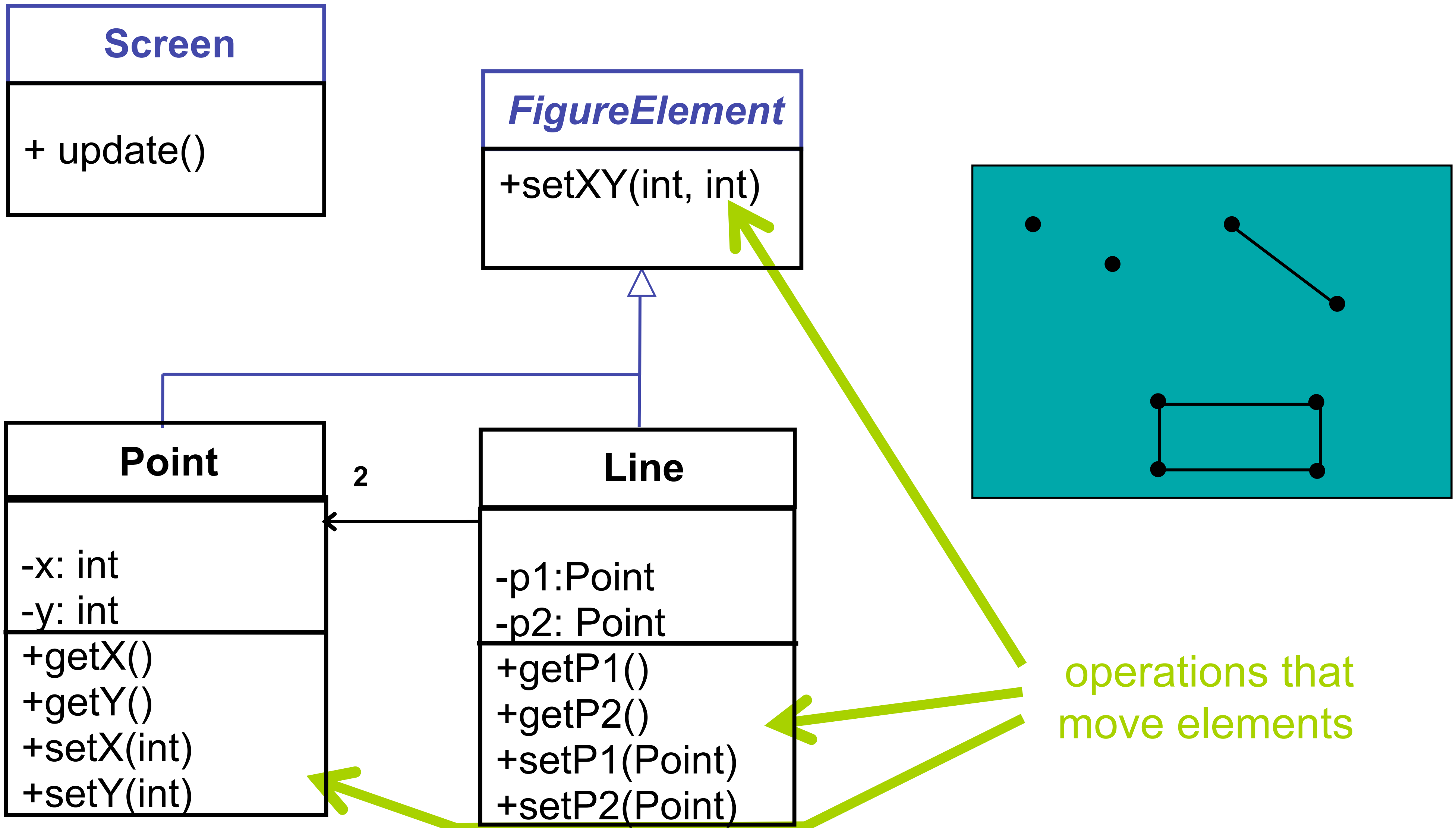
Subject Observers

- different types of GUI elements depicting the same application data
- different windows showing different views on the same application model

Also known as : Dependants, Publish-Subscribe


observers
window
window
window
a
b
c
x
60
30
10
y
50
30
20
z
80
10
10
a
b
c
b
a
c
a = 50%
b = 30%
c = 20%
change notification
requests, modification
subject

A simple figure editor



Screen updating

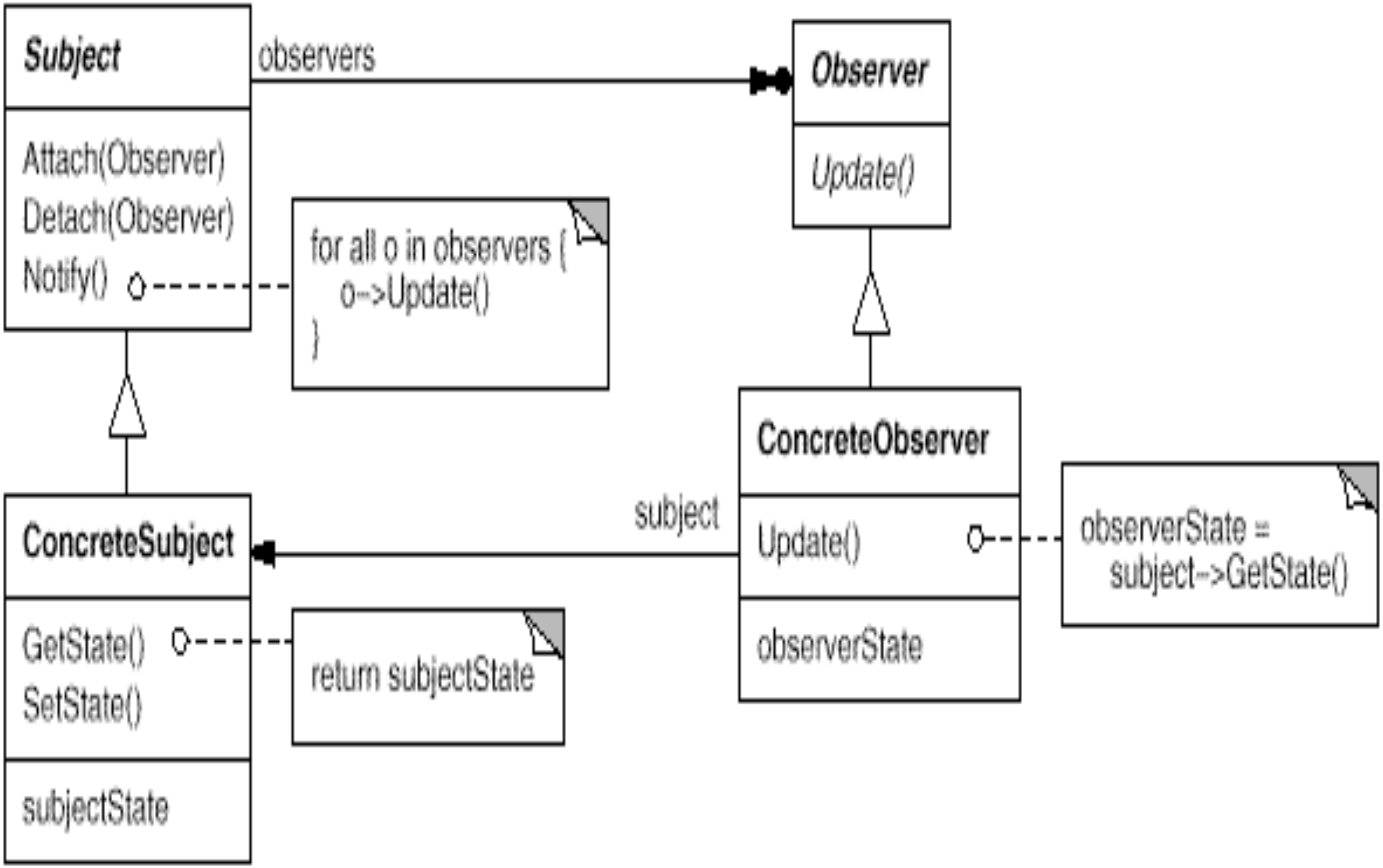
- Supports the code that refreshes the screen when a figure element moved
- Whithout AOP every method that updates the position of a figure element should call update

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.setXY(int, int))  ||
    call(void Line.setP1(Point))               ||
    call(void Line.setP2(Point))               ||
    call(void Point.setX(int))                 ||
    call(void Point.setY(int));

  after() returning: move() {
    Screen.update();
  }
}
```

Change monitoring

- Supports the code that monitors whether a figure element moved
- Whithout AOP every method that updates the position of a figure element should manipulate the dirty bit

