Components and Aspect-Oriented Design/Programming

Mira Mezini, David Lorenz and

Karl Lieberherr

Overview

- Our abstract component definition
- Problems with structuring software - function versus object structuring
- Reconciliation of both worlds: Aspectual components as the component construct
- Aspectual components for generic higher-level collaborative behavior
- Aspectual components and Aspect-Oriented Programming (AOP)
- Summary

What is a component?

any identifiable slice of functionality that describes a meaningful service, involving, in general, several concepts,

- with well-defined expected and provided interfaces,
- formulated for an ideal ontology - the expected interface
- subject to deployment into several concrete ontologies by 3rd parties
- subject to composition by 3rd parties
- subject to refinement by 3rd parties

An ontology is, in simple terms, a collection of concepts with relations among them plus constraints on the relations.

Component deployment/composition

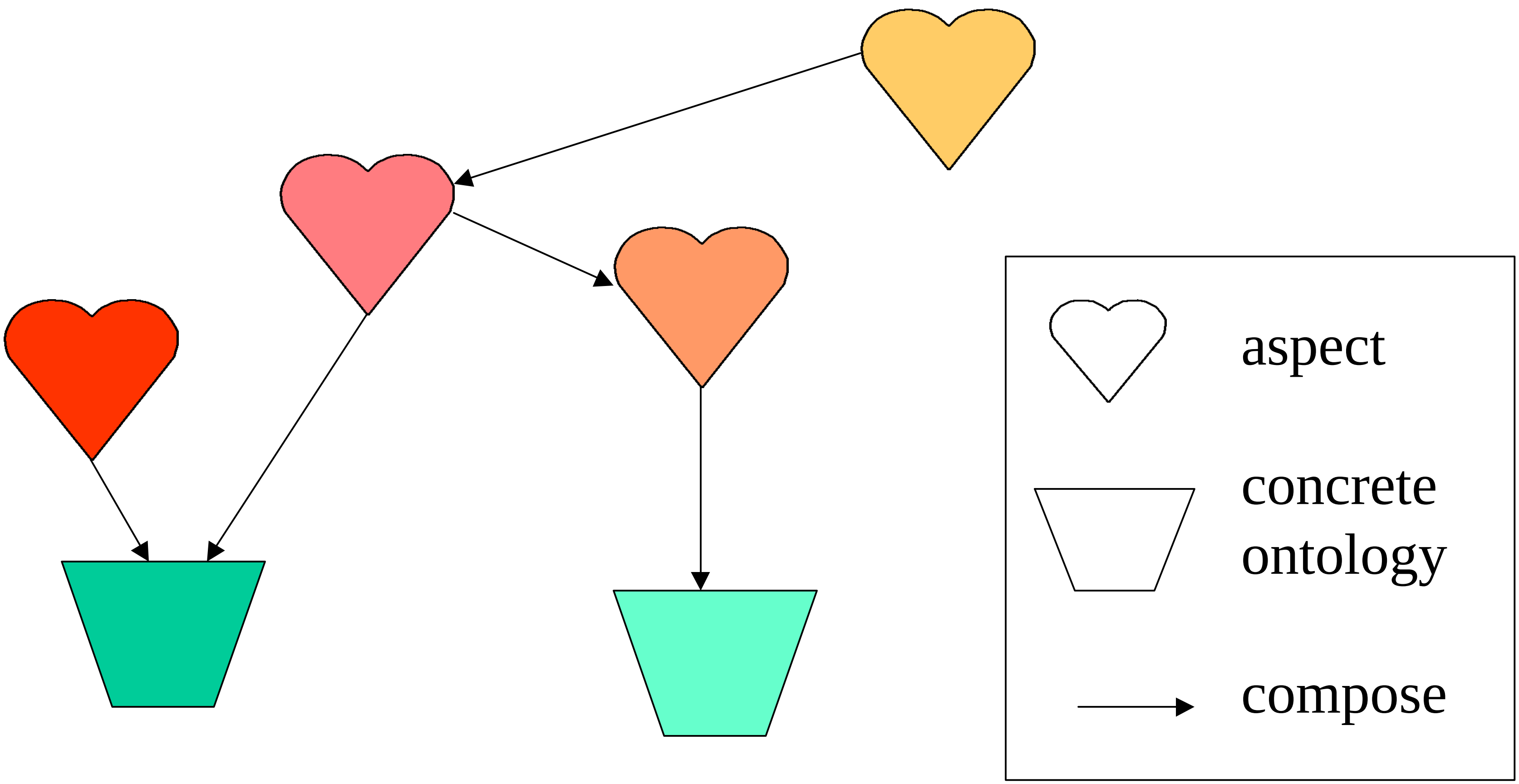
- Deployment is mapping idealized ontology to concrete ontology
  - specified by connectors separately from components
  - without mentioning irrelevant details of concrete ontology in map to keep deployment flexible
  - non-intrusive, parallel, and dynamic deployment
- Composition is mapping the provided interface of one (lower-level) component to the expected interface of another (higher-level) component
- deployment is a special case of composition, where the lower level component is a concrete ontology (no expected interface)

Graph of components

a directed graph

- nodes are components
- edges denote composition of components
- must be acyclic
- components without outgoing edges form the concrete ontology
- components with outgoing edges are called aspects (meaning both application and system level aspects of a software)

Graph of components



The goal

The goal is to separate concerns (each decision in a single place) and minimize dependencies between them (loose coupling):

- less tangled code, more natural code, smaller code
- concerns easier to reason about, debug and change
- a large class of modifications in the definition of one concern has a minimum impact on the others
- more reusable, can plug/unplug as needed

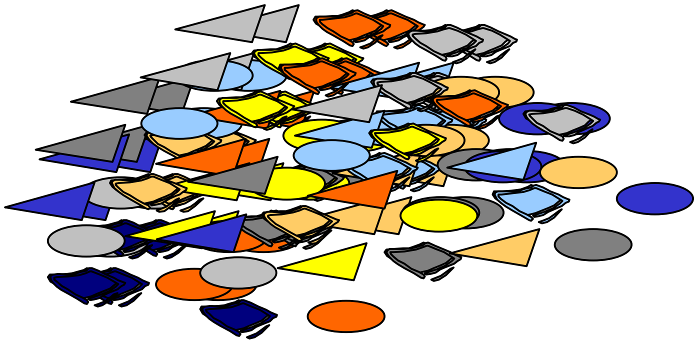
Problems with Software Structuring

Software =

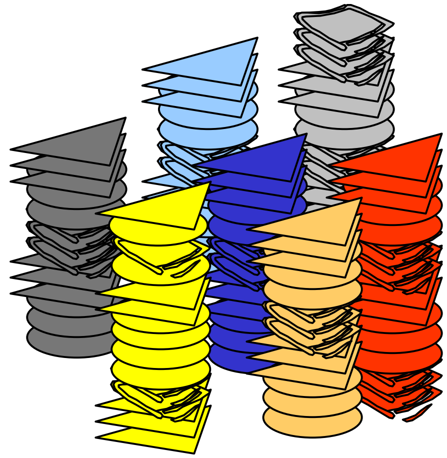
Data (Shapes)

\+

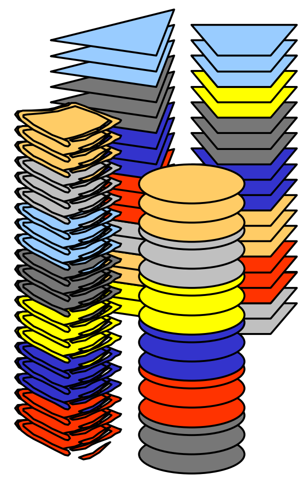
Functions (Colors)



1st Generation
Spaghetti-Code



2nd & 3rd Generation :
functional decomposition



4th Generation
object decomposition

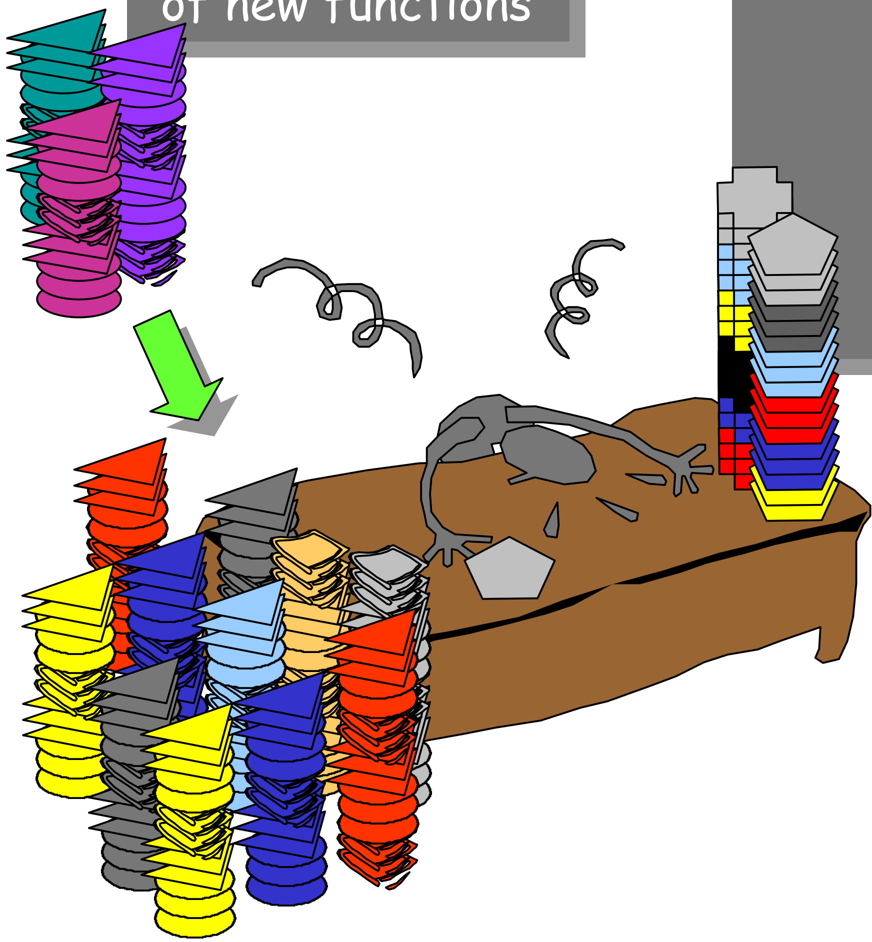
Problems with Functional Decomposition

Advantage: easy integration of new functions

Disadvantage: Data spread around

- integration of new data types ==> modification of several functions
- functions tangled due to use of shared data

Difficult to localize changes !



Problems with Object Decomposition

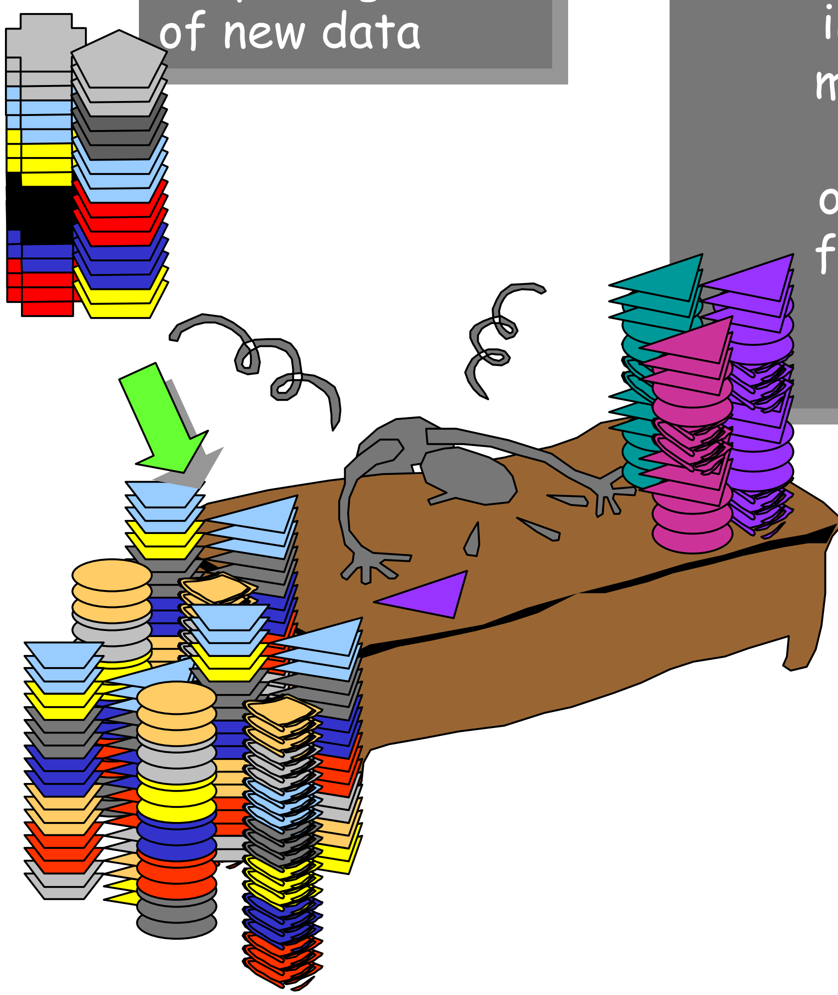
<u>Advantage:</u>
easy integration of new data

<u>Disadvantage:</u> functions spread around

integration of new functions ==> modifikation of several objects

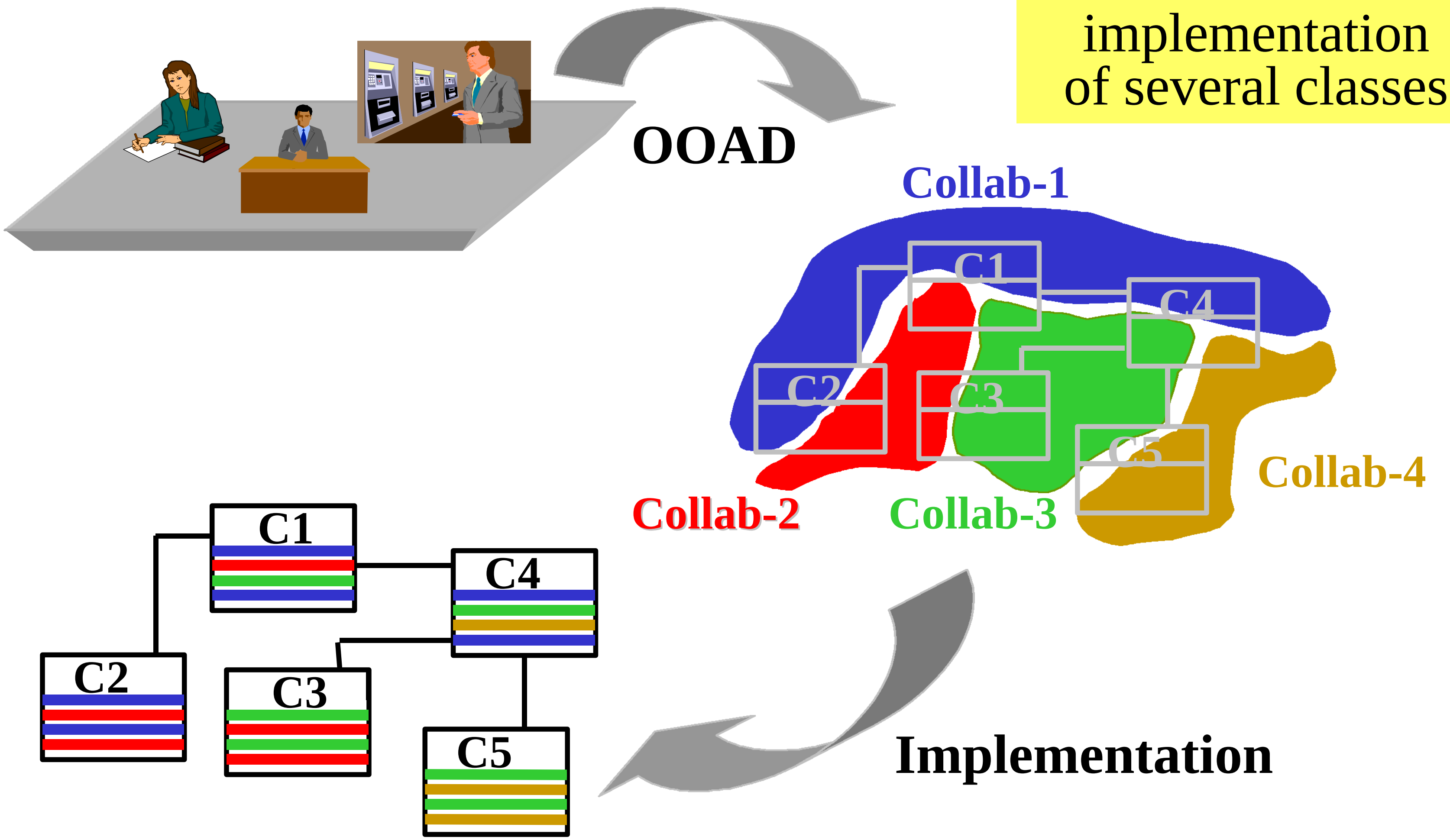
objects tangled due to higher-level functions involving several classes

Difficult to localize changes !

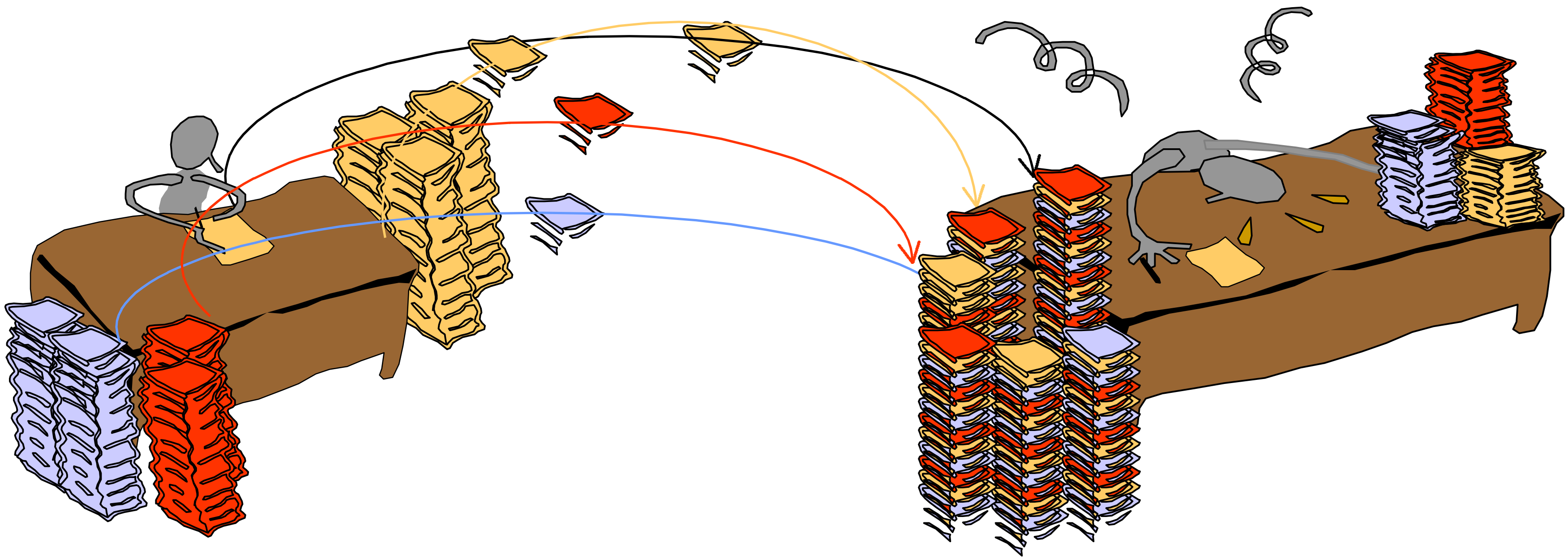


Problems with Object Decomposition

high-level behavior
scattered around the
implementation
of several classes



Problems with Object Decomposition

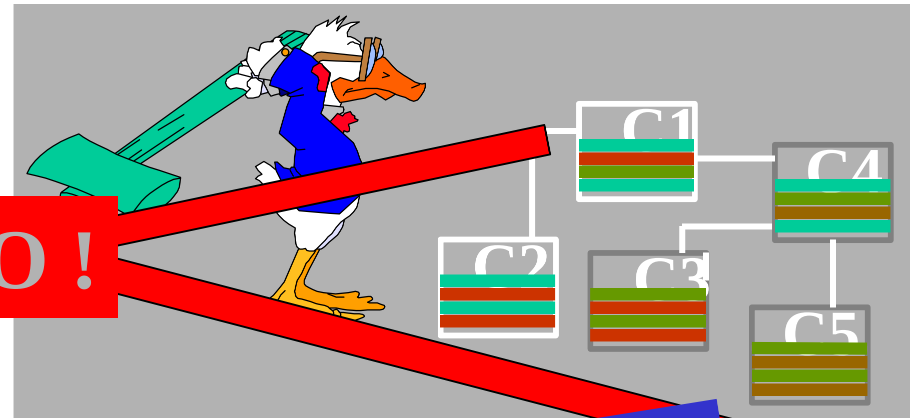


During implementation separate higher-level functions are mixed together

During maintenance/evolution individual collaborations need to be factored out of the tangled code

So what?

“Forget about objects”
[Udell, BYTE, May 94]

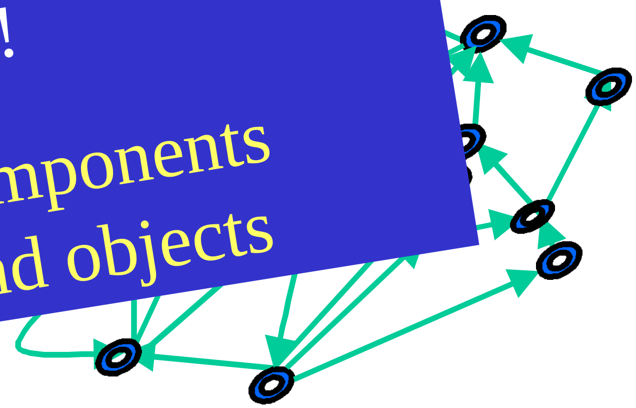


NO !

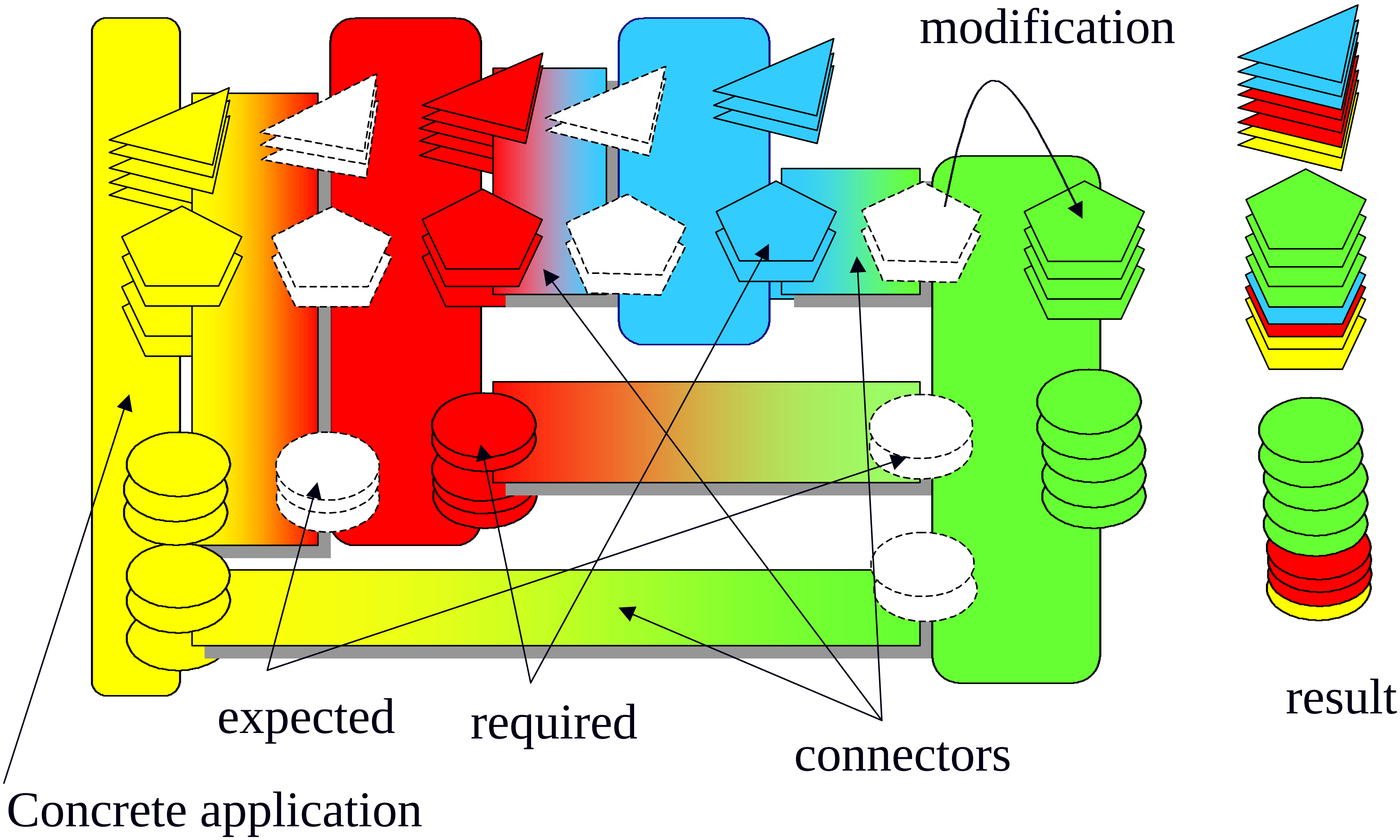
The
low-
princ
“hype

So, let’s organize!! Let’s have component constructs that capture functions cross cutting class boundaries !!

Let’s have Aspectual Components to reconcile functions and objects



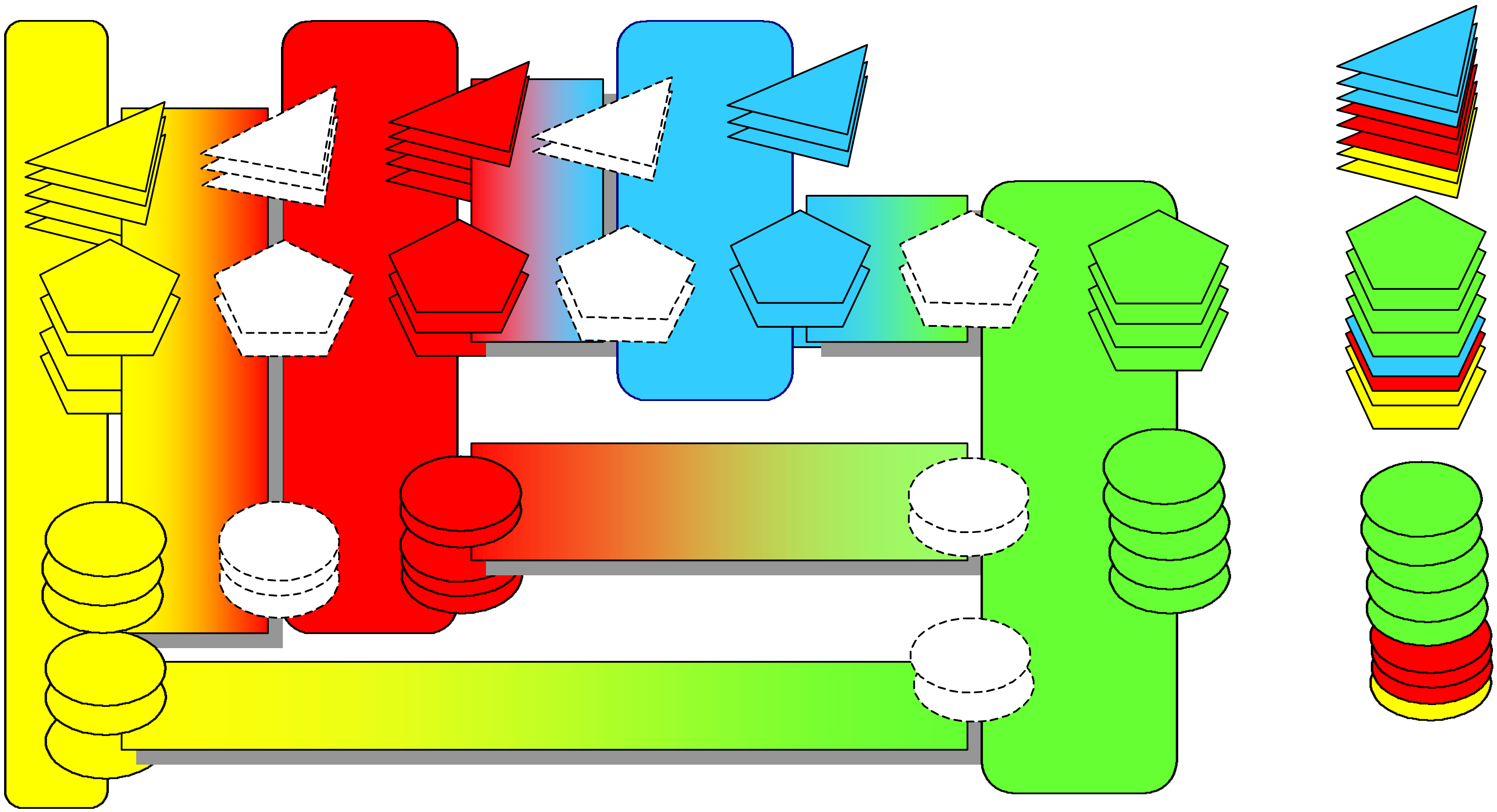
Reconciling objects and functions: the intuition behind aspectual components

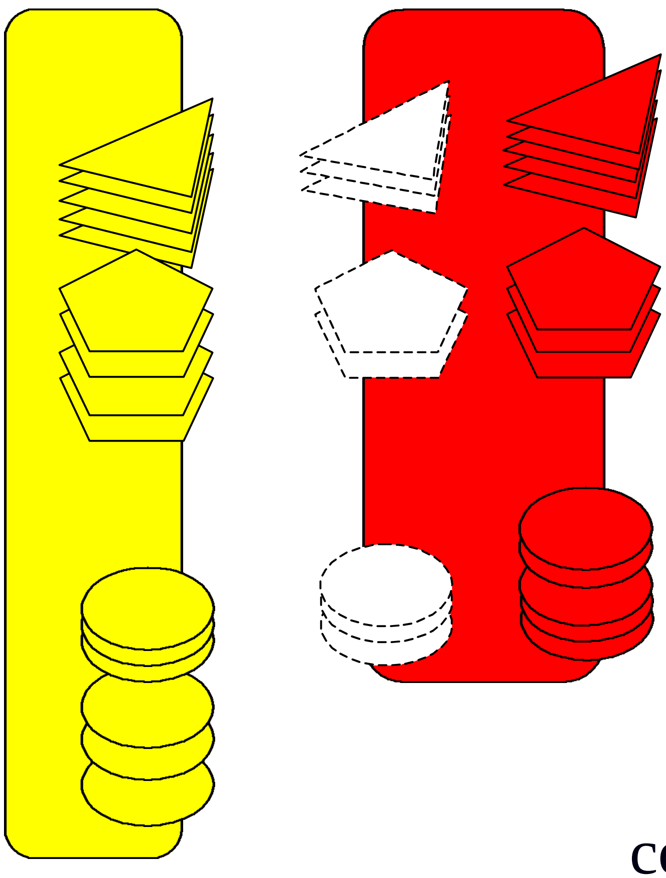


Aspectual component

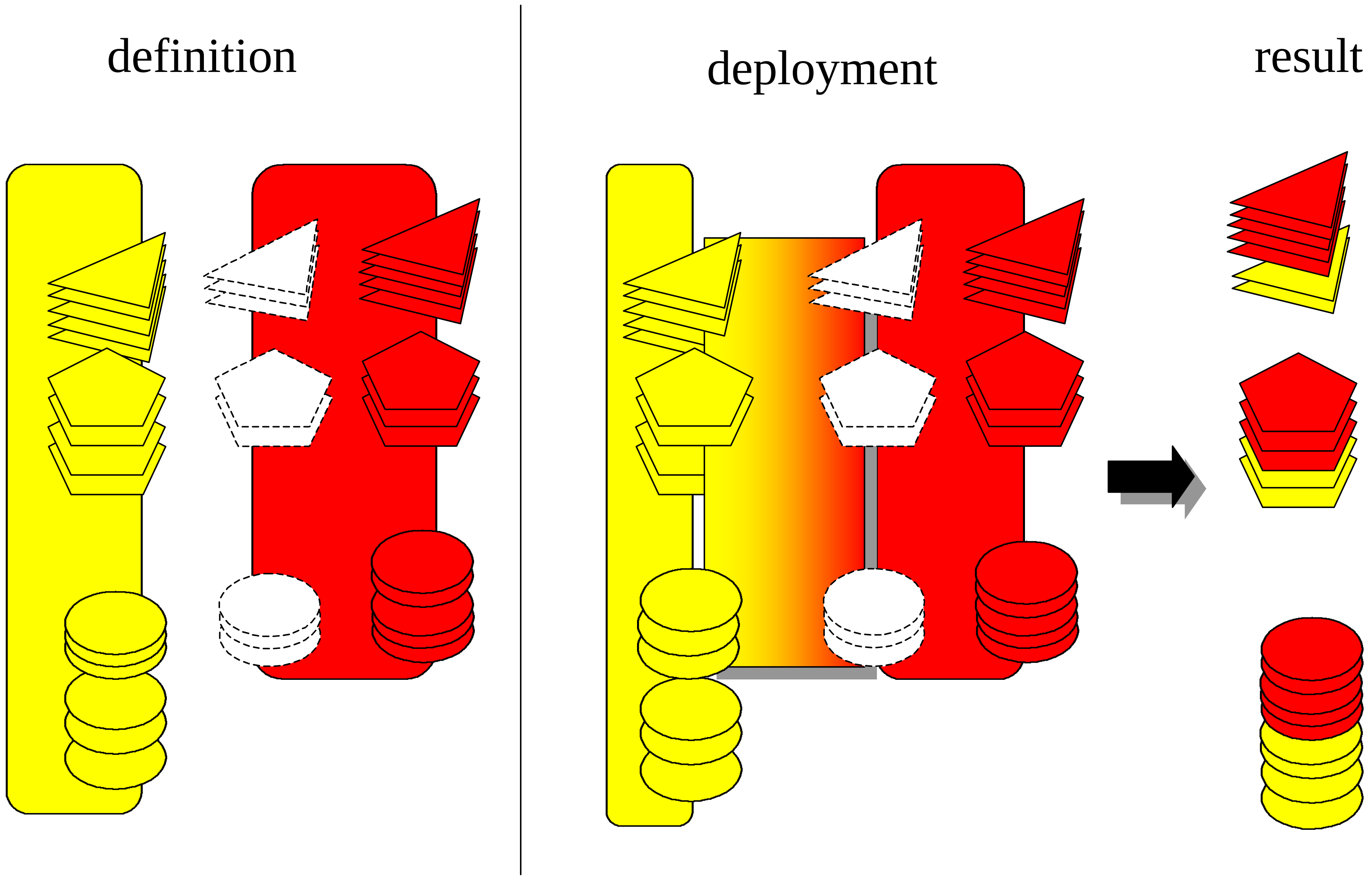
- Why not just “component”?
- “Aspectual” is not an English word.
- We want to distinguish between components that enhance and cross-cut other components and components that only provide new behavior.