```
aspect MoveTracking {
    private static boolean dirty = false;
    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;    }
    pointcut move():
        call(void FigureElement.setXY(int, int)) ||
        call(void Line.setP1(Point))             ||
        call(void Line.setP2(Point))             ||
        call(void Point.setX(int))               ||
        call(void Point.setY(int));
    after() returning: move() {
        dirty = true;
    }
}
```

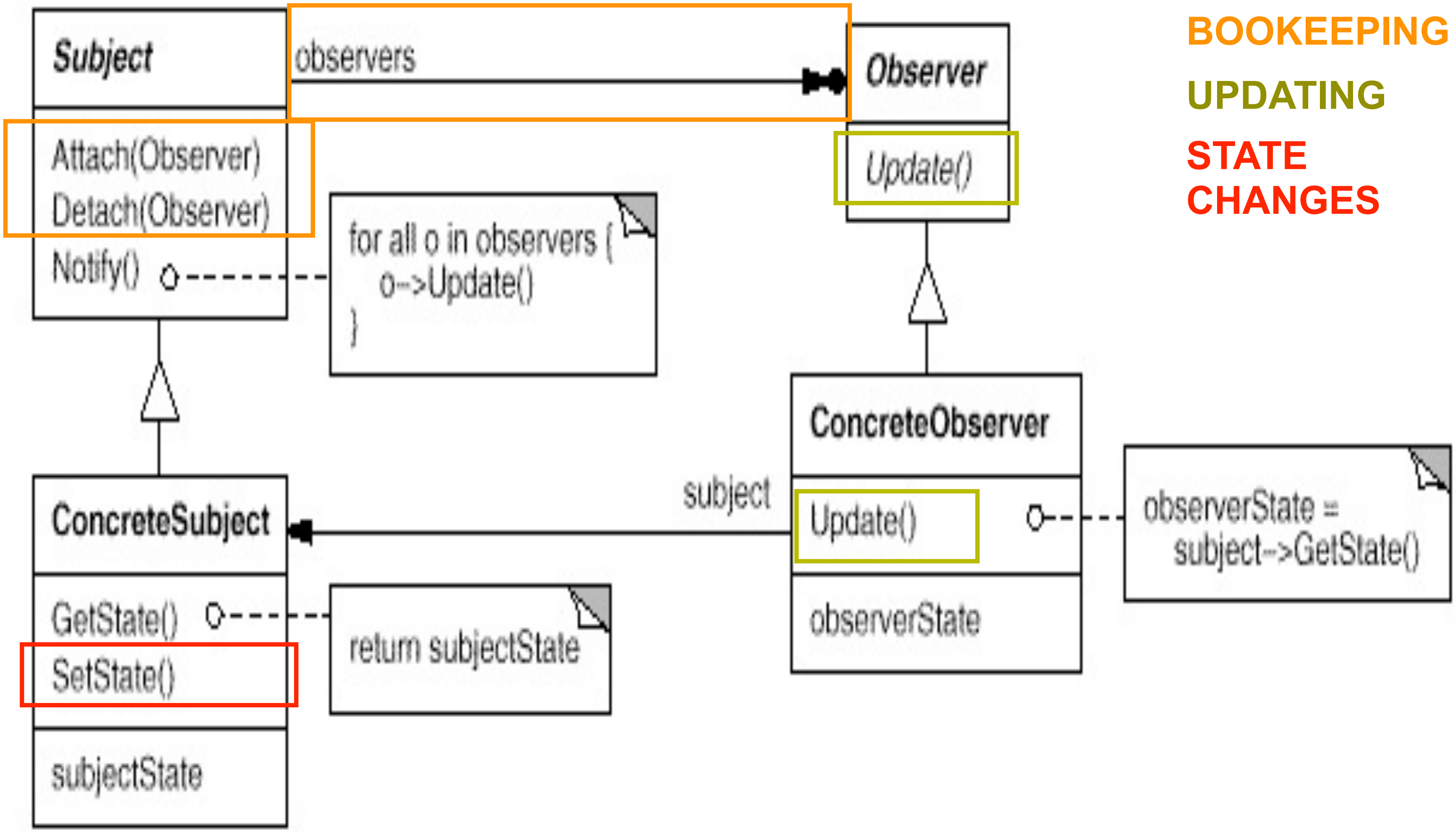

Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

The Observer Pattern Participants

- Subject: knows its observers, provides an interface for attaching (subscribe) and detaching (unsubscribe) observers and provides a *notify* method that calls *update* on all its observers
- Observer: provides an *update* interface
- ConcreteSubject: maintains a state relevant for the application at hand, provides methods for getting and setting that state, calls notify when its state is changed
- ConcreteObserver: maintains a reference to a concrete subject, stores a state that is kept consistent with the subject's state and implements the observer's *update* interface

Vrije Universiteit Brussel

The observer pattern



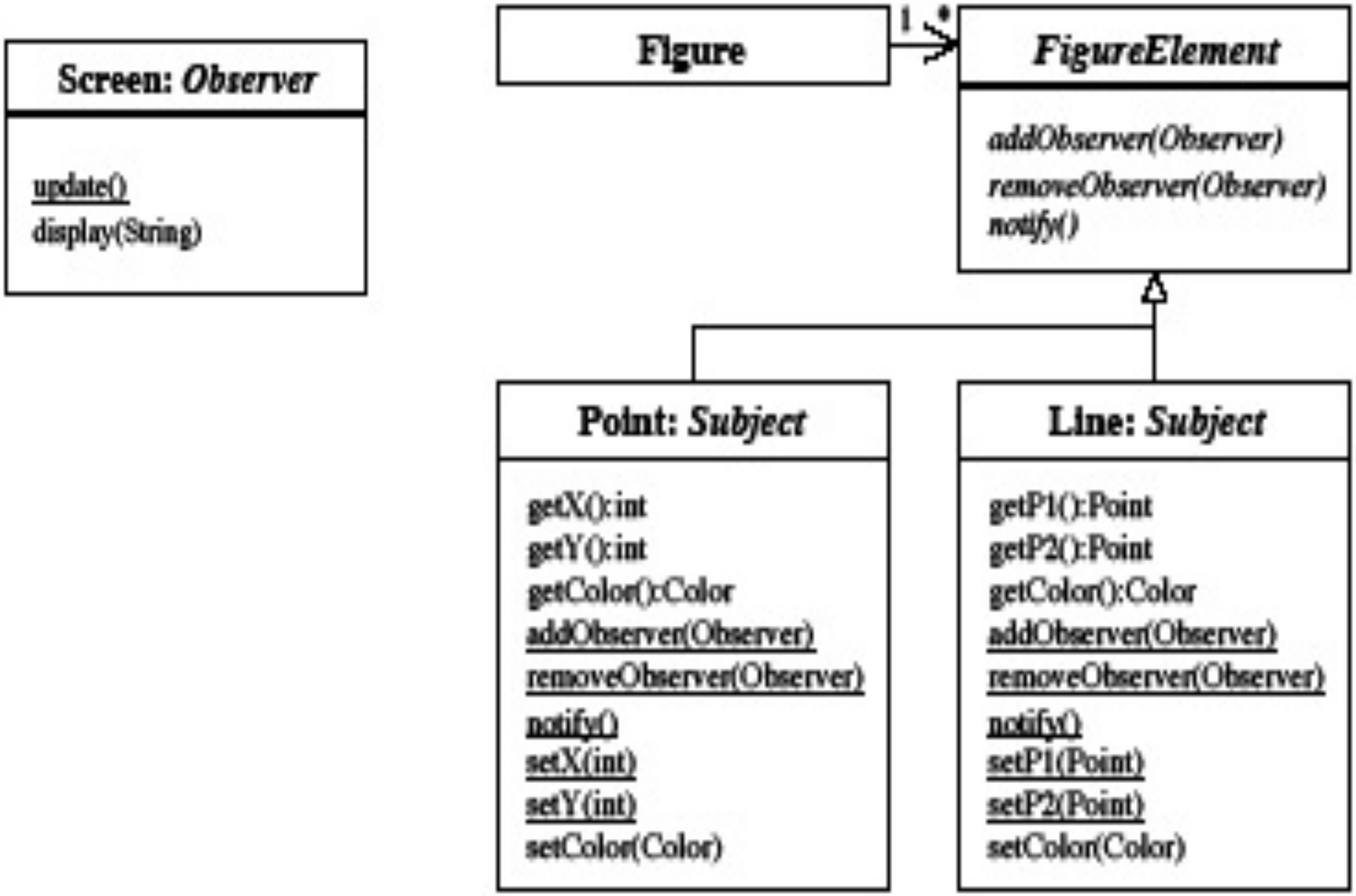
BOOKEEPING

UPDATING

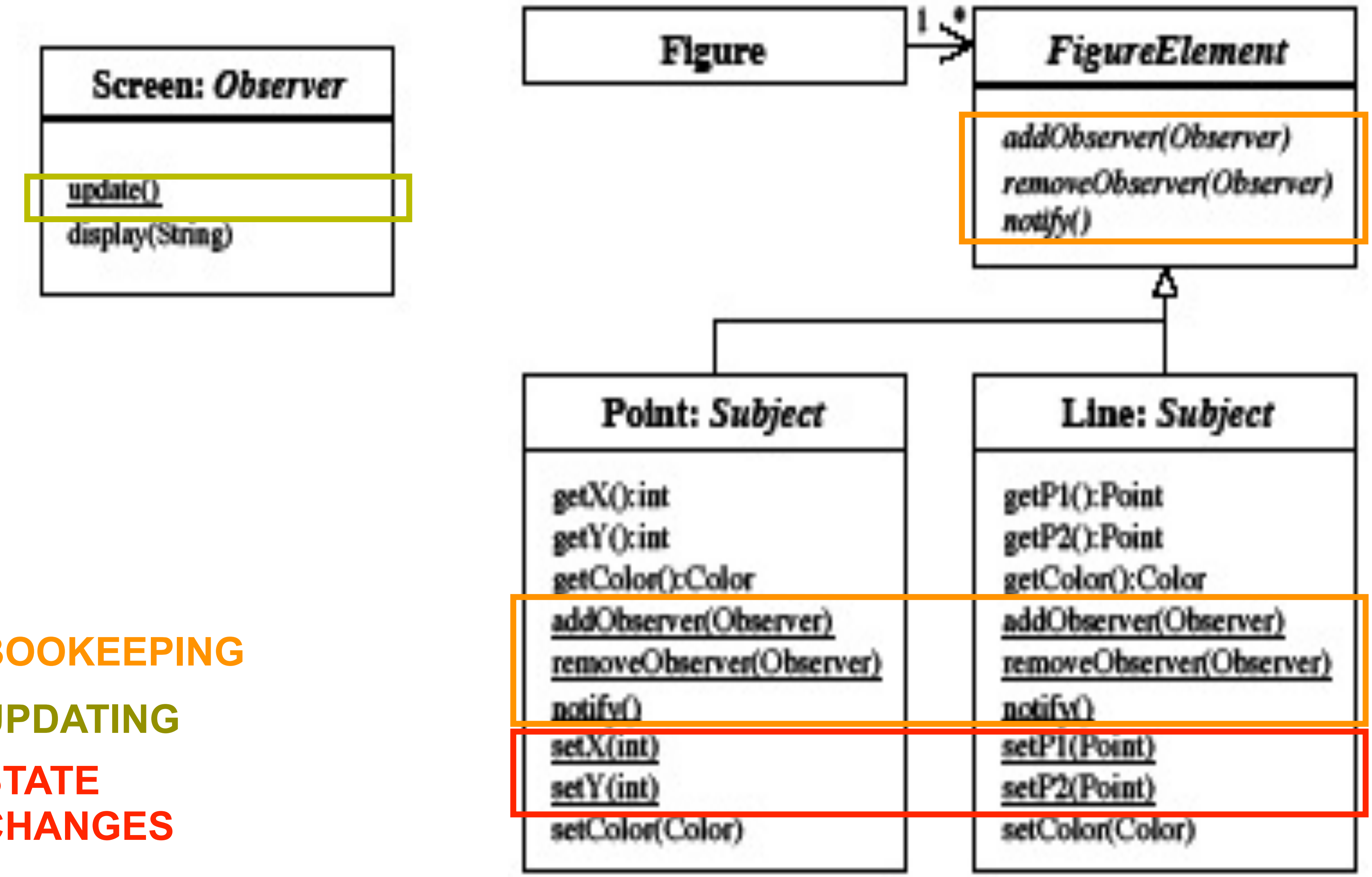
STATE CHANGES

Vrije Universiteit Brussel

The Observer Pattern in Java for Screen Updates



The Observer Pattern in Java for Screen Updates

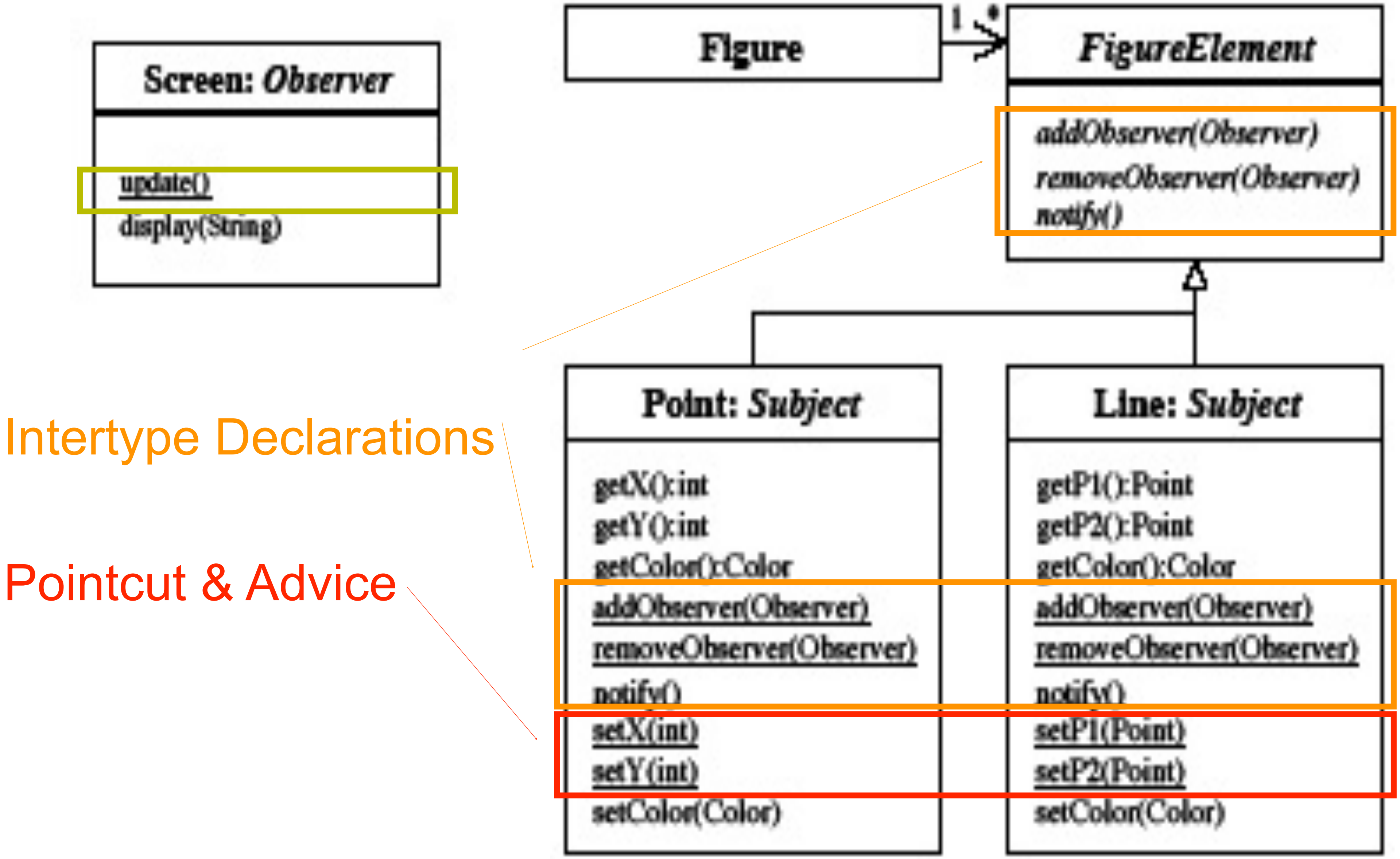


BOOKEEPING

UPDATING

STATE CHANGES

A naïve implementation of the Observer Pattern in AspectJ for screen updates



Friday 15 February 13

A naïve implementation (1)