Reconciling objects and functions: the intuition behind aspectual components

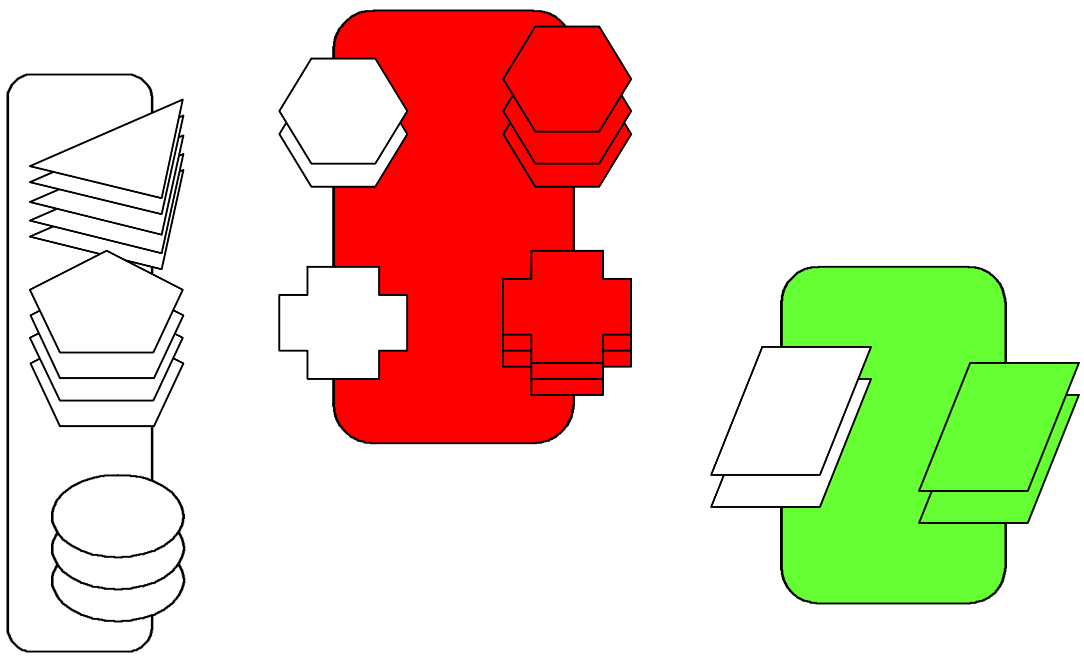


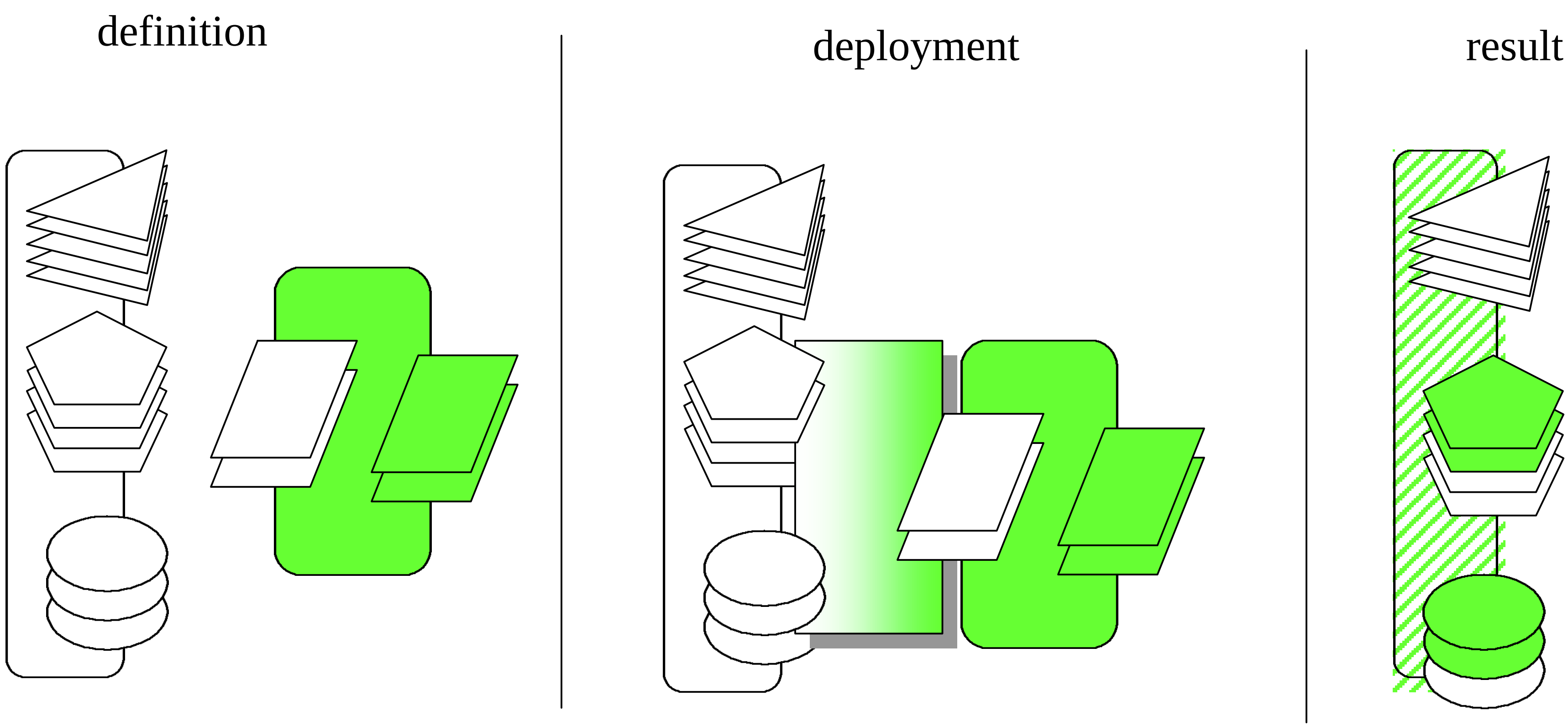


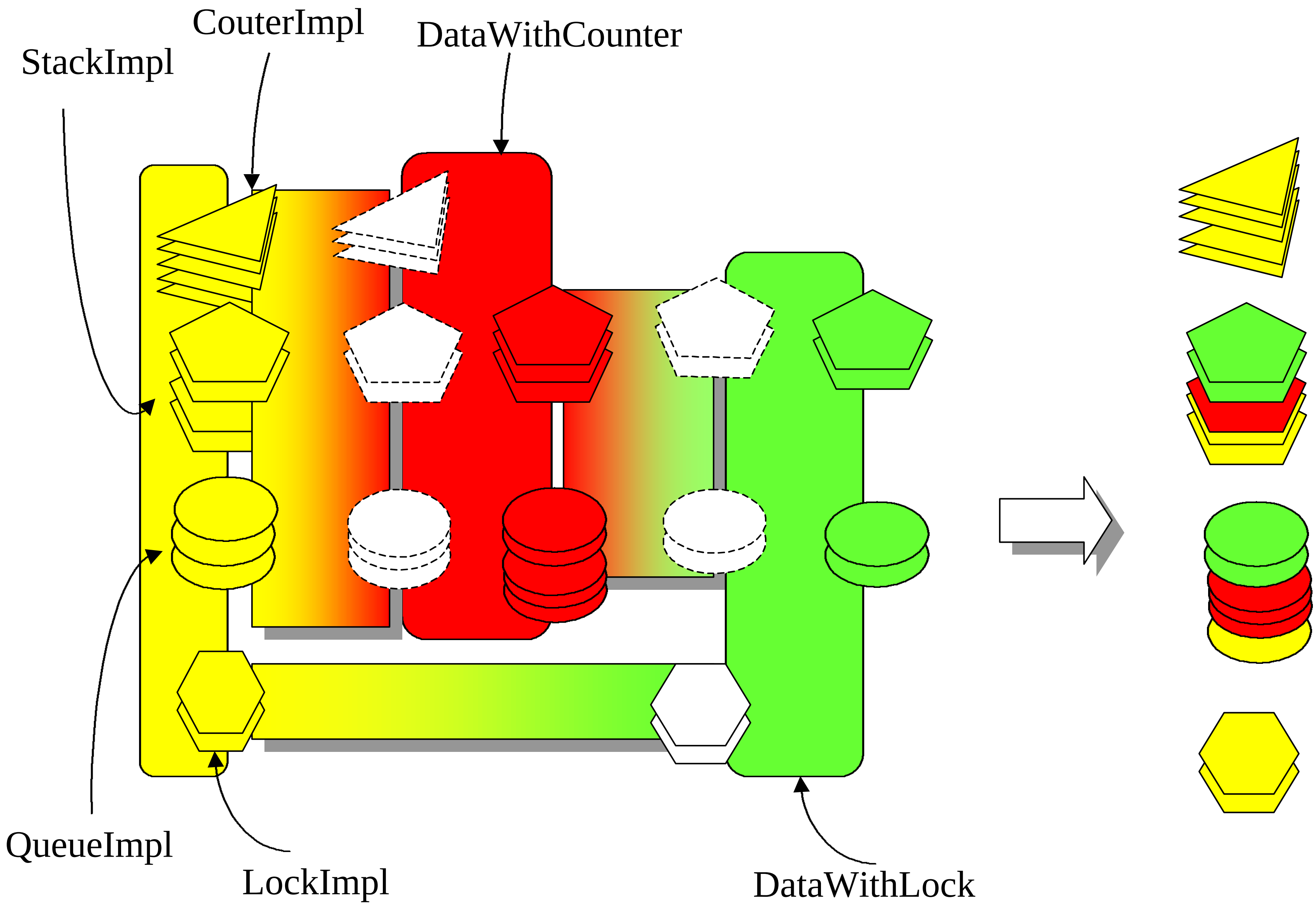
components

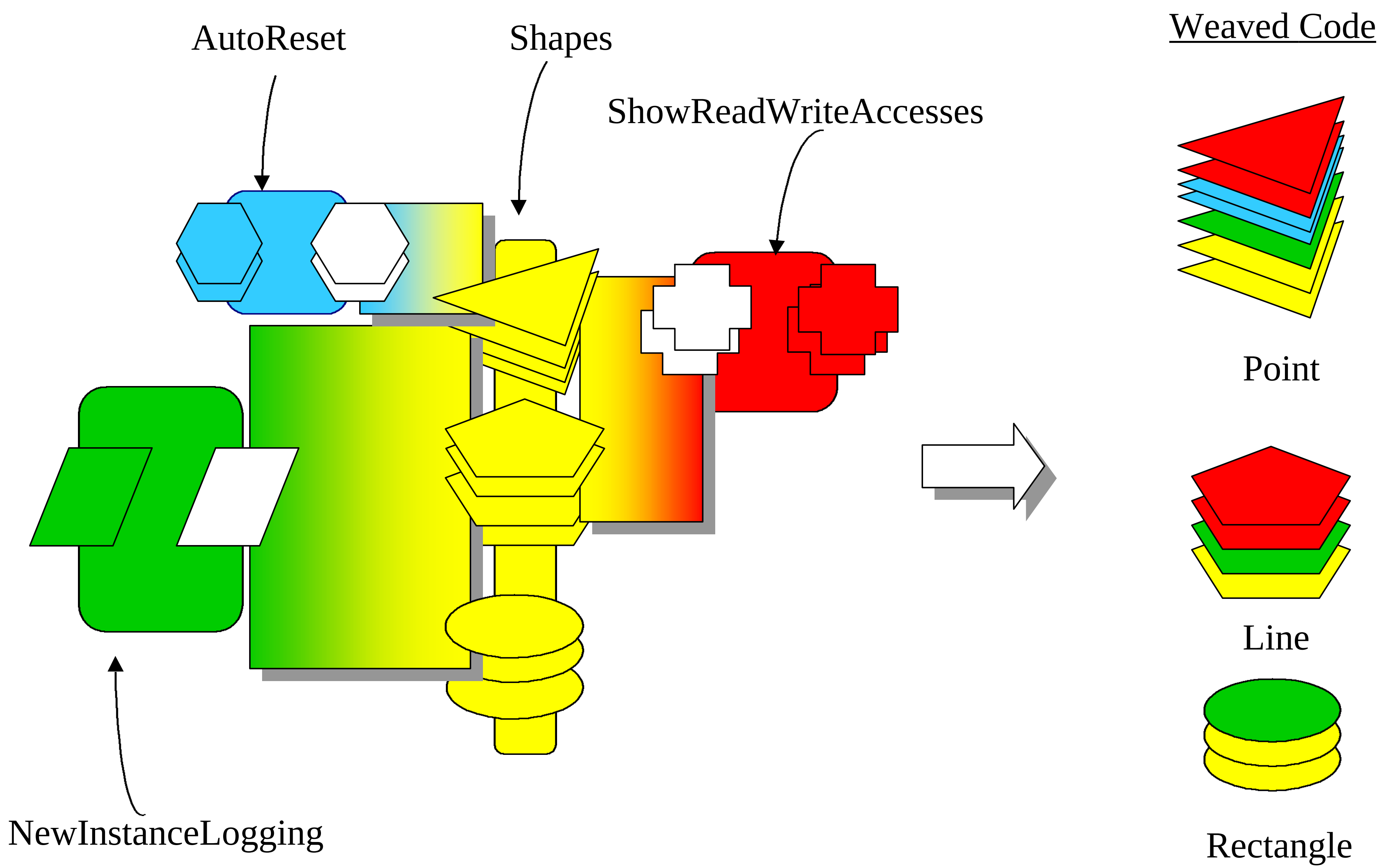


definition
deployment
result

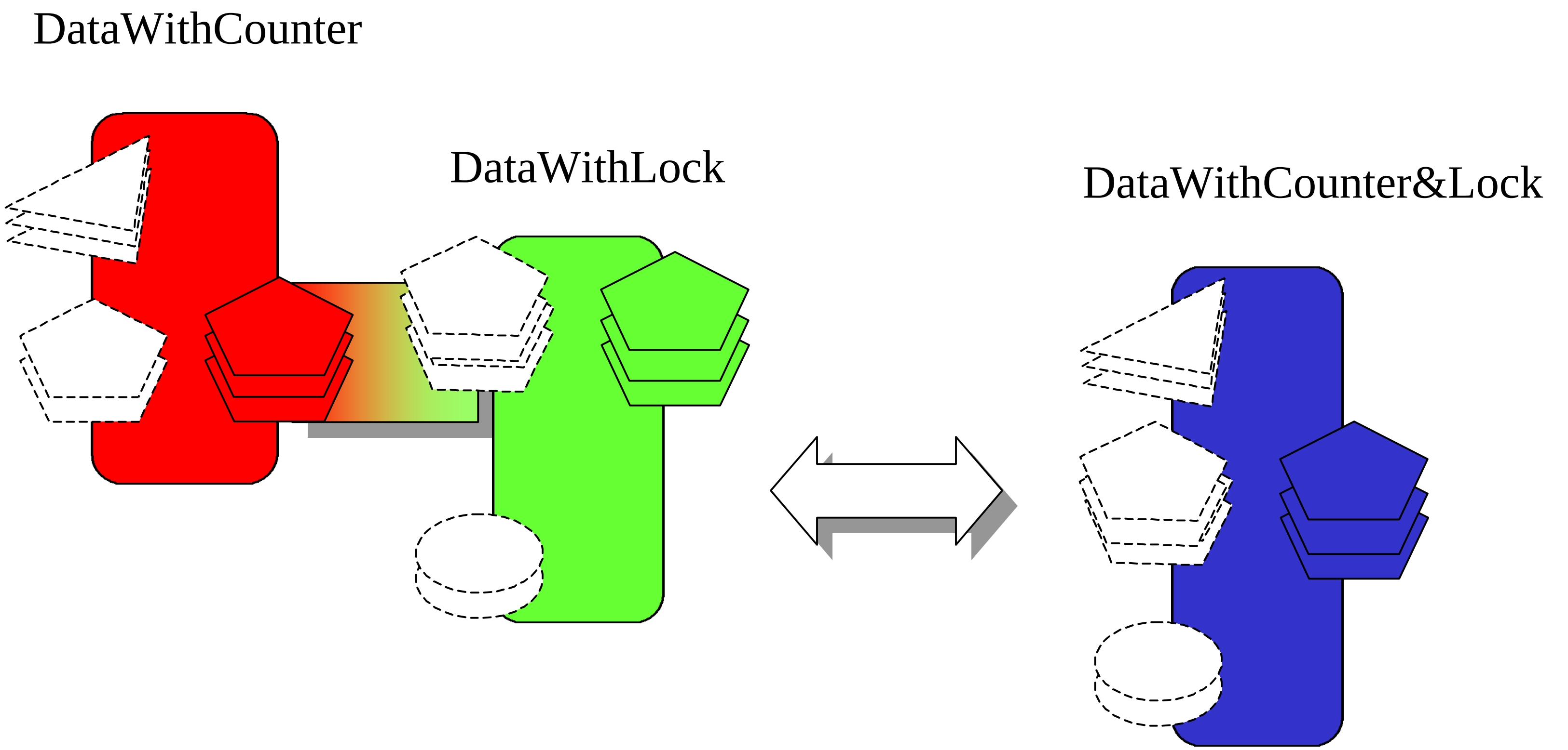



definition
deployment
result


CouterImpl
DataWithCounter
StackImpl
QueueImpl
LockImpl
DataWithLock



AutoReset
Shapes
ShowReadWriteAccesses
NewInstanceLogging
Weaved Code
Point
Line
Rectangle



DataWithCounter
DataWithLock
DataWithCounter&Lock

What is an aspect?