```
public aspect ScreenUpdate{
    private Set FigureElement.observers = new HashSet();
    public void FigureElement.addObserver(Screen s) {
        this.observers.add(s);
    }
    public void FigureElement.removeObserver(Screen s) {
        this.observers.remove(s);
    }
    public void FigureElement.notifyObservers() {
        Iterator it = observers.iterator();
        while(it.hasNext()) {
            ((Screen)it.next()).update();
        }
    }
```

BOOKEEPING

Vrije Universiteit Brussel

A naïve implementation (2)

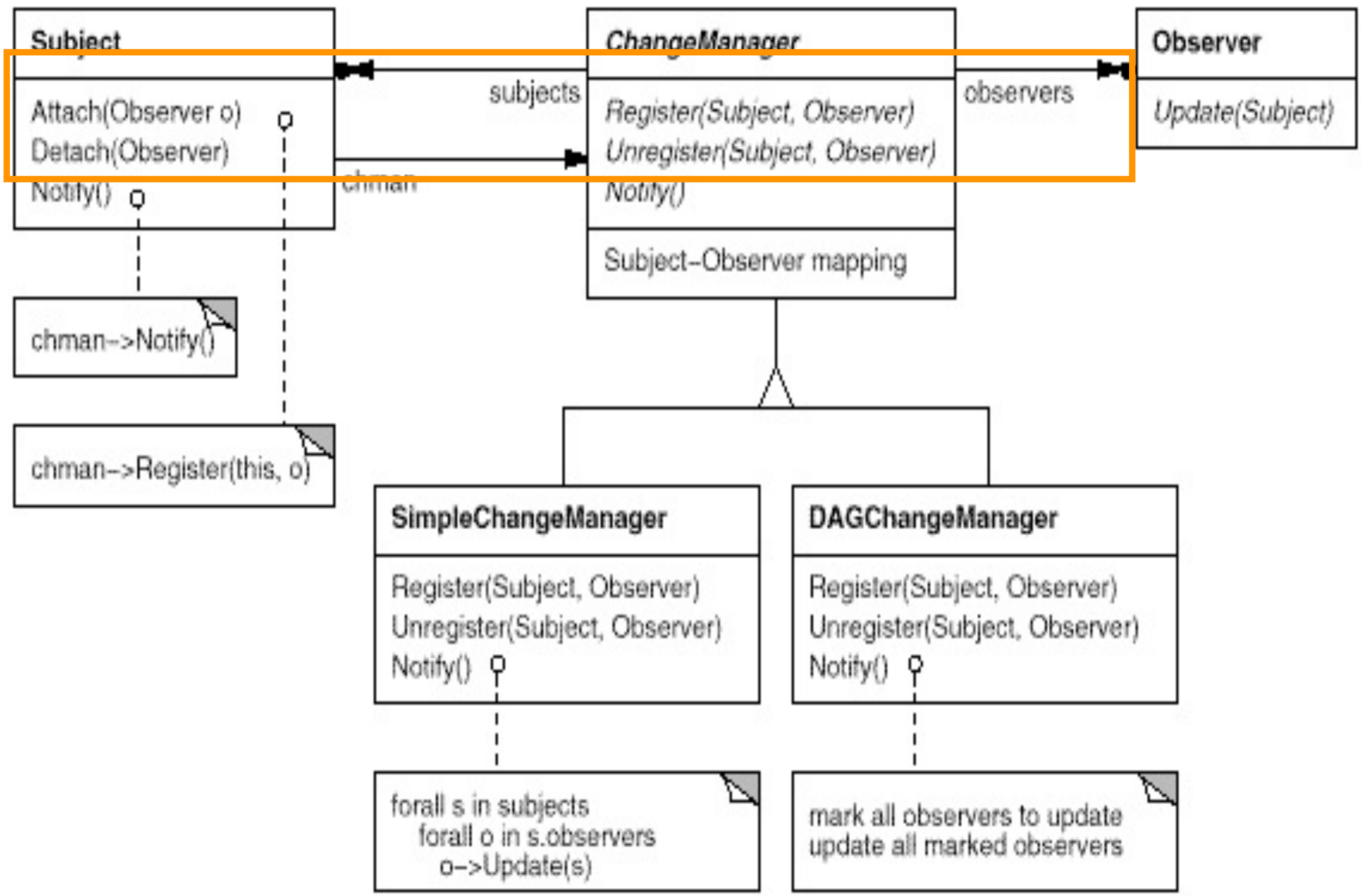
```
public void Screen.update() {
        // Update screen...
   }
```

UPDATING

STATE CHANGES

```
  pointcut subjectChange(FigureElement fe):
       (call(void Line.setP1(Point))              ||
        call(void Line.setP2(Point))              ||
        call(void Point.setX(int))                ||
        call(void Point.setY(int))) && target(fe);

  after(FigureElement fe): subjectChange(fe) {
        fe.notifyObservers();
  }
}
```

The observer pattern with an explicit change manager



A ColorObserving Aspect (1)

```
public aspect ColorObserver {
    private WeakHashMap perSubjectObservers;
    protected List getObservers(FigureElement subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers =(List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }
    public void addObserver(FigureElement subject, Screen observer) {
        getObservers(subject).add(observer);
    }
    public void removeObserver(FigureElement subject, Screen observer) {
        getObservers(subject).remove(observer);
    }
```

A ColorObserving Aspect (2)

```
pointcut subjectChange(FigureElement subject):
    (call(void Point.setColor(Color)) ||
     call(void  Line.setColor(Color)) ) && target(subject);

after(FigureElement subject): subjectChange(subject) {
    Iterator iter = getObservers(subject).iterator();
    while (iter.hasNext()) {
        updateObserver(subject, ((Screen)iter.next()));
    }
}

public void updateObserver(FigureElement subject, Screen observer) {
    // Update screen...
}
}
```

A CoordinateObserving Aspect (1)

```
public aspect CoordinateObserver {
    private WeakHashMap perSubjectObservers;
    protected List getObservers(FigureElement subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers =(List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }
    public void addObserver(FigureElement subject, Screen observer) {
        getObservers(subject).add(observer);
    }
    public void removeObserver(FigureElement subject, Screen observer) {
        getObservers(subject).remove(observer);
    }
```

Vrije Universiteit Brussel

A CoordinateObserving Aspect (2)

```
pointcut subjectChange(FigureElement subject):
     (call(void Point.setX(int)) ||
      call(void Point.setY(int)) ||
      call(void Line.setP1(Point)) ||
      call(void Line.setP2(Point)) ) && target(subject);

 after(FigureElement subject): subjectChange(subject) {
     Iterator iter = getObservers(subject).iterator();
     while (iter.hasNext()) {
         updateObserver(subject, ((Screen)iter.next()));
     }
 }
 public void updateObserver(FigureElement subject, Screen observer) {
     // Update screen...
 }
}
```

An Abstract Observer-Protocol Aspect (1)

```
public abstract aspect ObserverProtocol {

  protected interface Subject  { }
  protected interface Observer { }

  private WeakHashMap perSubjectObservers;

  protected List getObservers(Subject s) {
    if (perSubjectObservers == null) {
      perSubjectObservers = new WeakHashMap();
    }
    List observers =
      (List)perSubjectObservers.get(s);
    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(s, observers);
    }
    return observers;
  }
```

An Abstract Observer-Protocol Aspect (2)

```

public void addObserver(Subject s,Observer o){
  getObservers(s).add(o);
}
public void removeObserver(Subject s,Observer o){
  getObservers(s).remove(o);
}
```

An Abstract Observer-Protocol Aspect (3)

```
  abstract protected pointcut
    subjectChange(Subject s);

  abstract protected void
    updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) {
      updateObserver(s, ((Observer)iter.next()));
    }
  }
}
```

A Concrete ColorObserver Aspect

```
public aspect ColorObserver extends ObserverProtocol {

  declare parents: Point  implements Subject;
  declare parents: Line   implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    (call(void Point.setColor(Color)) ||
     call(void  Line.setColor(Color)) ) && target(s);

  protected void updateObserver(Subject s,Observer o) {

    ((Screen)o).display("Color change.");
  }
}
```

A Concrete CoordinatorObserver Aspect

```
public aspect CoordinateObserver extends
  ObserverProtocol {

  declare parents: Point  implements Subject;
  declare parents: Line   implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    (call(void Point.setX(int))
     || call(void Point.setY(int))
     || call(void Line.setP1(Point))
     || call(void Line.setP2(Point)) ) && target(s);

  protected void updateObserver(Subject s, Observer o) {
     ((Screen)o).display("Coordinate change.");
  }
```

A Concrete ScreenObserver Aspect

```
public aspect ScreenObserver
                    extends ObserverProtocol {

  declare parents: Screen implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    call(void Screen.display(String)) && target(s);

  protected void updateObserver(
    Subject s, Observer o) {
      ((Screen)o).display("Screen updated.");
  }
}
```

Properties of this solution

- Locality:
  - All pattern code is in the abstract and concrete observer aspects
  - The participants are free of pattern context and therefor there is no coupling between them
- Reusability:
  - The abstract ObserverProtocol aspect can be reused and shared
- Composition Transparency
  - Because the participants are in no way coupled with the pattern they can take part in many other patterns
- (Un)pluggability
  - Switching between using/not using the pattern is easy because all the code is in the aspects

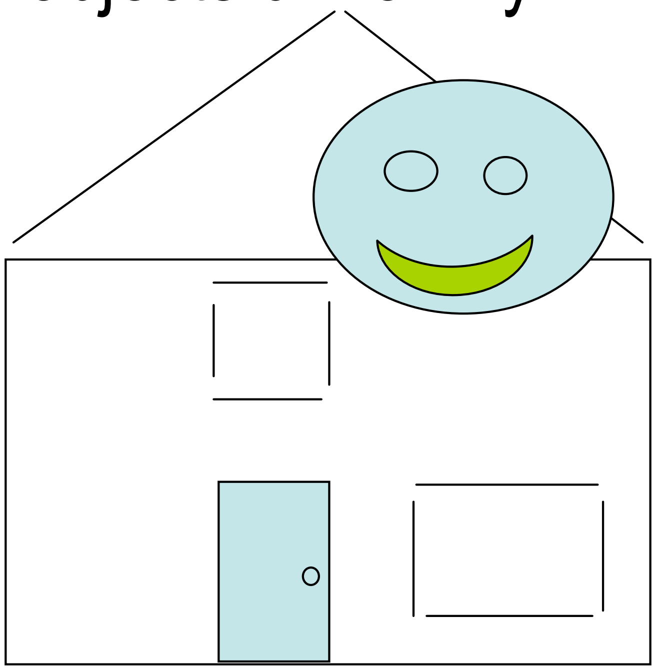
Generalizing the Results

- Patterns that introduce roles that need no client-accessible interface and that are only used within the pattern:
  - The role can be realized with empty (protected) interfaces in an aspect. The interfaces introduce types to be used within the pattern protocol
  - An abstract aspect can define the roles and attach default implementations where possible
  - The abstract aspect can define an abstract pointcut to capture join points that should trigger important events
- Composite, Command, Mediator, Chain of Responsibility

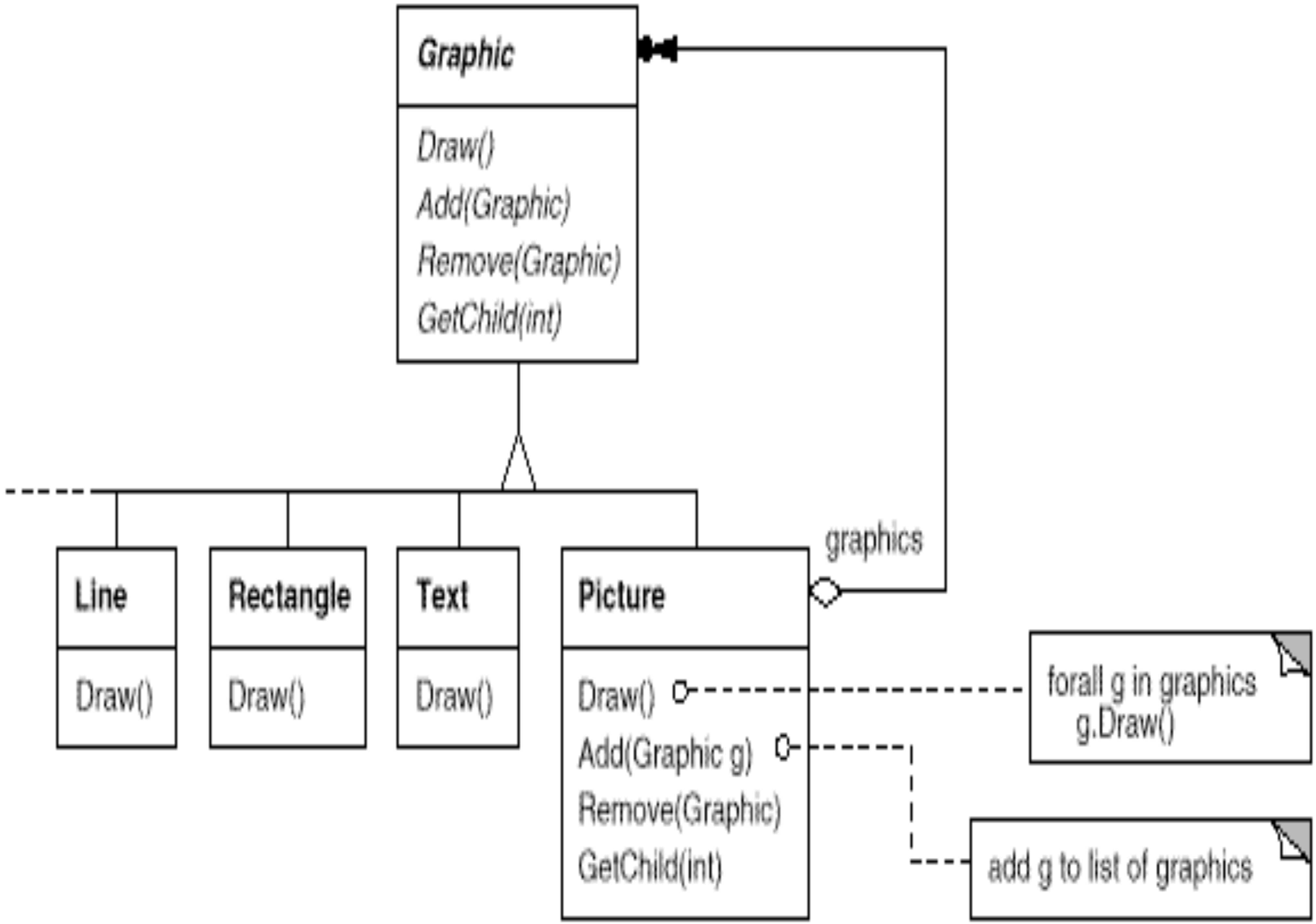
COMPOSITE

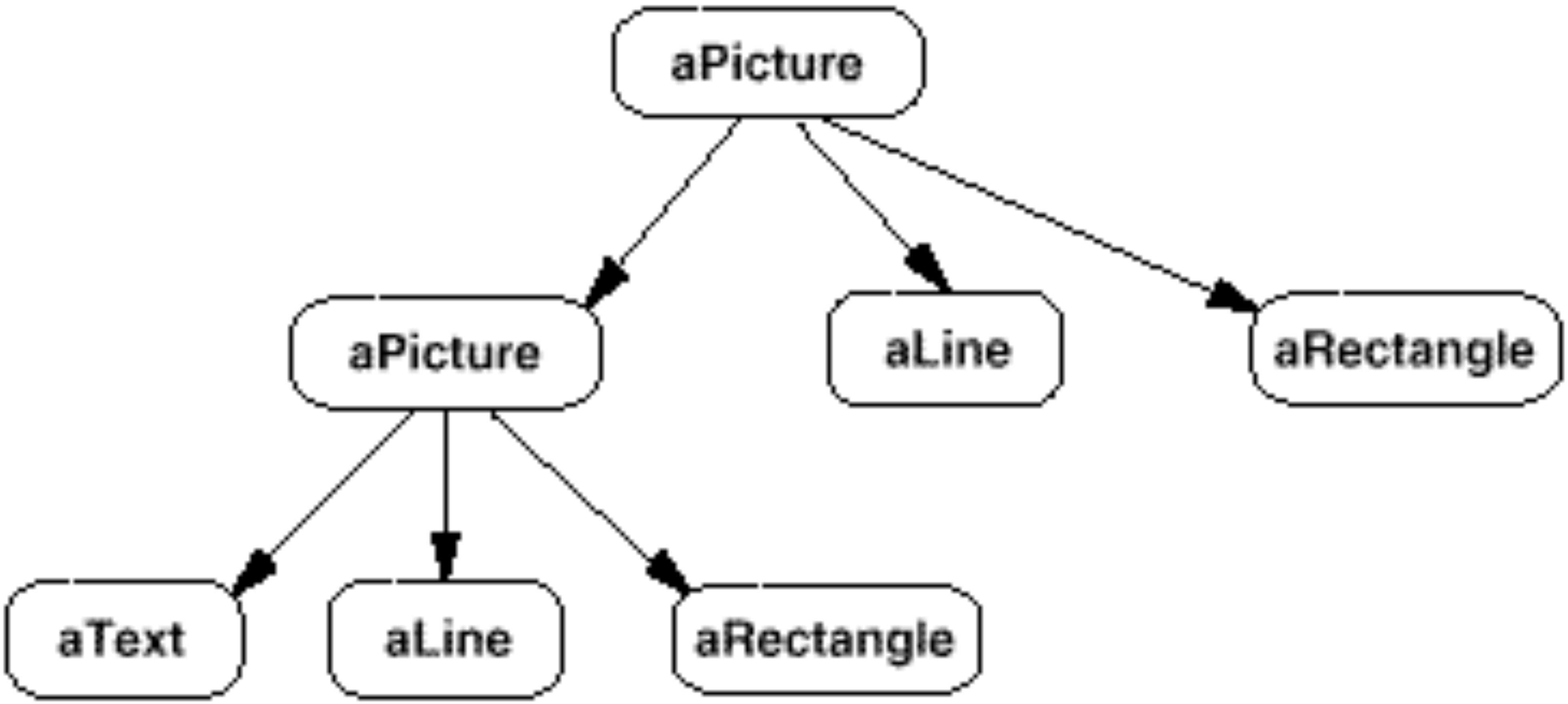
The Composite Pattern: The Problem

Compose objects into tree-like structures to represent part-whole hierarchies and let clients treat individual objects and compositions of objects uniformly