- A slice of high-level, system/application level functionality. Slice: not self-contained.
- High-level: three meanings
  - multi-party functionality involving several participants
  - one participant may be mapped to a set of otherwise not structurally related classes
  - two neighboring participants may be mapped to classes that are “far apart” (many intermediate classes)
- Aspect cross-cuts object structure.

Examples

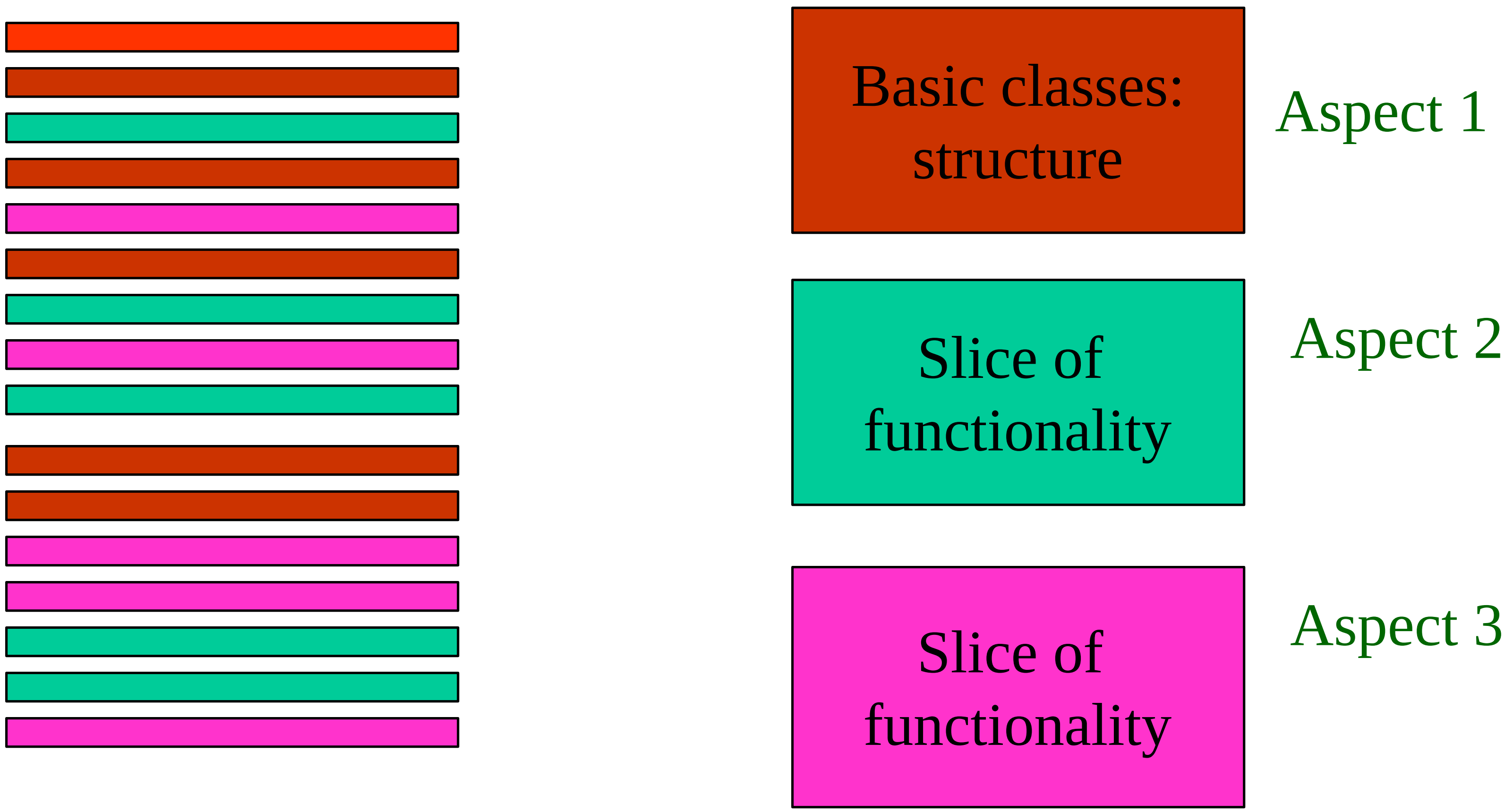
- Publisher-subscriber protocol: it applies in general to multiple sets of classes in different places in a system's object structure.
- Logging execution behavior
- Synchronization

Need a construct to express aspects

- Otherwise have tangled code. Would have to spread fragments of aspect definition manually.
- Resulting in tangled code. Need to control tangling (cannot eliminate it)
- Solution: aspectual components

Cross-cutting of aspects

ordinary program better program



Informal aspect description: ShowReadAccess

``For any data type in an application, say DataToAccess, any read access operation, AnyType readOp() defined for DataToAccess, and any invocation of this operation on an instance of DataToAccess, dataInstance, display Read access on <string representation of dataInstance>}´´.

Example of an aspectual component for ShowReadAccess

```
component ShowReadAccess {
  participant DataToAccess {
    expect Object readOp();
    replace Object readOp() {
     System.out.println("Read access on "
       + this.toString());
     return expected(); // this calls the
        // expected version of readOp()
    }
  }
```

Concrete class graph: in Java

```
class Point {
    private int x = 0;
    private int y = 0;
    void set(int x,int y) {this.x = x;this.y = y;}
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
    int getX(){ return this.x; }
    int getY(){ return this.y; }
  }
  class Line { ... }
  class Rectangle {... }
```

Deployment

```
connector  ShowReadAccessConn1 {
   Point is ShowReadAccess.DataToAccess
     with {readOp = get*};
  }
connector ShowReadAccessConn3 {
   {Point, Line, Rectangle}
   is ShowReadAccess.DataToAccess
     with {readOp = get*; }
}
```

Inheritance between components

```
component ShowReadWriteAccess extends
  ShowReadAccess {
  participant DataToAccess {
     expect void writeOp(Object[] args);
     replace void writeOp(Object[] args){
        System.out.println(
        "Write access on " +
            this.toString());
        expected(args);}}
 }
```

Inheritance between connectors

```
connector ShowReadWriteAccessConn2
  extends ShowReadAccessConn3  {
   {Point,Line,Rectangle}
    is DataToAccess with {
      writeOp = set*;
   }
```

Components have flavor of classes

- Common
  - Have local data and function members
  - One component can inherit from another component
- Different
  - component/connector separation. Component adaptation code is not part of application.

What are aspectual components?

- Aspectual components are language constructs that capture behaviour involving several classes (cross-cuts class boundaries)
- the programmer uses classes to implement the primary data (object) structure
- the programmer uses aspectual components to implement higher-level behavior cross-cutting the primary structure in a modular way

What are aspectual components?

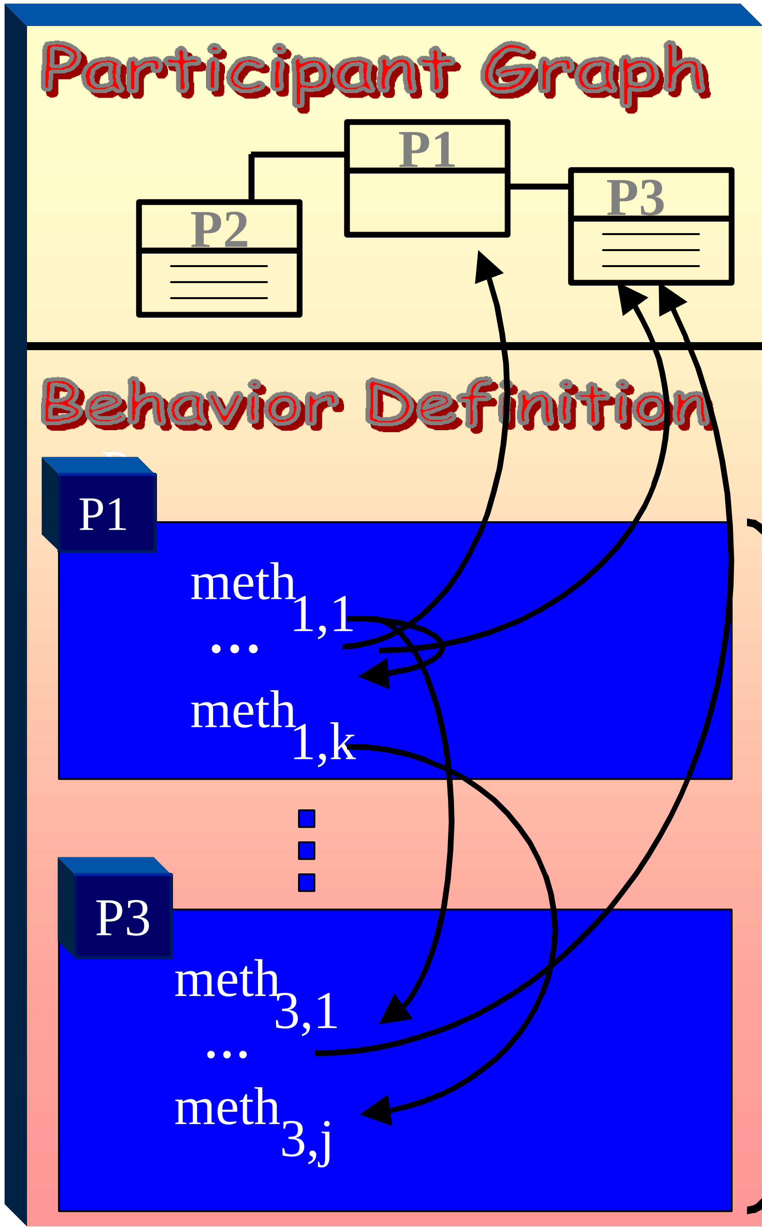
- Aspectual components have provided and expected interfaces
- The expected interface consists of an ideal class graph (Participant Graph, PG) to enable defining one aspect of the system with limited knowledge about the object model and/or other aspects defined by other components
- Aspectual components can be deployed into PGs or concrete class graphs and/or composed/refined by 3rd parties (reuse) by mapping interfaces via explicit connectors

Aspectual Components (AC)

minimal assumptions on application structure

+

expected interfaces



written to the PG similar to an OO program is written to a concrete class graph

add new functionality
+
enhance the expected

provided
=
everything declared public

Aspectual Component Def.

- A set of participants forming a graph called the participant graph (represented by a UML class diagram). Participant
  - formal argument to be mapped
  - expects function members (keyword **`expect`**)
  - reimplementations (keyword **`replace`**)
  - local data and function members

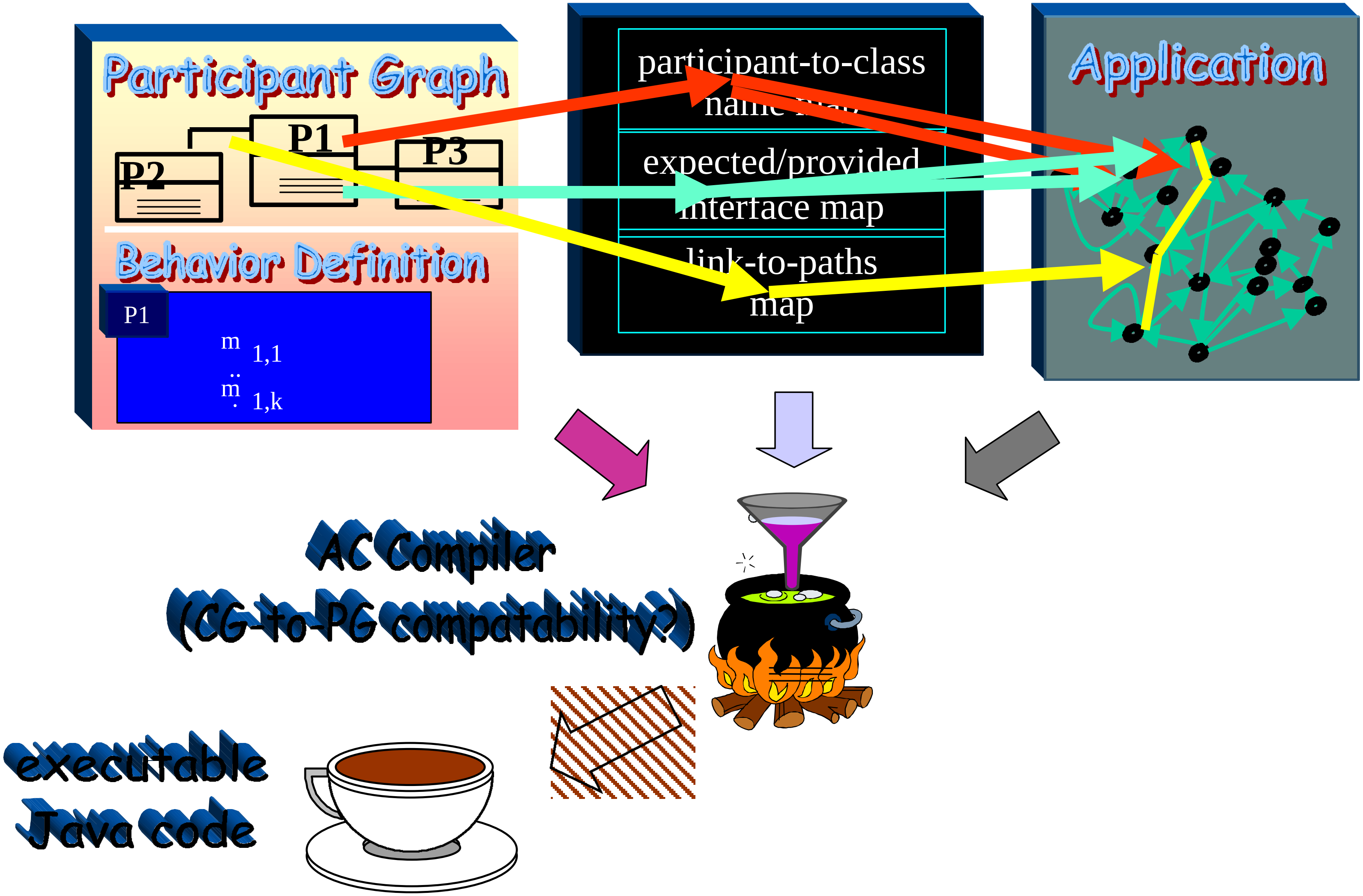
Aspectual Component Def. (continued)

- Local classes: visibility: aspectual component
- Aspectual component-level data and function members. There is a single copy of each global data member for each deployment