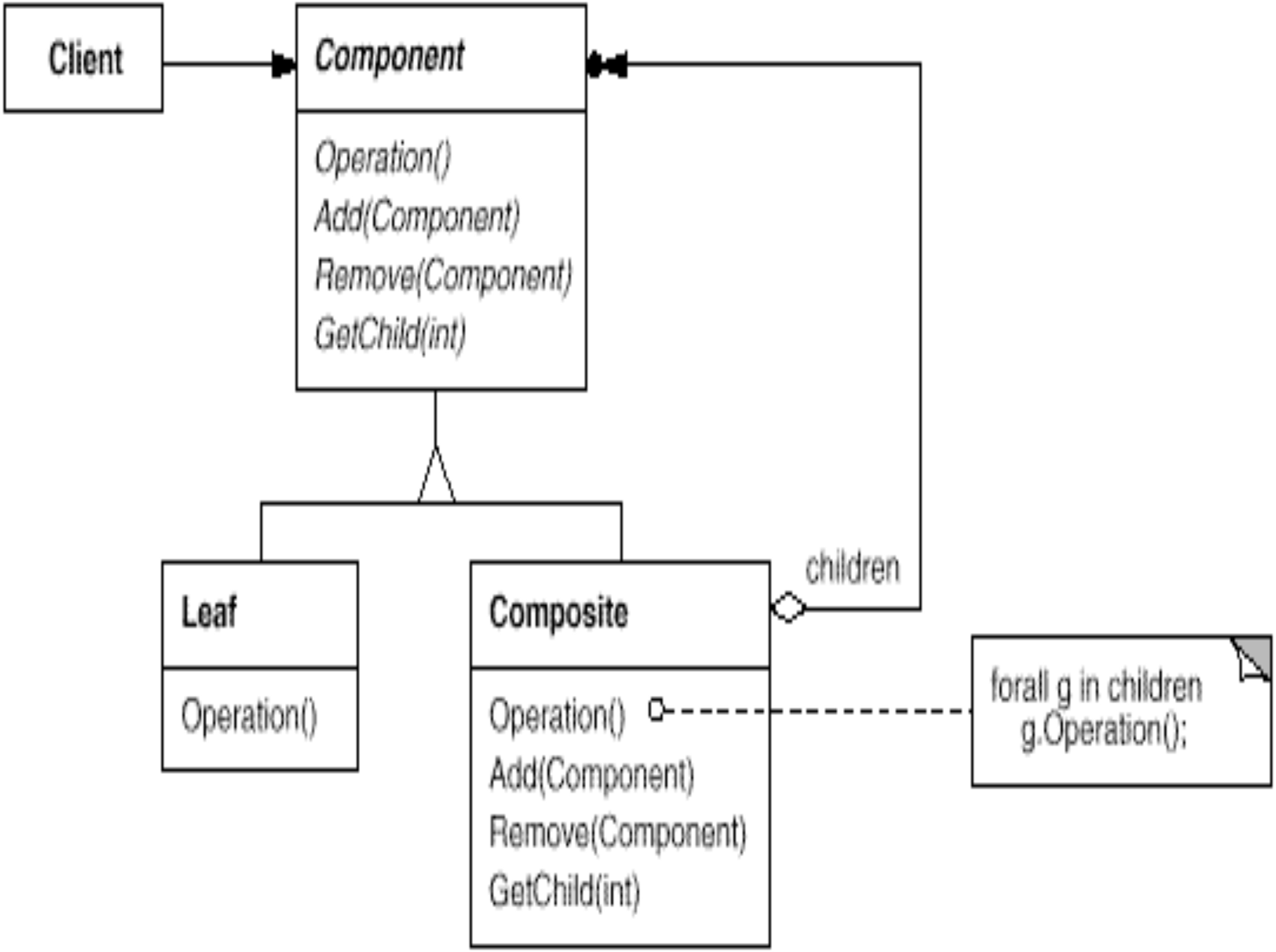
- a drawing tool that lets users build complex diagrams from simple elements
- trees with heterogeneous nodes e.g. the parse tree of a program
- a containment hierarchy for technical equipment


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics


aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

The Composite Pattern Participants

- Component: declares the interface for objects in the composition, implements default behavior for the interface common to all objects, declares an interface for accessing and managing child components, (optional) defines/implements an interface for accessing a component's parent
- Leaf: defines behavior for primitive objects in the composition
- Composite: defines behavior for components having children, stores child components, implements child access and management operations in the component interface
- Client: manipulates objects in the composition through the component interface


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

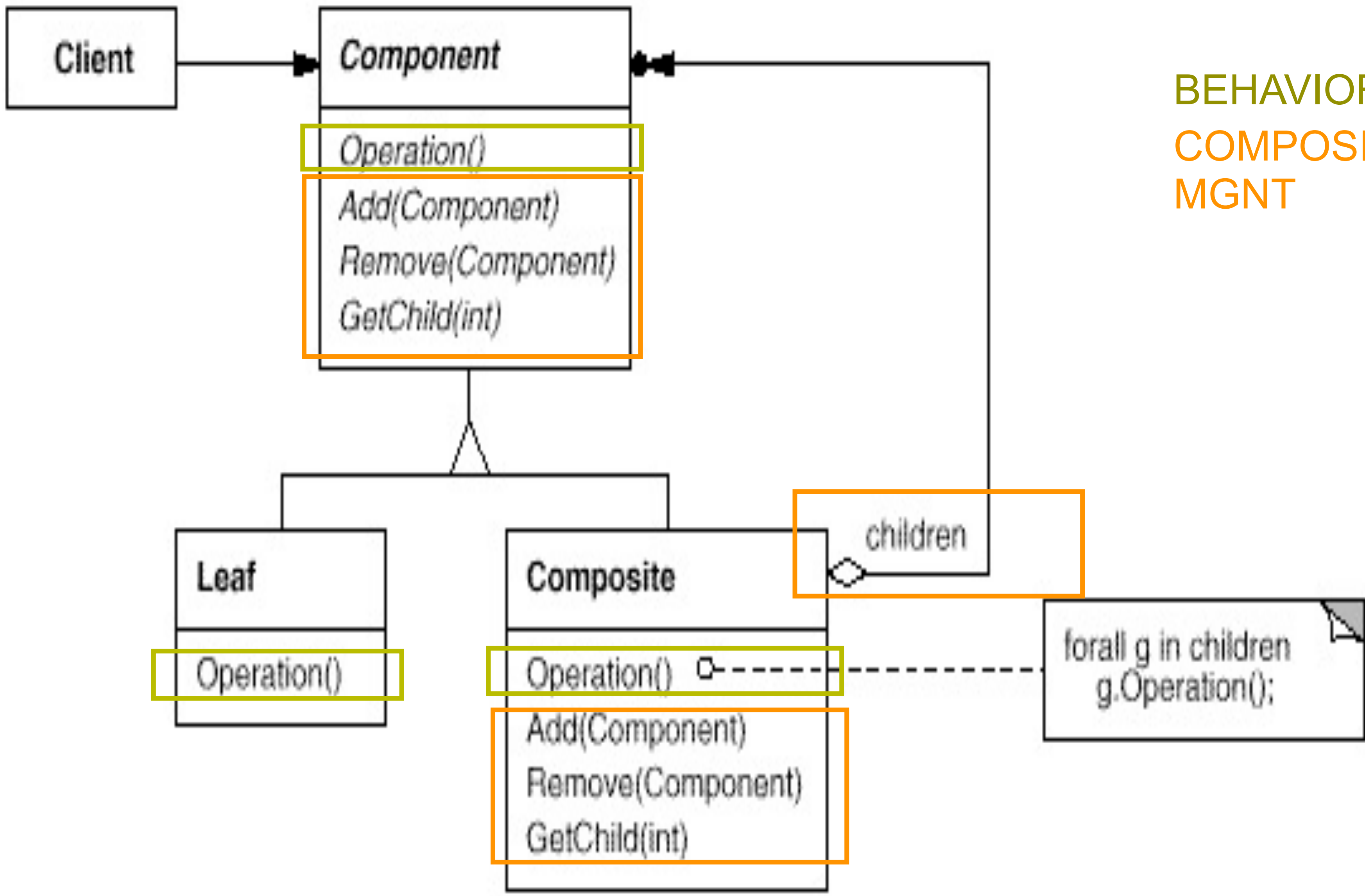
The Composite Pattern Collaboration

- Clients use the Component class interface to interact with objects in the composition
- If the recipient is a Leaf, the request is handled directly
- If the recipient is a Composite the request is usually forwarded to child components, some additional operations before and/or after the forwarding can happen

The Composite Pattern Consequences

- + Makes the Client simple: clients can treat composite structures and individual objects uniformly, clients normally don't know and should not care whether they are dealing with a leaf or a composite
- + Makes it easier to add new types of components: client code works automatically with newly defined Composite or Leaf subclasses
- - Can make a design overly general: the disadvantage of making it easy to add new components is that it is difficult to restrict the components of a composite, sometimes you want a composite to have only certain types of children, with the Composite Patterns you cannot rely on the type system to enforce this for you, you have to implement and use run-time checks

The Composite Pattern



BEHAVIOR

COMPOSITION MGNT

An Abstract Composition Aspect (1)

BEHAVIOR

```
public abstract aspect CompositionProtocol {

  protected interface Component {}
  protected interface Composite extends Component {}
  protected interface Leaf     extends Component {}

  private WeakHashMap perComponentChildren =
    new WeakHashMap();

  private Vector getChildren(Component s) {
    Vector children;
    children = (Vector)perComponentChildren.get(s);
    if ( children == null ) {
      children = new Vector();
      perComponentChildren.put(s, children);
    }
```

An Abstract Composition Aspect (2)

BEHAVIOR

```
public void addChild(Composite composite,
                     Component component) {
  getChildren(composite).add(component);
}
public void removeChild(Composite composite,
                        Component component) {
  getChildren(composite).remove(component);
}

public Enumeration getAllChildren(Component c) {
  return getChildren(c).elements();
}
}
```

A Concrete Composition Aspect (1)

```
public aspect FileSystemComposite extends
                CompositeProtocol {

 declare parents: Directory implements Composite;
 declare parents: File      implements Leaf;

 public int sizeOnDisk(Component c) {
   return c.sizeOnDisk();
 }

 private abstract int Component.sizeOnDisk();
```

BEHAVIOR

COMPOSITION MGNT

Vrije Universiteit Brussel

A Concrete Composition Aspect (2)

```
private int Directory.sizeOnDisk() {
  int diskSize = 0;
  java.util.Enumeration enum;
  for (enum =
   SampleComposite.aspectOf().getAllChildren(this);
   enum.hasMoreElements(); ) {
     diskSize +=
     ((Component)enum.nextElement()).sizeOnDisk();
  }
  return diskSize;
}
private int File.sizeOnDisk() {
  return size;
}
```

BEHAVIOR

Observer, Composite, Command, Mediator, Chain of Responsibility

- Patterns that introduce roles that need no client-accessible interface and that are only used within the pattern:
  - The roles can be realized with empty (protected) interfaces in an aspect. The interfaces introduce types to be used within the pattern protocol
  - An abstract aspect can define the roles and attach default implementations where possible
  - The abstract aspect can define an abstract pointcut to capture join points that should trigger important events
  - Clients use a public method on the aspect to access the new functionality. The methods that are used on the participants can be introduced privately and only visible to the aspect

Vrije Universiteit Brussel

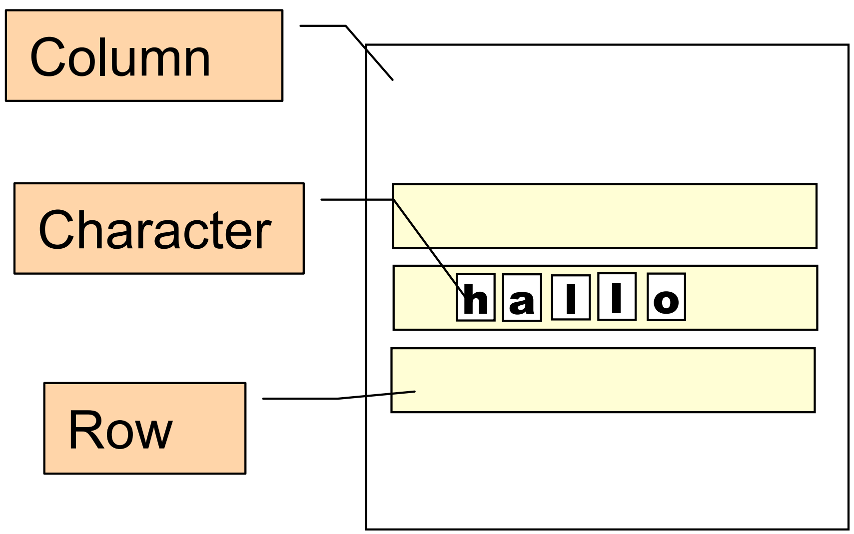
Singleton, Prototype, Memento, Iterator, Flyweight: **aspects as object factories**

- Patterns that administrate access to specific object instances. They offer factory methods to clients and share a create on demand strategy
  - These patterns have an abstracted (reusable) implementation in AspectJ with code for the factory in the aspect
  - The factory methods can be parameterized methods on the abstract aspect or methods introduced to the participants