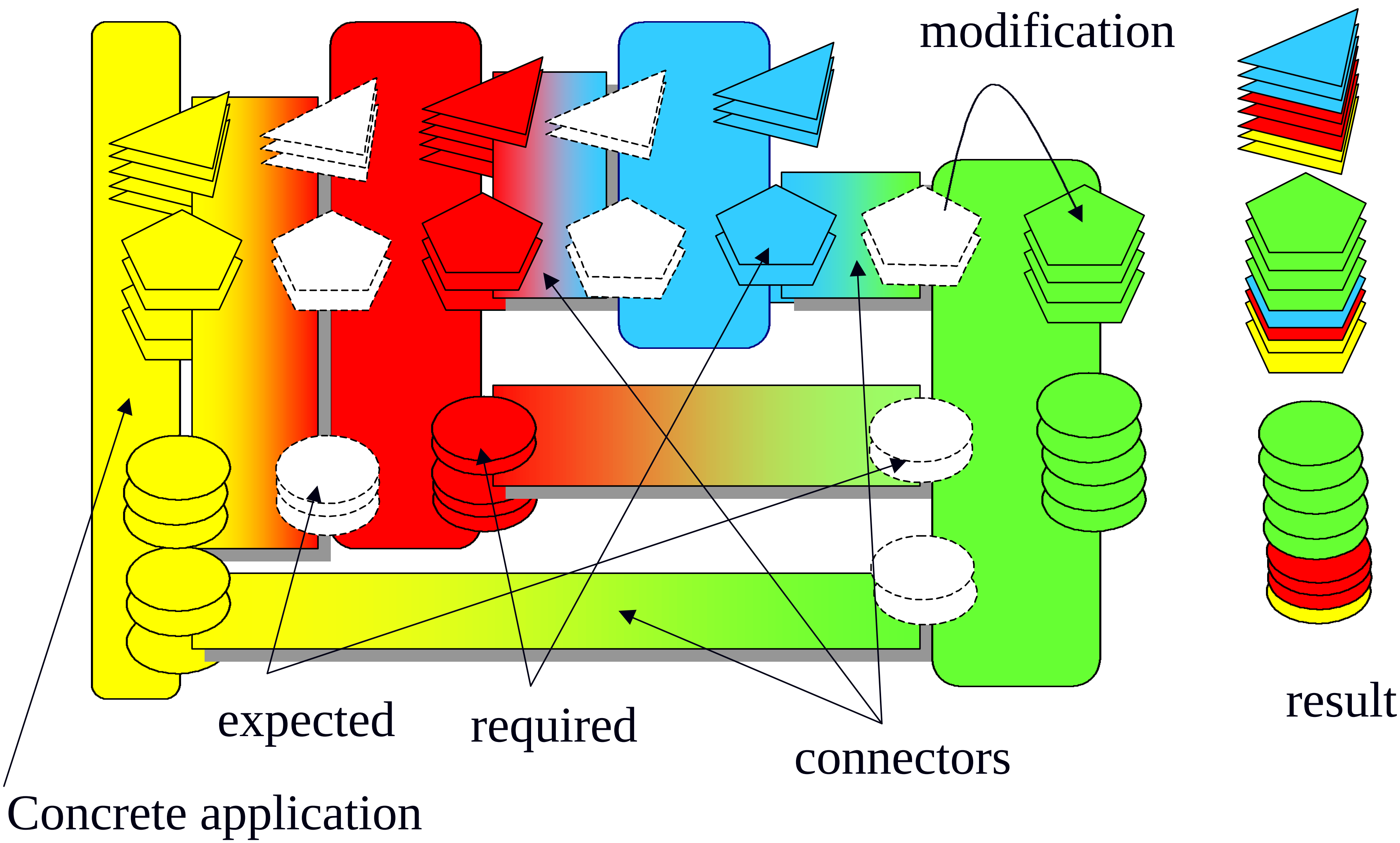
Deployment/Composition of ACs

- Specified by connectors separately from aspectual components
- Connectors use
  - regular-expressions to express sets of method names and class names and interface names
  - standard code everywhere simple method name mapping is not enough
  - graphs and regular expression-like constructs for mapping graphs

Deploying/Composing ACs



Reconciling objects and functions: the intuition behind aspectual components



```
component UsingComparables {

  participant Comparable {
    public int compareTo(Object that);
   }

 class ComparableClient {
    Comparable[] c;
    public Comparable[]
              filterAllSmaller(Object that) {
        Comparable[] t;
         int j = 0;
        for (int i = 0; i < c.length; i++) {
          if (c[i].compareTo(obj) >= 0)  {
                 t[j] = c[i];
                 j = j +1;}
          }
      }
}
```

```
connector applWithComparison {
   appl.Byte implements UsingComparables.Comparable {
    public int compareTo(Object that) {
           return  myCompareTo((Byte) that); }
   }
```

```
package appl;
  ...
  class Byte {
   private byte value;
   public Byte(byte value) {this.value = value; }
   public byte byteValue() {return value;}
   public myCompareTo(Byte that) {
      return this.value - that.value;}
```

incomplete

```
component UsingComparables {

  interface Comparable {
    public int compareTo(Object that);
  }

 class ComparableClient {
    Comparable[] c;
    public Comparable[]
              filterAllSmaller(Object that) {
       Comparable[] t;
        int j = 0;
       for (int i = 0; i < c.length; i++) {
          if (c[i].compareTo(obj) >= 0)  {
                 t[j] = c[i];
                  j = j +1;}
         }
      }
}
```

```
connector applWithComparables {
   appl.Byte implements UsingComparable.Comparable {
    public int compareTo(Object that) {
         return this.byteValue() -
                 (Byte)  that.byteValue(); } }
  }
```

```
package appl;
   ...
   class Byte {
     private byte value;
     public Byte(byte value) {this.value = value; }
     public byte byteValue() {return value;}
```

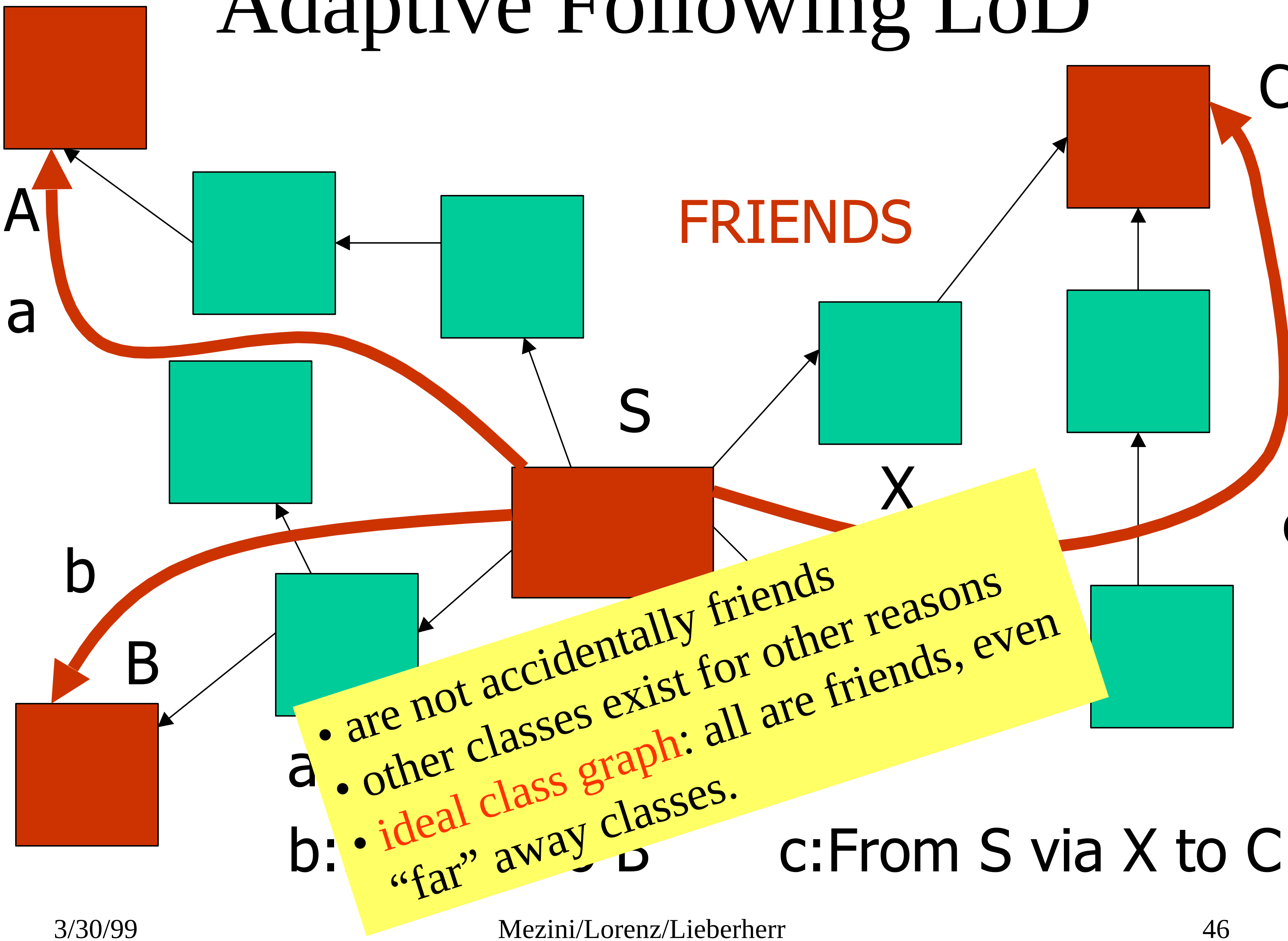
Ideal Class Graph
Where Have We Seen That Before ?

Quote:

Avoid traversing multiple links or methods. **A method should have limited knowledge of an object model.** A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class.

Rumbaugh and the Law of Demeter (LoD)

Adaptive Following LoD



Deploying/Composing ACs

an example ...

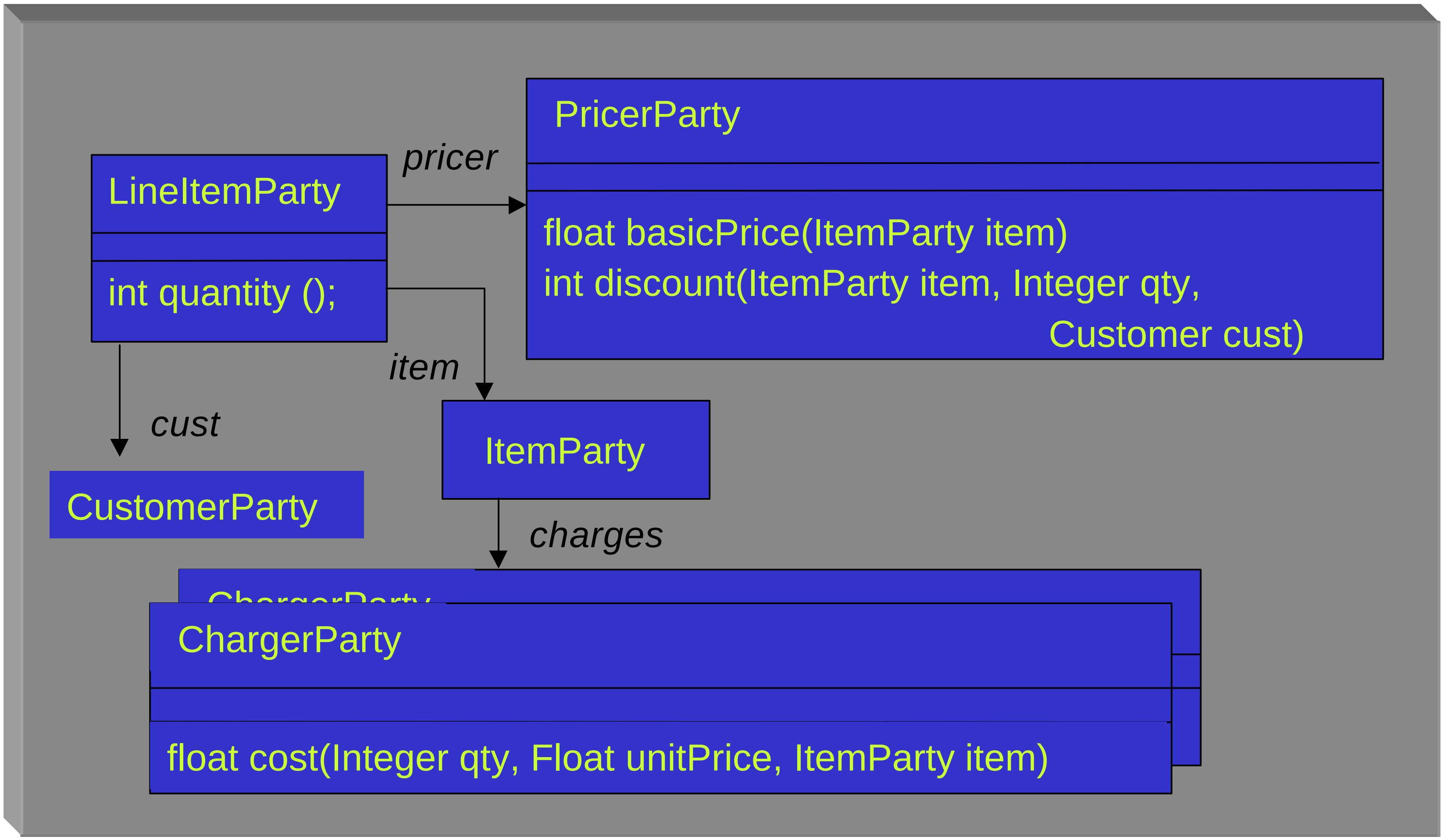
an application generator from IBM (‘70)

Hardgoods Distributors Management Accounting System

encode a generic design for order entry systems which could be subsequently customized to produce an application meeting a customer’s specific needs

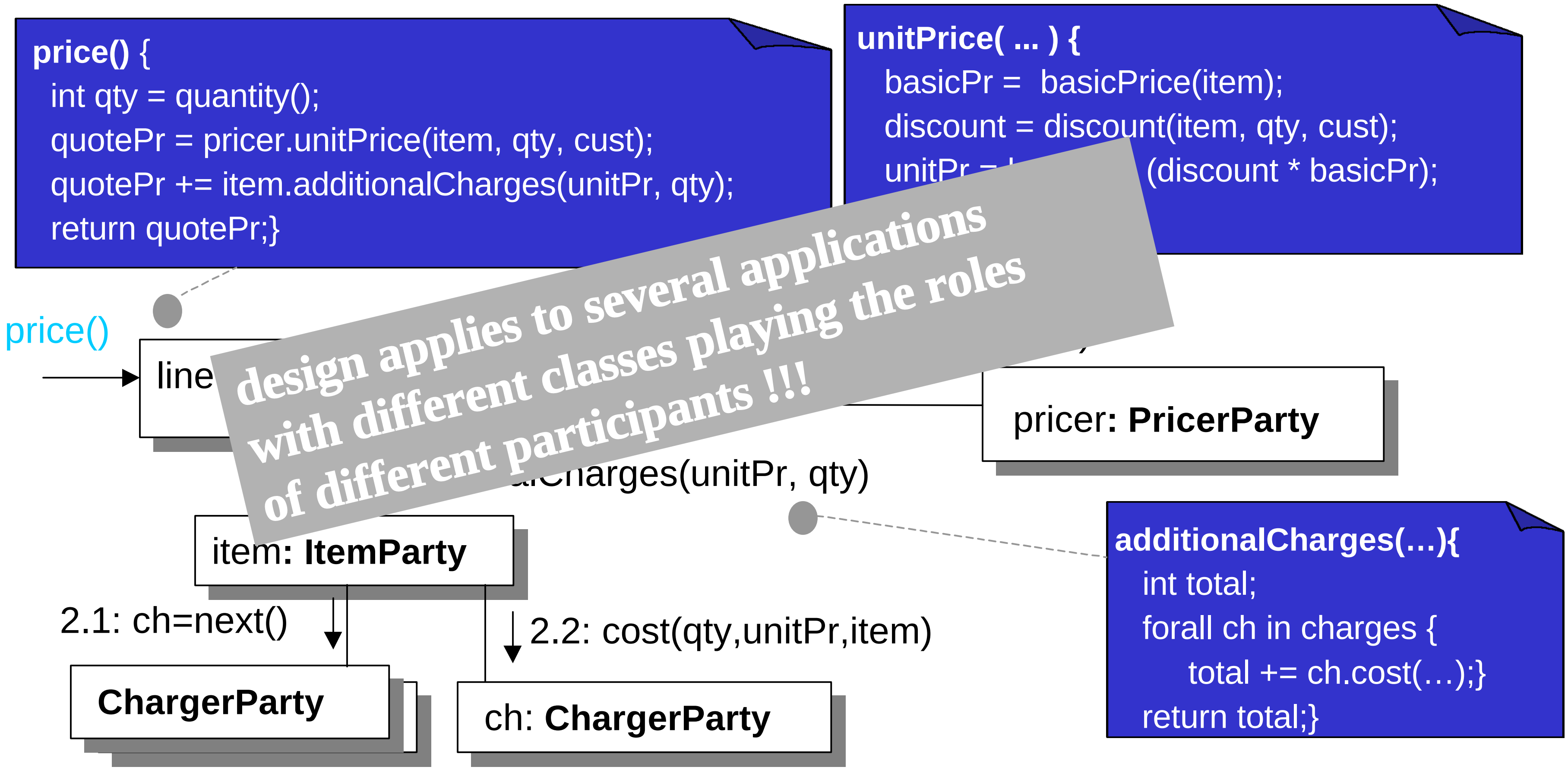
consider the pricing component ...