  - In the former case, multiple pattern instances compose transparently, even if all factory methods have the same name
  - The singleton is special, the original constructor can be turned into the factory method using around advice to return the unique object on all constructor calls
  - Parameterized factory methods can be implemented by making the factory method return a null of default object and then have other objects returned by around advice on that method. This allows to extend the factory in terms of new products without changing the code

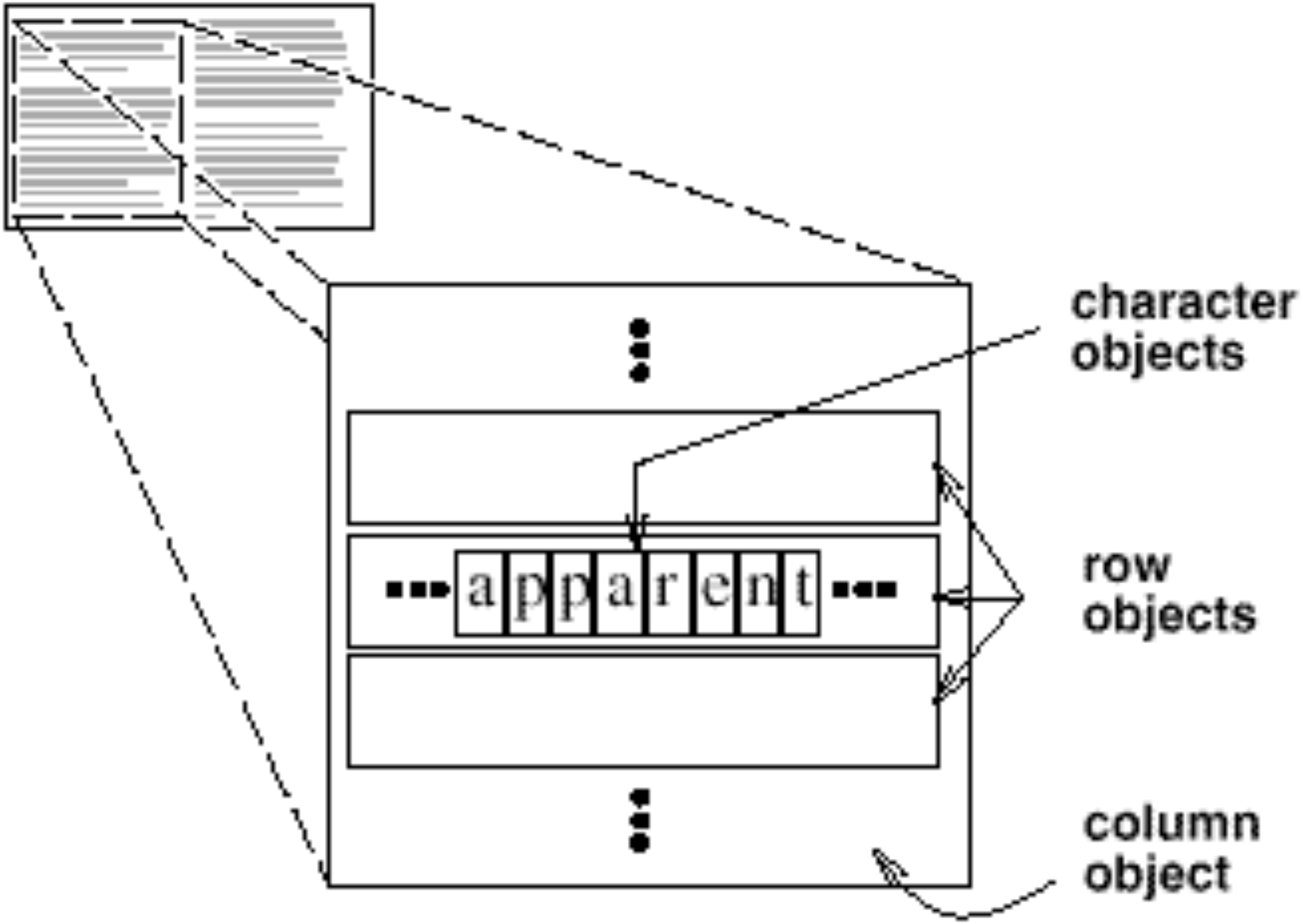
FLYWEIGHT

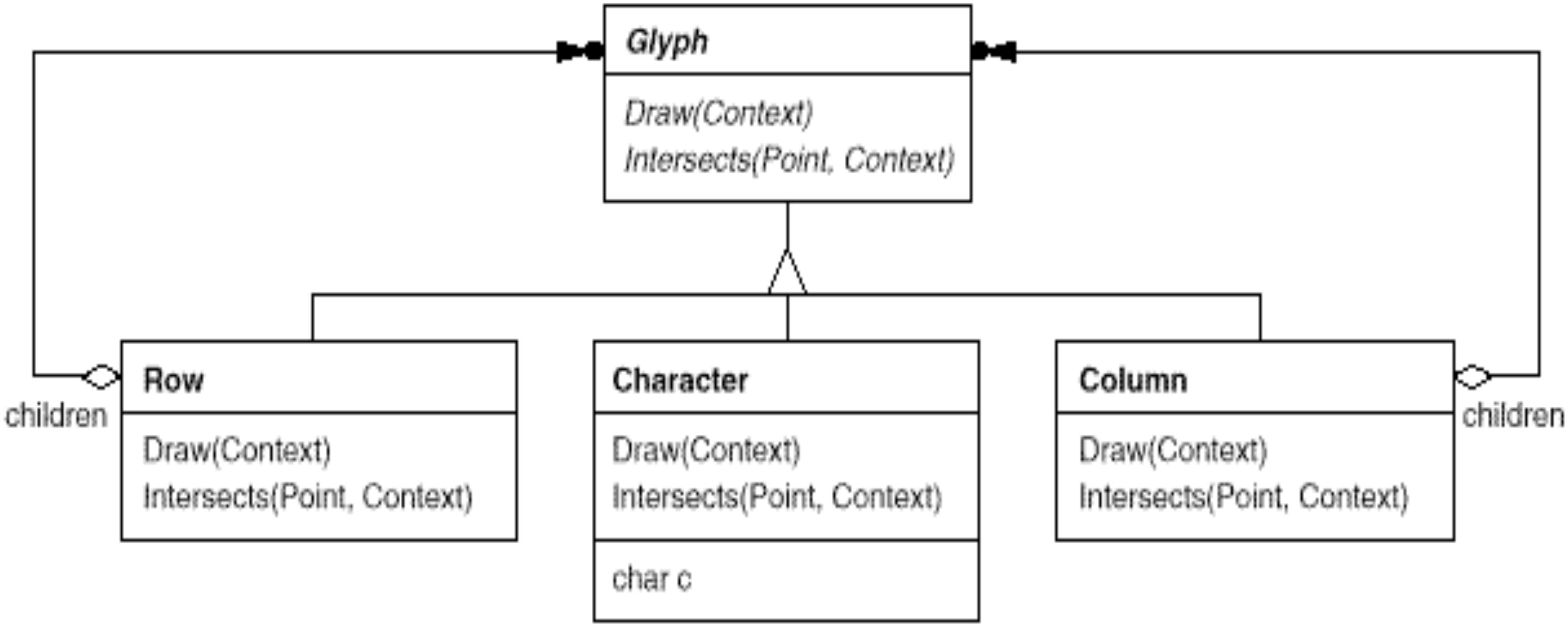
The Flyweight Pattern: The Problem

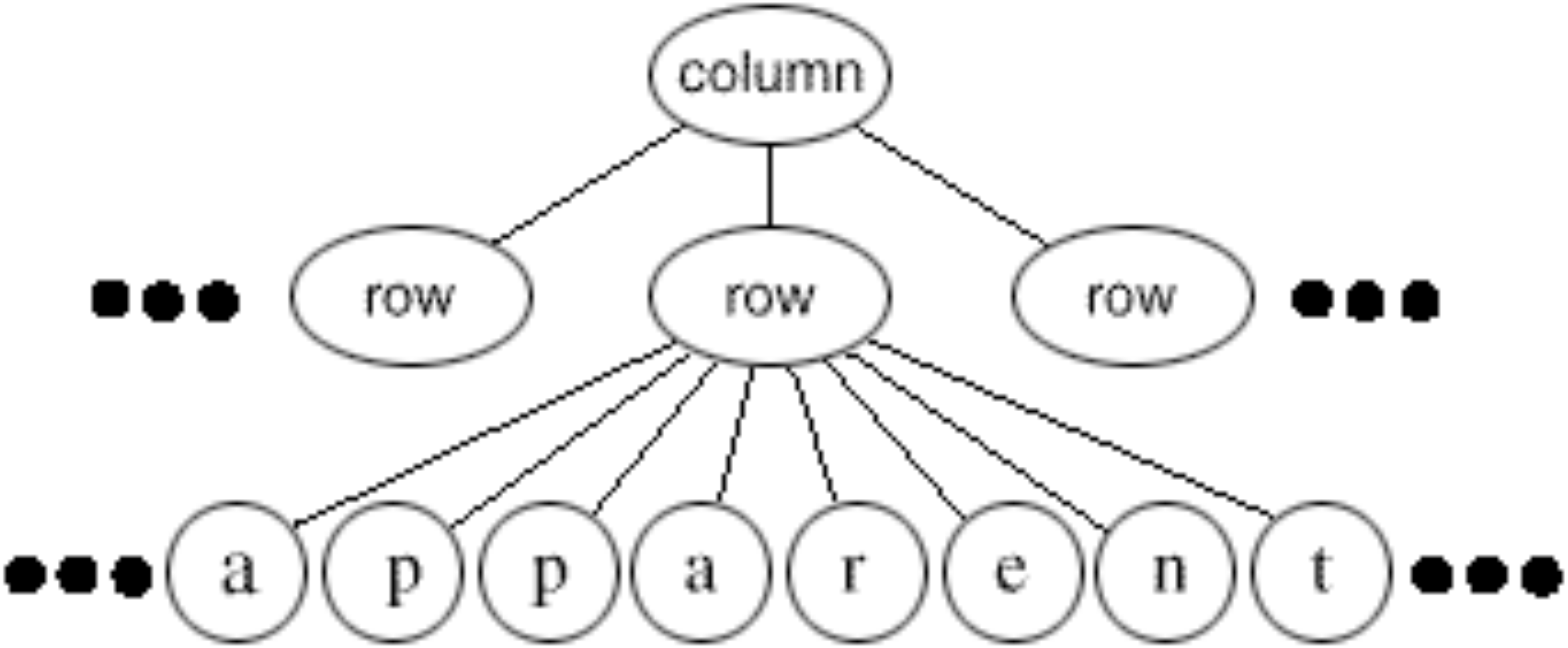
Some applications benefit from using objects in their design but a naïve implementation is prohibitively expensive because of the large number of objects

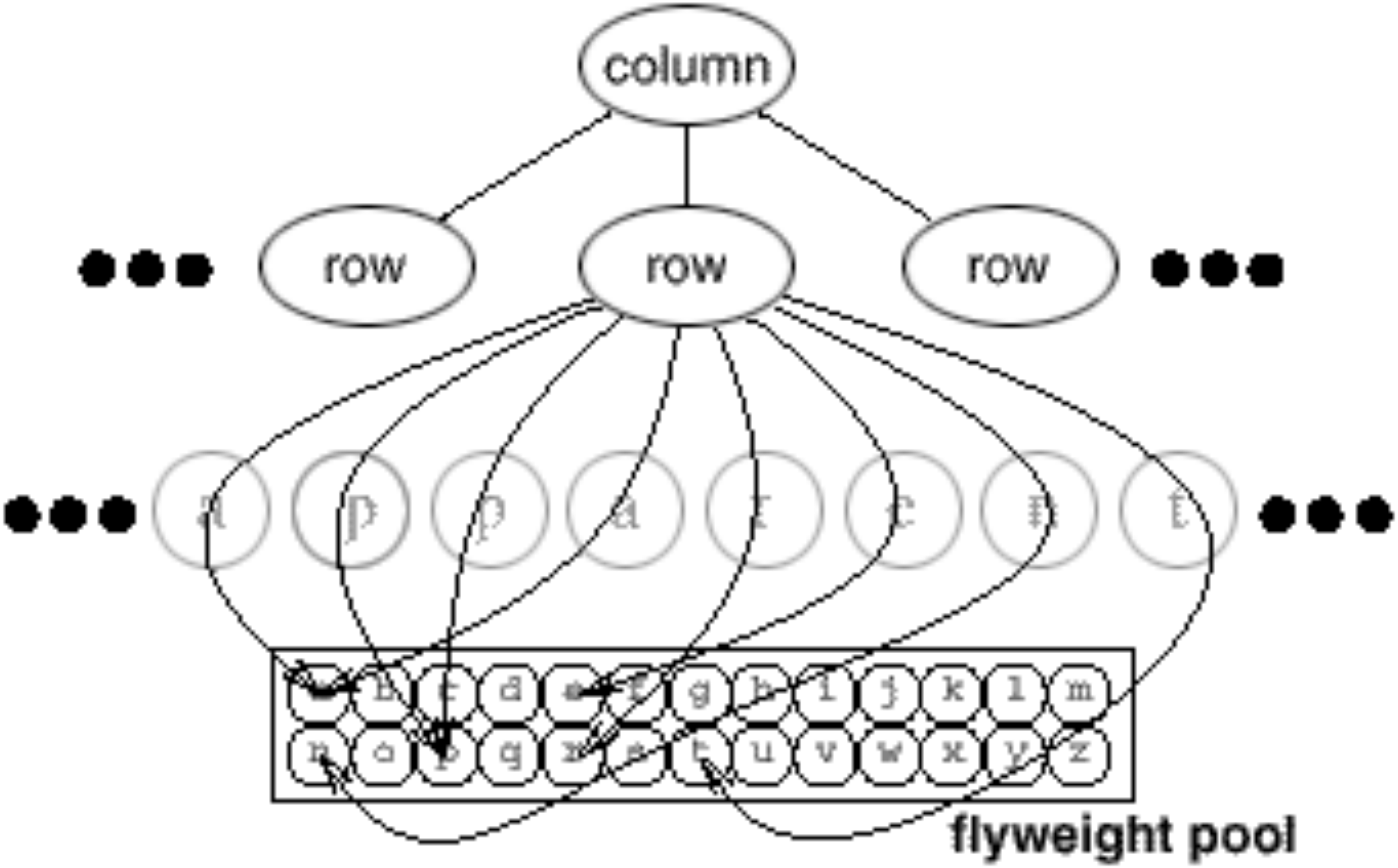


- use an object for each character in a text document editor
- use a layout object for each widget in a GUI


character objects
apparent
row objects
column object


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
children
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children


column
row
row
row
a
p
p
a
r
e
n
t


column
row
row
row
a
p
p
a
r
e
n
t
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
flyweight pool

The Flyweight Pattern Applicability

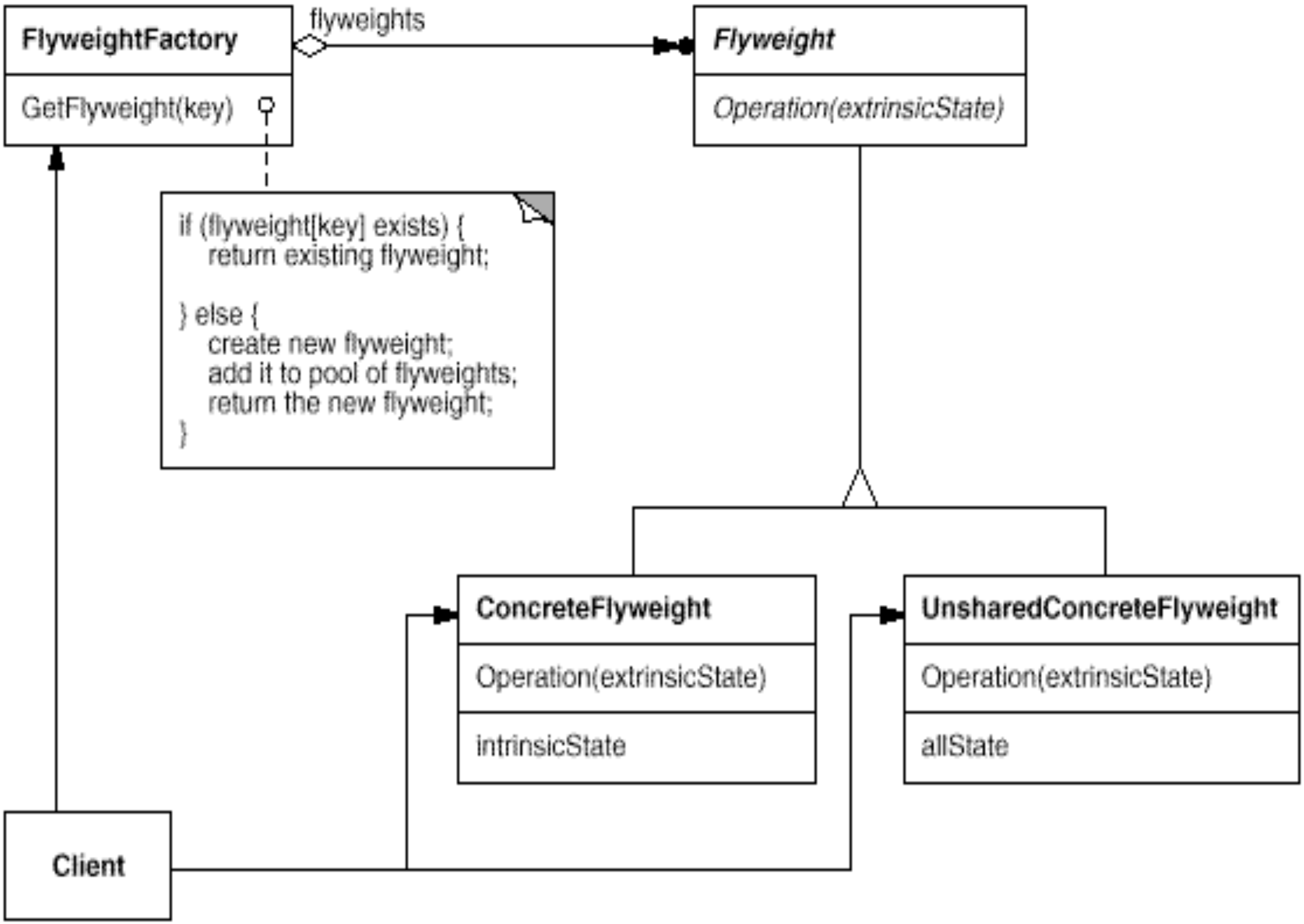
- Apply flyweight when ALL of the following are true:
  - An application uses a large number of objects
  - Storage cost is high because of the quantity of objects
  - Most objects can be made extrinsic
  - Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed
  - The application does not depend on object identity

The Flyweight Pattern Participants (1)

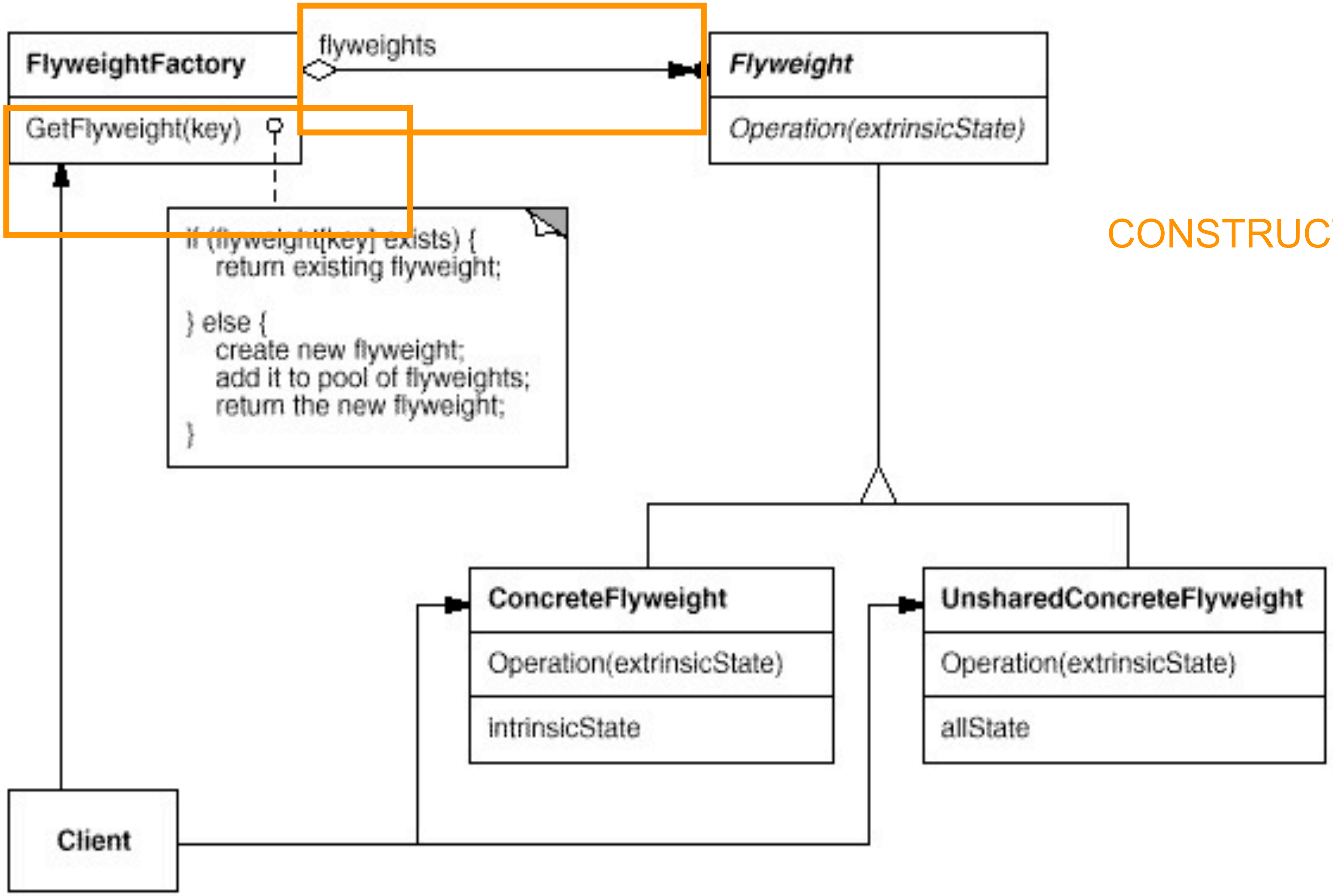
- Flyweight
  - Declares a n interface through which flyweights can receive and act upon extrinsic state
- Concrete Flyweight
  - Implements the flyweight interface and adds storage for intrinsic state
  - A concrete flyweight object must be sharable, i.e. all state must be intrinsic
- Unshared Concrete Flyweight
  - Not all flyweights subclasses need to be shared, unshared concrete flyweight objects have concrete flyweight objects at some level in the flyweight object structure

The Flyweight Pattern Participants (2)

- Flyweight Factory
  - Creates and manages flyweight objects
  - Ensures that flyweights are shared properly; when a client requests a flyweight the flyweight factory supplies an existing one from the pool or creates one and adds it to the pool
- Client
  - Maintains a reference to flyweight(s)
  - Computes or stores the extrinsic state of flyweight(s)


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Flyweight



CONSTRUCTION

The abstract flyweight aspect