Deploying ACs



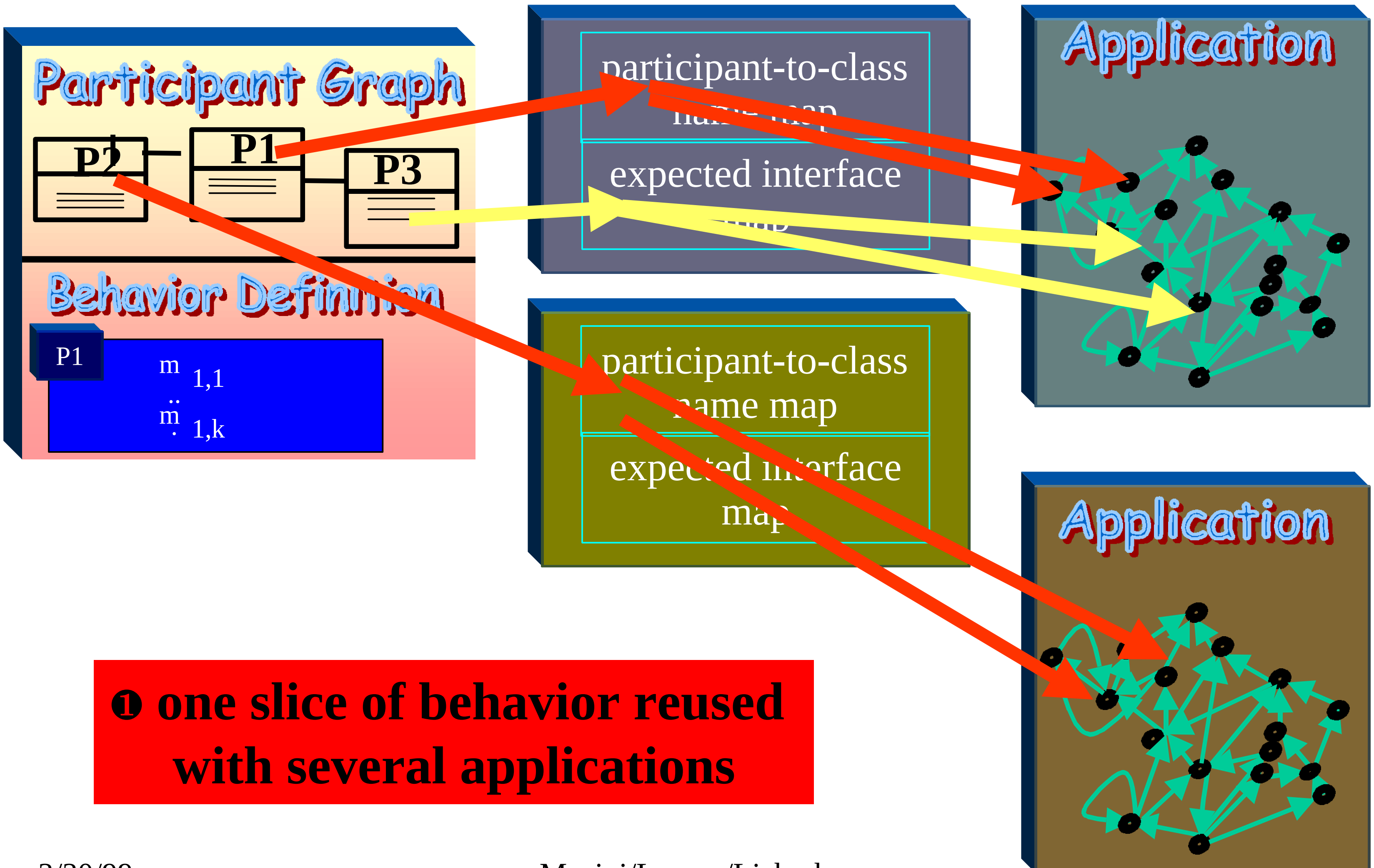
pricing component: class diagram

Deploying ACs



pricing component: collaboration diagram

One AC deployed into several applications



❶ one slice of behavior reused with several applications

Deploying/Composing/Refining ACs

- ❶ **one slice of high-level behavior reused with several applications**

- ❷ **one slice of behavior multiply reused in different places of a single application**

- ❸ **behavior defined in terms of lower-level behavior; high-level behavior definition reused with different lower-level behavior implementations**

- ❹ **define new behavior by refining existing behavior**

Multiply deploying an AC into an application

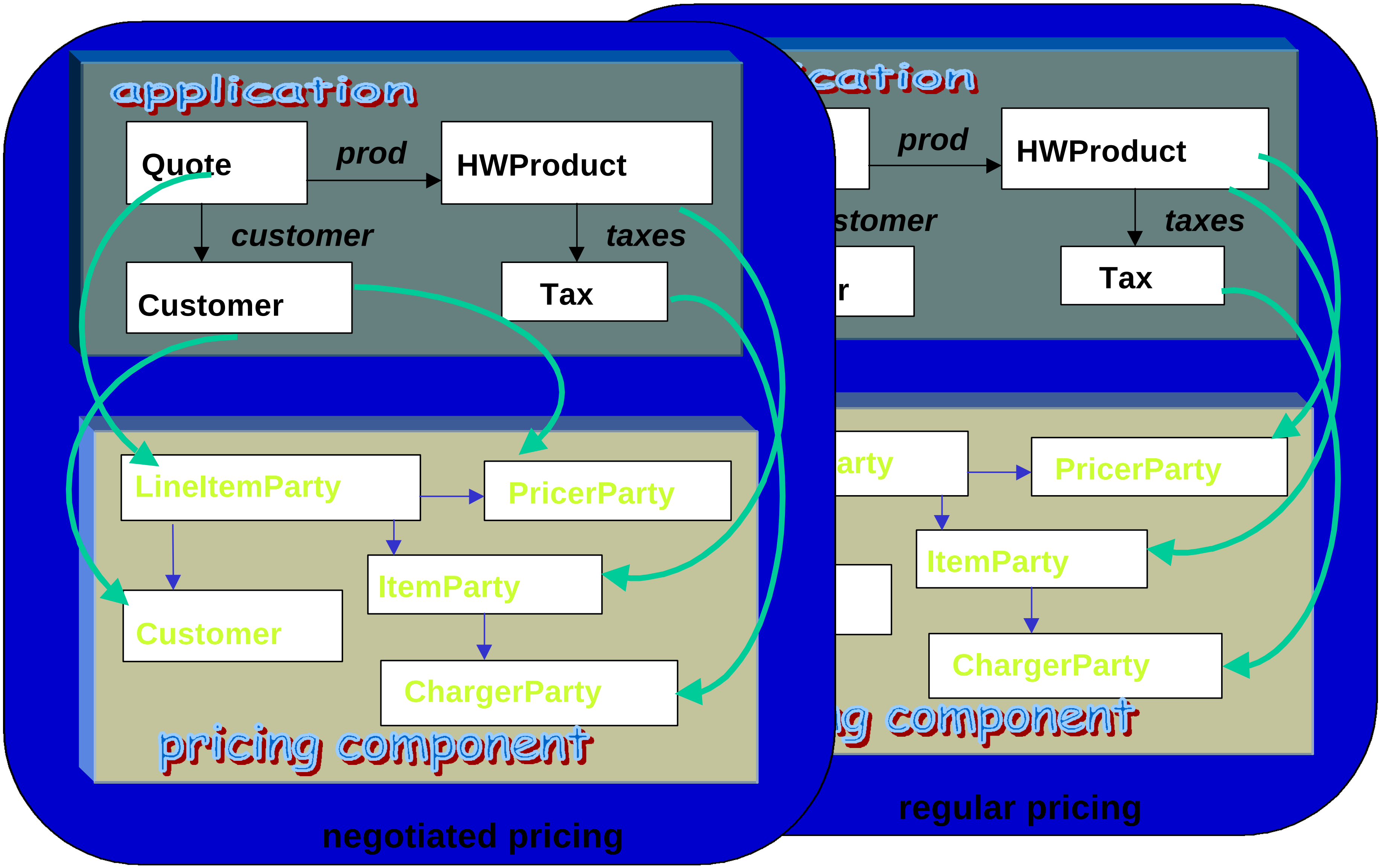
❷ one slice of behavior multiply deployed into different places of a single application

☛ may need to represent several pri...

- regular pricing: di... of ordere...
- ne... have negotiated pric...
- sale pricing: each product has a designated sale price and no discounting allowed

Design is the same for all schemes !!!
Given a concrete application, each scheme might require the application class model to conform to the design in a specific way

Multiply deploying an AC into



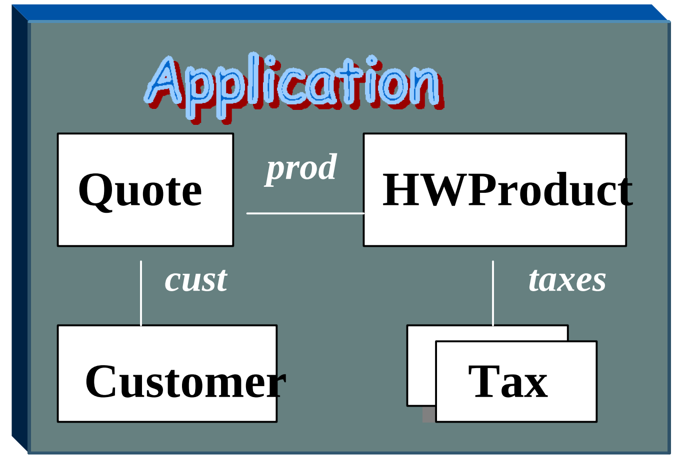
❷ one slice of behavior multiply reused in different places of a single application

Multiply deploying an AC into an application

Map 1

```
connector HWApplWithRegPricing {
   // connects HWApp, Pricing;
   Quote  is LineItemParty {
         with{regularPrice = price }
      };
   HWProduct is PricerParty {
      with {
        float basicPrice() {return regPrice();}
        float discount() {return regDiscount();}
       };
   HWProduct is ItemParty;
   Tax is ChargerParty;}
```

Pricing AC

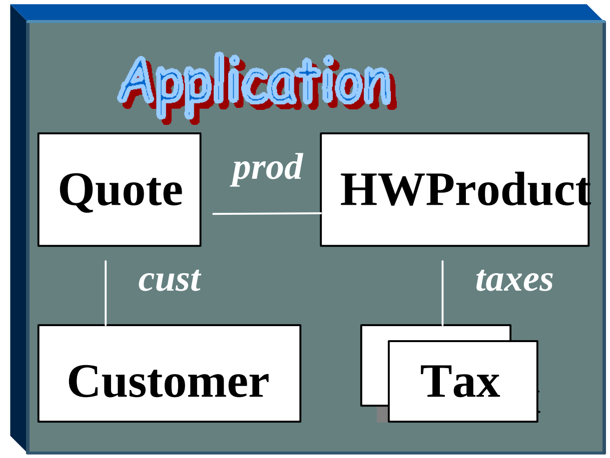


AC compiler
(CG-to-PG compatability?)

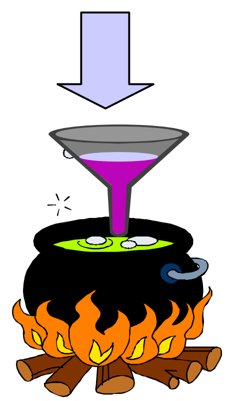
Multiply deploying an AC into an application

Map 2

```
connector HWApplWithNegPricing {
  connec ts HWApp, Pricing;
  Quote  implements LineItemParty {
        provided {negotiatedPrice = price }
     }
 Customer implements PricerParty {
   expected {
     float basicPrice() {return negProdPrice();}
      float discount() {return negProdDiscount();}
      } }
 HWProduct implements ItemParty;
 Tax implements  ChargerParty;}
```



Pricing AC



AC compiler
(CG-to-PG compatibility?)

Deploying/Composing/Refining ACs

❶ one slice of high-level behavior reused with several applications

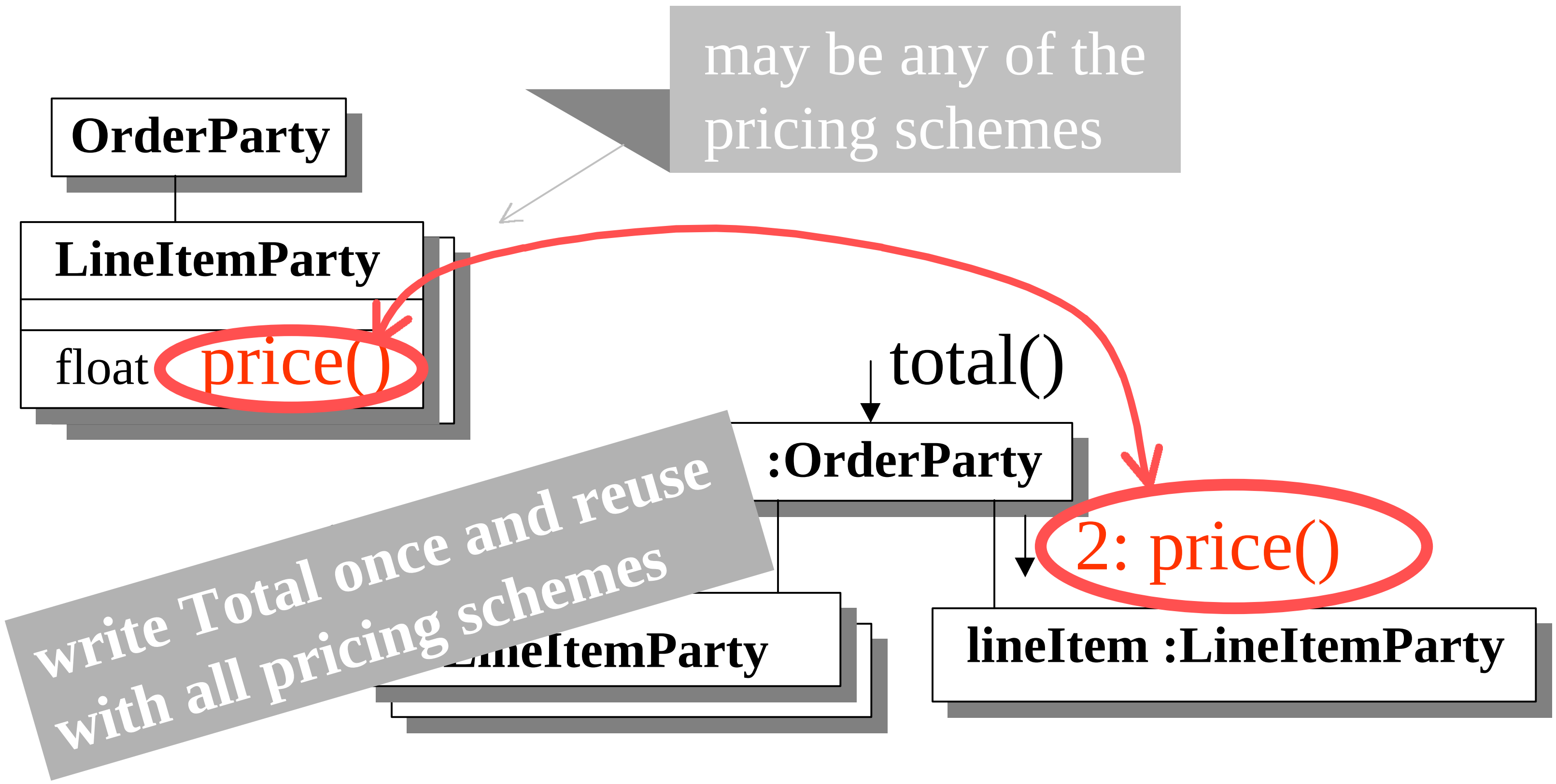
❷ one slice of behavior multiply reused in different places of a single application

❸ behavior defined in terms of lower-level behavior; high-level behavior definition reused with different lower-level behavior implementations

❹ define new behavior by refining existing behavior

Composing ACs

➌ define higher-level behavior in terms of lower-level behavior



Composing ACs

expected interface of one AC mapped to provided interface of other AC

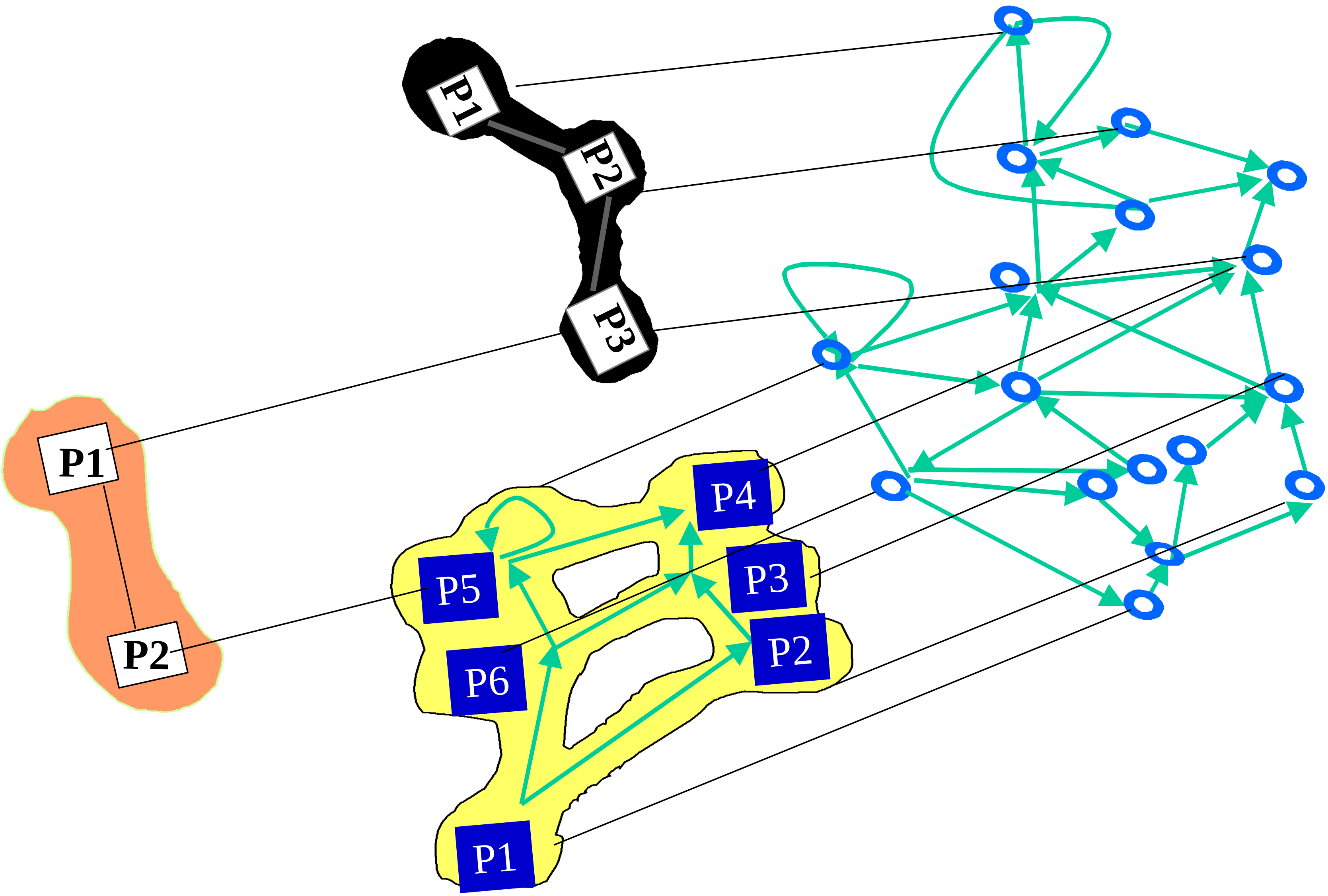
```
component Total {
  Participant-Graph:
    participant OrderParty {
      expect Customer customer
        expect LineItemParty[] lineItems)
    participant LineItemParty  { float price(); }

  Behavior-Definition:
    OrderParty {
      public  float total()  {
          ...
            while lineItems.hasElements()) {
            total += nextLineItem.price(); }
            return total; }
      }
}
```

```
connector applWithTotal{
    connects HWAppl, Total;
    Order implements OrderParty ;
    LineItemParty  implements  Quote
      expected {
        price() { return regularPrice();
      };
    }
}
```

```
connector ApplWithPricing {
    { . . . regularPrice() }
```

Pricing AC

Software Structure with ACs



Deploying/Composing/Refining ACs

❶ **one slice of high-level behavior reused with several applications**

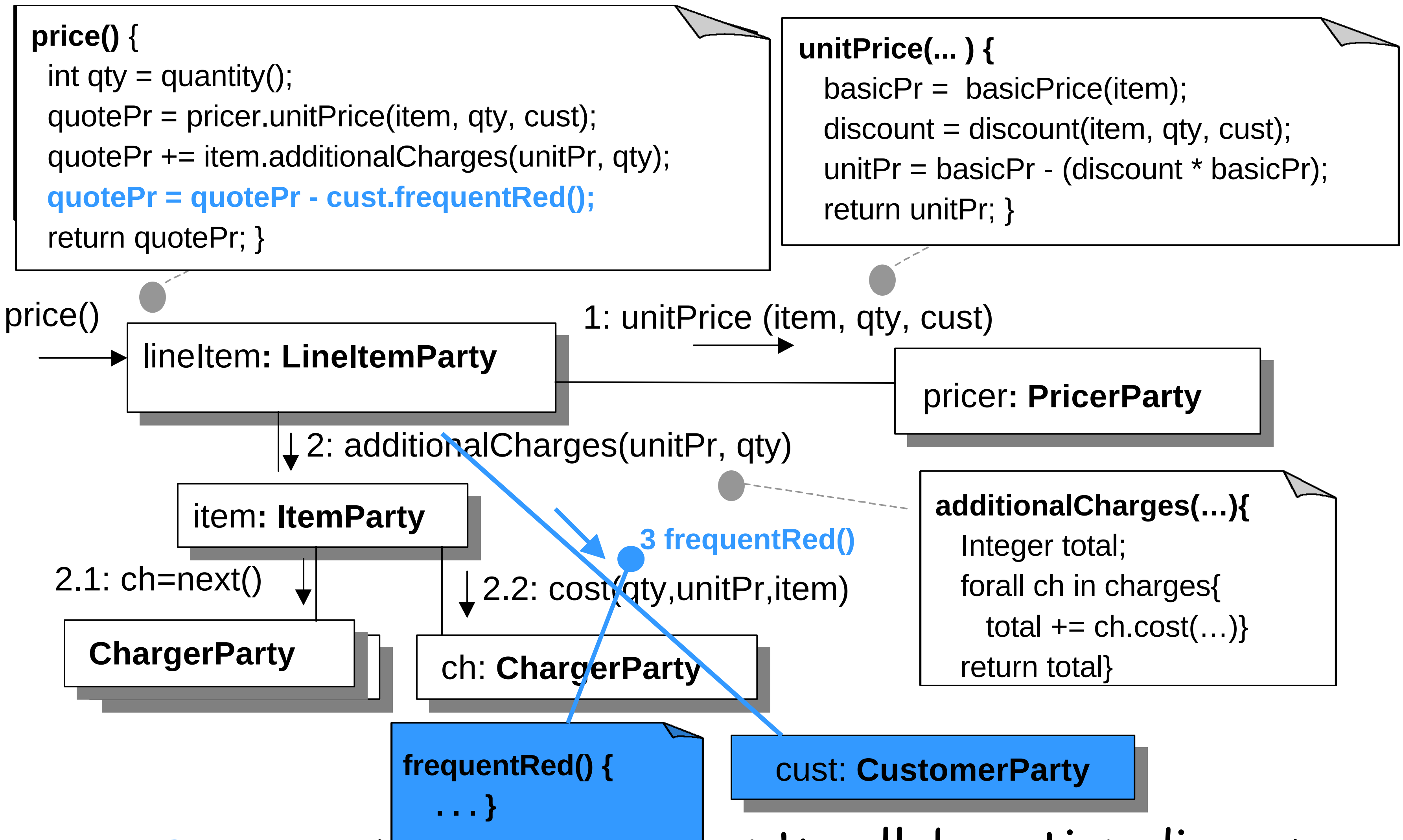
❷ **one slice of behavior multiply reused in different places of a single application**

❸ **behavior defined in terms of lower-level behavior; high-level behavior definition reused with different lower-level behavior implementations**

❹ **define new behavior by refining existing behavior**

Refining ACs

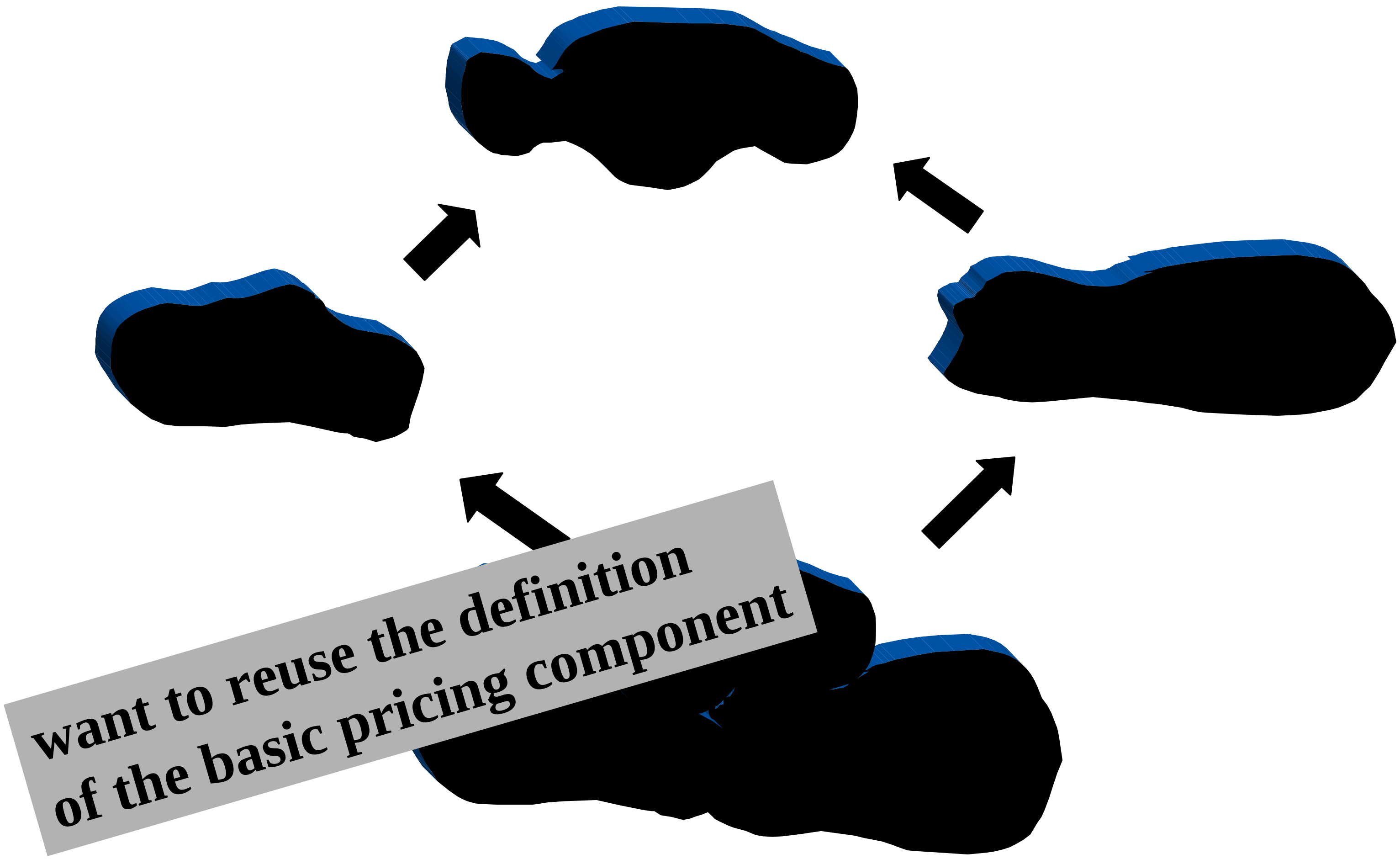
❹ define new behavior by refining existing behavior



frequent pricing component: collaboration diagram

Refining ACs

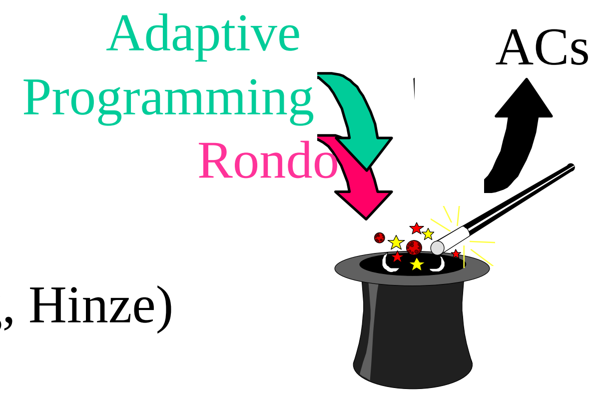
❹ define new behavior by combining existing behavior



Summary so far

- ACs as larger-grained constructs that complement classes in modeling collaborations or behavior that cross-cut class boundaries
- Generic behavior that can be reused with a family of applications
- Independent development of components
- Independent connectors of ACs with applications
- Independent interfaces that are adapted explicitly
- Decoupled black-box composition of collaborations
- Definition of new collaborations as refinements of existing collaborations

Related work



- visitor pattern (GOF, Chrishnamurthi & al)
- polytypic programming (Jansson & Jeuring, Hinze)
- role modeling with template classes (VanHilst & Notkin)
- mixin-layers (Smaragdakis & Batory)
- contracts (Holland)
- AOP (Kiczales & Lopes)
- SOP (Harrison & Ossher)

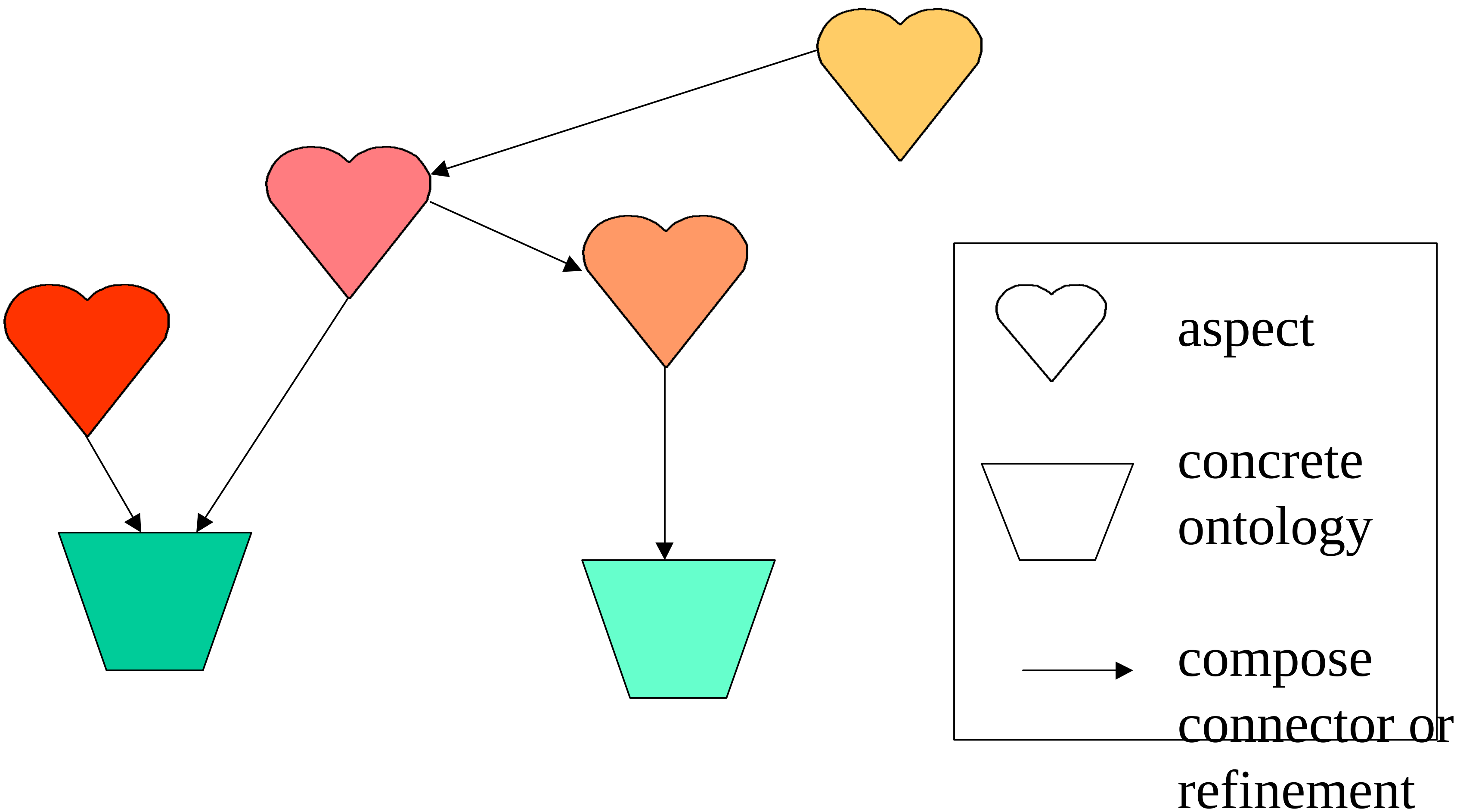
Aspect-Oriented Programming (AOP) Definition

- Aspect-oriented programs consist of complementary, collaborating aspects, each one addressing a different application/system level concern
- Two aspects A1 and A2 are complementary collaborating aspects if an element a1 of A1 is formulated in terms of partial information about elements of A2 and A1 adds information to A2 not provided by another aspect.

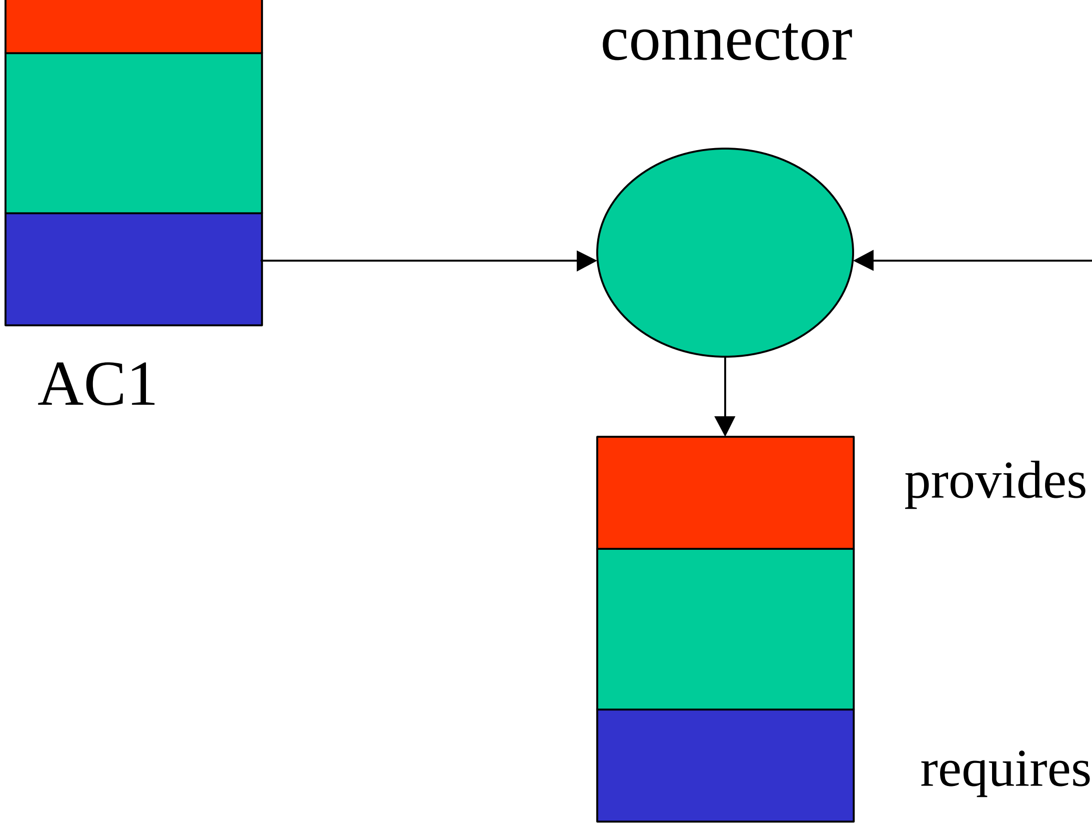
AOP Definition (cont.)