```
public abstract aspect FlyweightProtocol {
   private Hashtable flyweights = new Hashtable();
   protected interface Flyweight{};
   protected abstract Flyweight createFlyweight(Object key);

   public Flyweight getFlyweight(Object key) {
       if (flyweights.containsKey(key)) {
            return (Flyweight) flyweights.get(key);
       } else {
            Flyweight flyweight = createFlyweight(key);
            flyweights.put(key, flyweight);
            return flyweight;
       }
   }
}
```

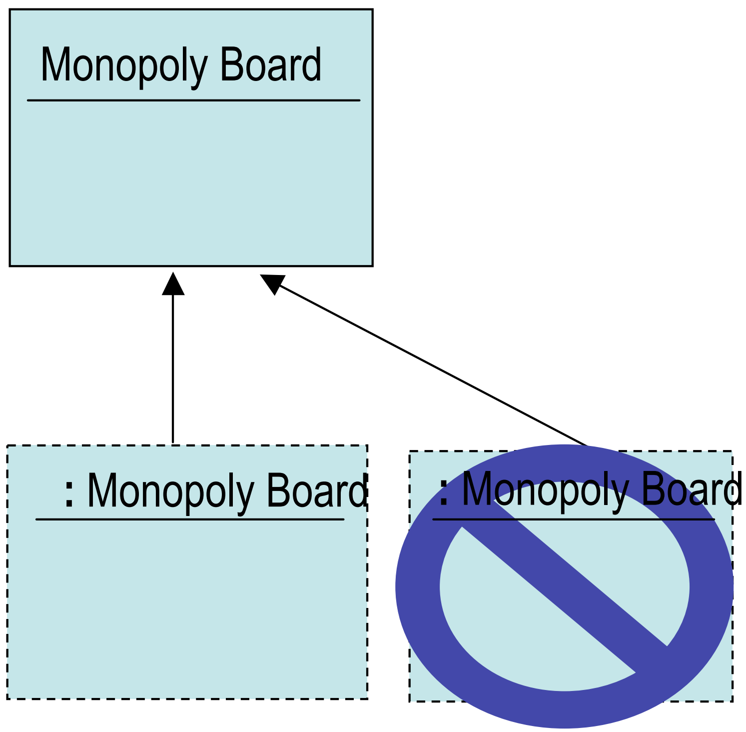
A concrete flyweight aspect

```
public aspect FlyweightImplementation extends FlyweightProtocol {
   declare parents: CharacterFlyweight implements Flyweight;
   protected Flyweight createFlyweight(Object key) {
        char c = ((Character) key).charValue();
        Flyweight flyweight = new CharacterFlyweight(c);
        return flyweight;
   }
}
public class CharacterFlyweight {
   private char c;
   public CharacterFlyweight(char c) {
        this.c = c;
   }
   public void print(boolean uppercase) {
        System.out.print(uppercase ? Character.toUpperCase(c) : c);
   }
}
```

SINGLETON

The Singleton Pattern: The Problem

Ensure that a class has exactly one instance and provide a global point of access to it



- There can be only one print spooler, one file system, one window manager in a standard application
- There is only one game board in a monopoly game; one maze in a maze-game

The Singleton Pattern Participant & Collaboration

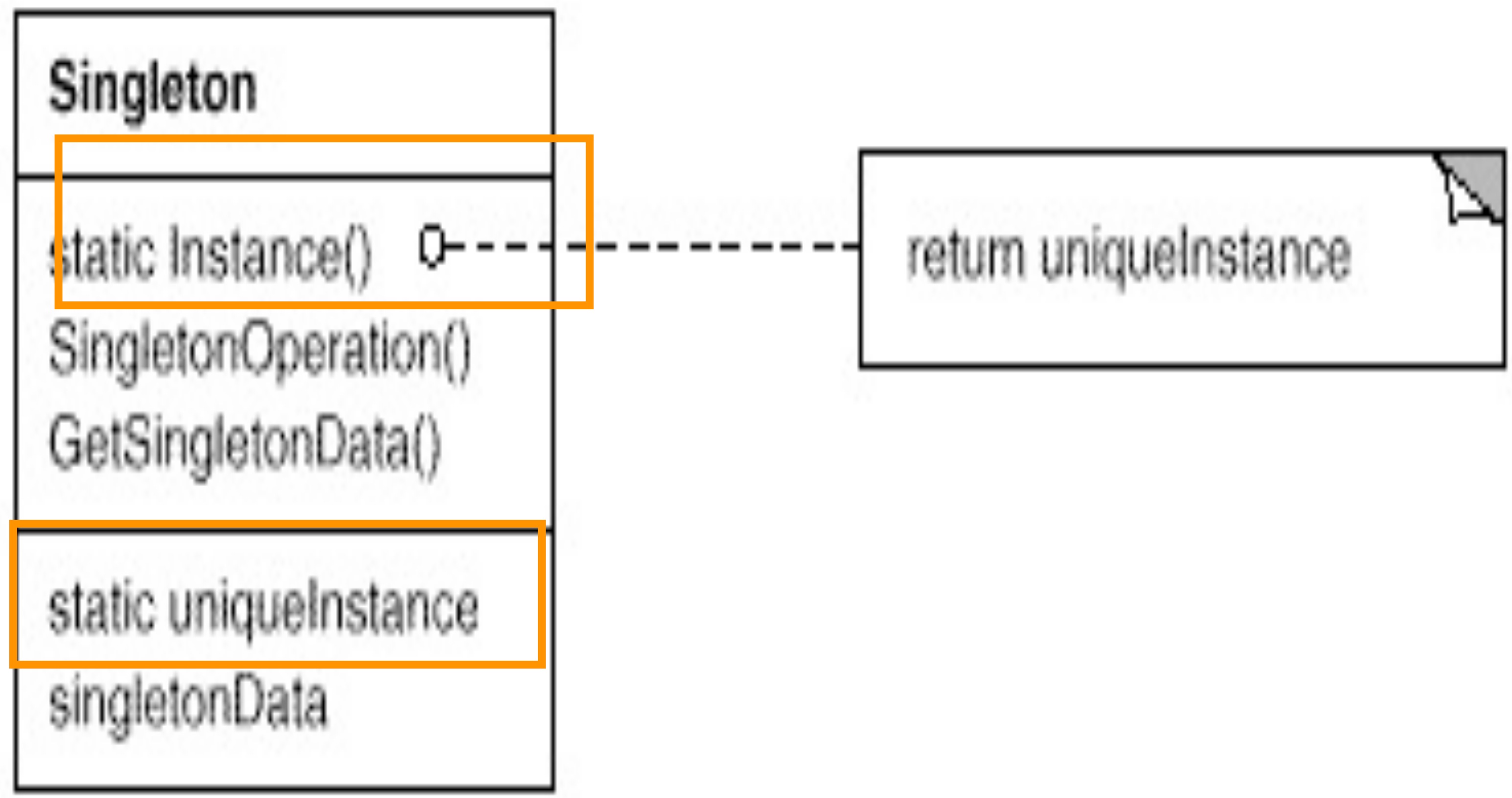
Participant:

- Singleton:
  - is responsible for creating and storing its own unique instance
  - defines an Instance operation that lets clients access its unique instance

Collaboration:

- the “class level” Instance operation will either return or create and return the sole instance; a “class level” attribute will contain either a default indicating there is no instance yet or the sole instance

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Abstract singleton aspect

```
public abstract aspect SingletonProtocol {

   public interface Singleton {}
   private Singleton the-singleton = null;

    Object around(): call((Singleton).new(..)) {
      if (the-singleton == null) {
         the-singleton = proceed();
      }
      return the-singleton;
      }
}
```

Concrete singleton aspect

```
public class Printer {
    protected static int objectsSoFar = 0;
    protected int id;
    public Printer() {
        id = ++ objectsSoFar;
    }
    public void print() {
        System.out.println("My ID is "+id);
    }
}

public aspect SingletonInstance extends SingletonProtocol {
    declare parents: Printer implements Singleton;
}
```

Vrije Universiteit Brussel

Adapter, Decorator, Strategy, Visitor, Proxy: **language constructs**

- The implementation of these patterns can (partially) disappear because AspectJ language constructs implement them directly
- Examples:
  - Visitor and Adapter (Wrapper) can be realised by extending the interface of the ConcreteElements or the Adaptee with intertype declarations
  - Delegation and protection proxy's can have alternate implementations based on attaching advice. (This can not be done for remote and virtual proxy's since the Proxy and Subject need to be distinct objects in these cases)
- These altermatives are often more modular and simple but less flexible
  - The interface adaptation in Adapter cannot be realised if an existng method must be replaced by one with the same argument signature but a different return type
  - Dynamic reordering of decorators is not possible with advice based implementations of Decorator