- The partial information about A2 is called join points and provides the range of the weaving in A2.
- The domain of the weaving is in A1 and consists of weaves that refer to the join points. The weaves describe enhancements to A2.
- The join points may be spread through A2. After the weaving, enhancements from a1 effectively cross-cuts A2

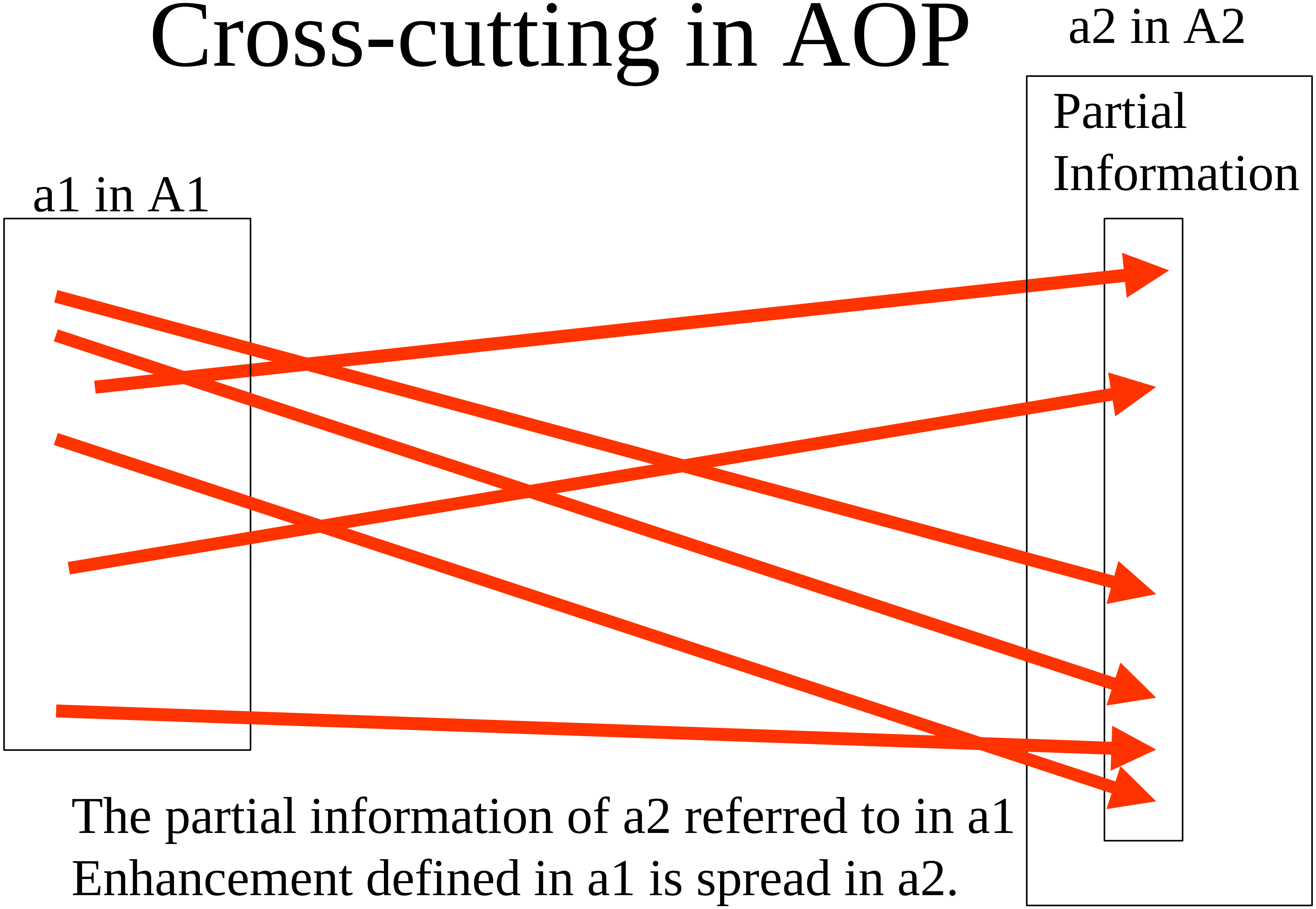
Graph of components



Components and connectors



Cross-cutting in AOP



The partial information of a2 referred to in a1
Enhancement defined in a1 is spread in a2.
a1 adds to a2.

Example: Write accesses

aspect

```
aspect ShowAccesses {
  static before Point.set,
                Point.setX,
                Point.setY {
    System.out.println("W");
  }
}
```

application

```
class Point {
  int _x = 0;
  int _y = 0;

  void set(int x, int y) {
    _x = x; _y = y;
  }

  void setX(int x)
      { _x = x; }

  void setY(int y)
       { _y = y; }

  int getX(){
     return _x; }

  int getY(){
      return _y; }
}
```

AOP example with AC

```
component ShowWAccesses {
  expect {
   Data-To-Access{
     void writeOp(*);}
     replace Object writeOp(){
       System.out.println("W");
       expected(*);}
}
```

```
connector AddShowWAccesses {
  //connects appl, ShowWAccesses ...
     Point is Data-To-Access {
          writeOp = set* ...
     }
  }
```

```
class Point {
  int _x = 0;
  int _y = 0;

  void set(int x, int y) {
    _x = x; _y = y;
  }

  void setX(int x)
      { _x = x; }

  void setY(int y)
       { _y = y; }

  int getX(){
     return _x; }

  int getY(){
      return _y; }
}
```

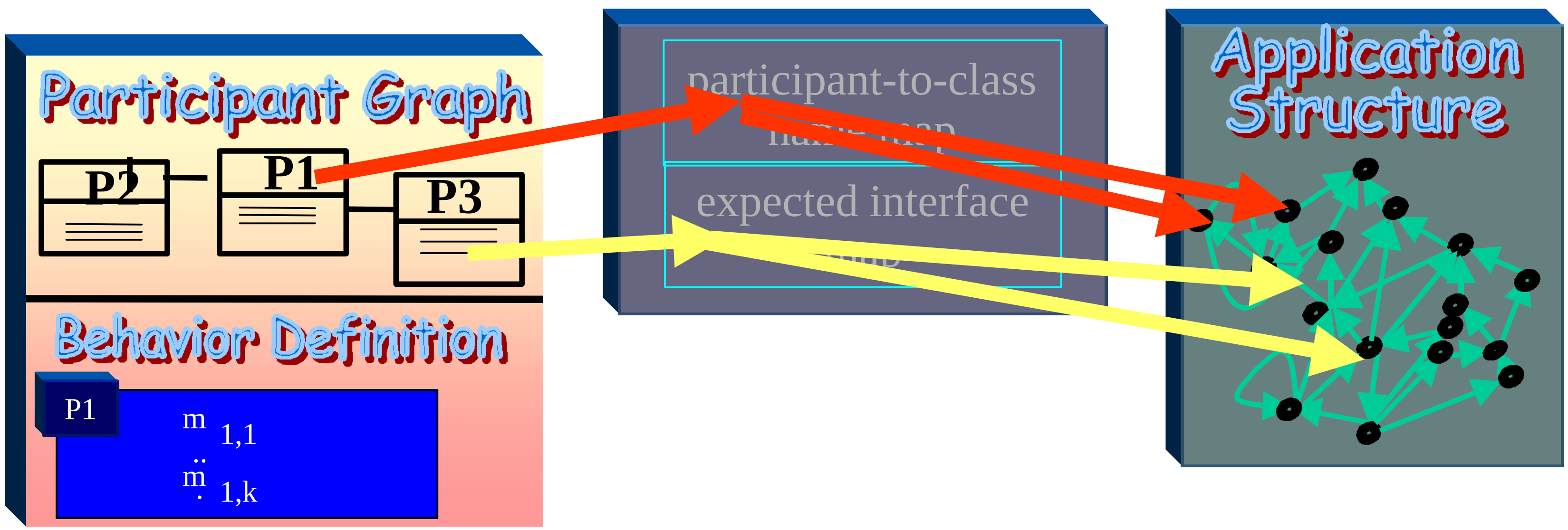
Alternative syntax?

```
component ShowWAccesses {
  expected {
   Data-To-Access{
       * write-op(*);}
   }
  provided {
   Data-To-Access {
     * write-op(*) {
      System.out.println("W");
      write-op(*);}
    }}
}
```

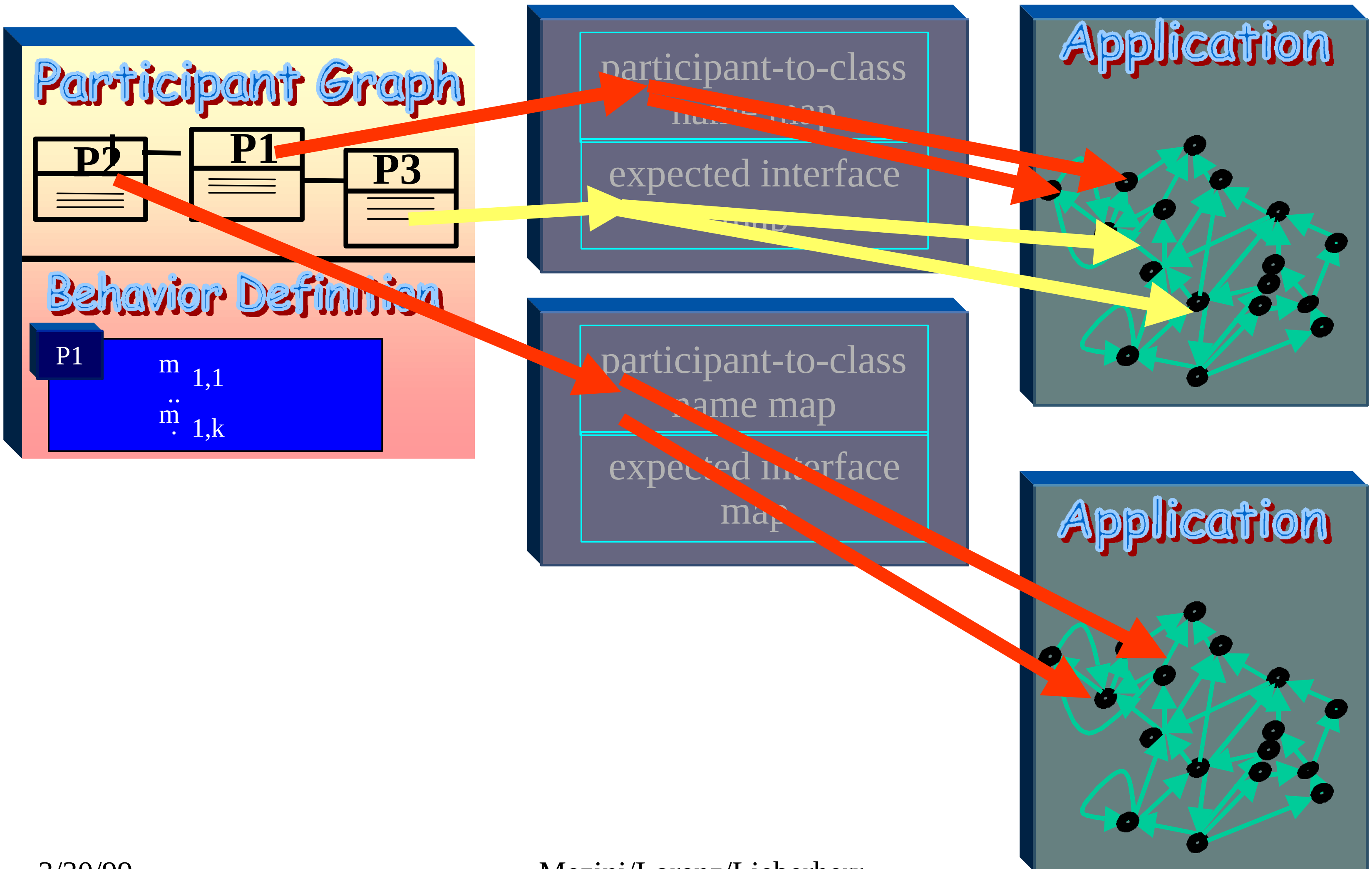
```
connector AddShowWAccesses {
  connects appl, ShowWAccesses ...
      Point is Data-To-Access {
             write-op = set* ...
      }
  }
```

```
class Point {
  int _x = 0;
  int _y = 0;

  void set(int x, int y) {
    _x = x; _y = y;
  }

  void setX(int x)
      { _x = x; }

  void setY(int y)
      { _y = y; }

  int getX(){
     return _x; }

  int getY(){
      return _y; }
}
```

AOP with ACs


Participant Graph
P2
P1
P3
Behavior Definition
P1
m 1,1
m 1,k
participant-to-class
expected interface
Application Structure

AOP with ACs


Participant Graph
P2
P1
P3
Behavior Definition
P1
m 1,1
m 1,k
participant-to-class name map
expected interface map
participant-to-class name map
expected interface map
Application
Application

AOP with ACs

```
Application {
  . . .
  FIFOQueue {
     List elements = new List();

      public void put(Object e) {
           elements.insertLast(e); }

      public Object get() {
          e = elements.removeFirst();
          return e;}
}
```

```
component Monitor {
   expected {
       Data-To-Protect {* access-op(*);}
      }

   provided {
      private Semaphore mutex = new  Semaphore(1);

      Data-To-Protect {

           * access-op(*) {
                  mutex.P();
                  * access-op(*);
                  mutex.V(); }
```

```
connector  ConcurentApplication {
   connects Application, Monitor;
      FIFOQueue  implements  Data-To-Protect {
         expexted { access-op = {put, get} }
      }
     ...
  }
```

AOP with ACs

```
Application {
  . . .
  class HTTPServer {

    public HTMLDocument
      getURL(String url) { . . .  }

    public void
```

```
component Rendez-Vous-Synchronization {
expected {
    Data-To-Protect {* access-op(*);}
    }

 provided {
   Semaphore mutex =  new Semaphore(0);
```

```
connector  ConcWebApplication {
   connects Application, Rendez-Vous-Synchronization;

   Application.HTTPServer  implements Rendez-Vous-Synchronization.Data-To-Protect {
       expexted { access-op = {putURL, getURL} }
      }
    }

ConcWebApplication.HTTPServer myServer = new ConcWebApplication. HTTPServer();
// Thread 1
while (true) {myServer.accept();}
```

```
//Thread 2
Browser b1 = new Browser();
b1.connect(myServer);
```

```
// Thread 3
Browser b2 = new Browser();
b2.connect(myServer);
```

Generalized Parameterized Programming

- Loose coupling is achieved by writing each component in terms of interfaces expected to be implemented by other components. This leads to a parameterized program with cross-cutting parameters P(C1, C2, ...).

Enterprise Java Beans (EJB) and Aspectual components

- EJB: a hot Java component technology from SUN/IBM
- Aspectual components: a conceptual tool for the design of enterprise Java beans (and other components)

Enterprise JavaBeans (EJB)

- Addresses aspectual decomposition.
- An enterprise Bean provider usually does not program transactions, concurrency, security, distribution and other services into the enterprise Beans.
- An enterprise Bean provider relies on an EJB container provider for these services.

EJB

- Beans
- Containers: to manage and adapt the beans. Intercept messages sent to beans and can execute additional code. Similar to reimplementation of expected interface in aspectual component.

Aspectual components for EJB design/implementation

- Use ACs to model transactions, concurrency, security, distribution and other system level issues. Translate ACs to deployment descriptors (manually, or by tool).
- Use ACs to model beans in reusable form. Generate (manually or by tool) Java classes from ACs and connectors.

Example: Use AC for EJB persistence

As an example we consider how persistence is handled by EJB containers. The deployment descriptor of a bean contains an instance variable ContainerManagedFields defining the instance variables that need to be read or written. This will be used to generate the database access code automatically and protects the bean from database specific code.

Aspectual component: Persistence

```
component Persistence { PerMem p;
  participant Source {
    expect Target[] targets;
    expect void writeOp();}
      // for all targets:writeOp
  participant Target
    expect void writeOp();
    replace void writeOp() {
     // write to persistent memory p
     expected();}}
```

Deployment

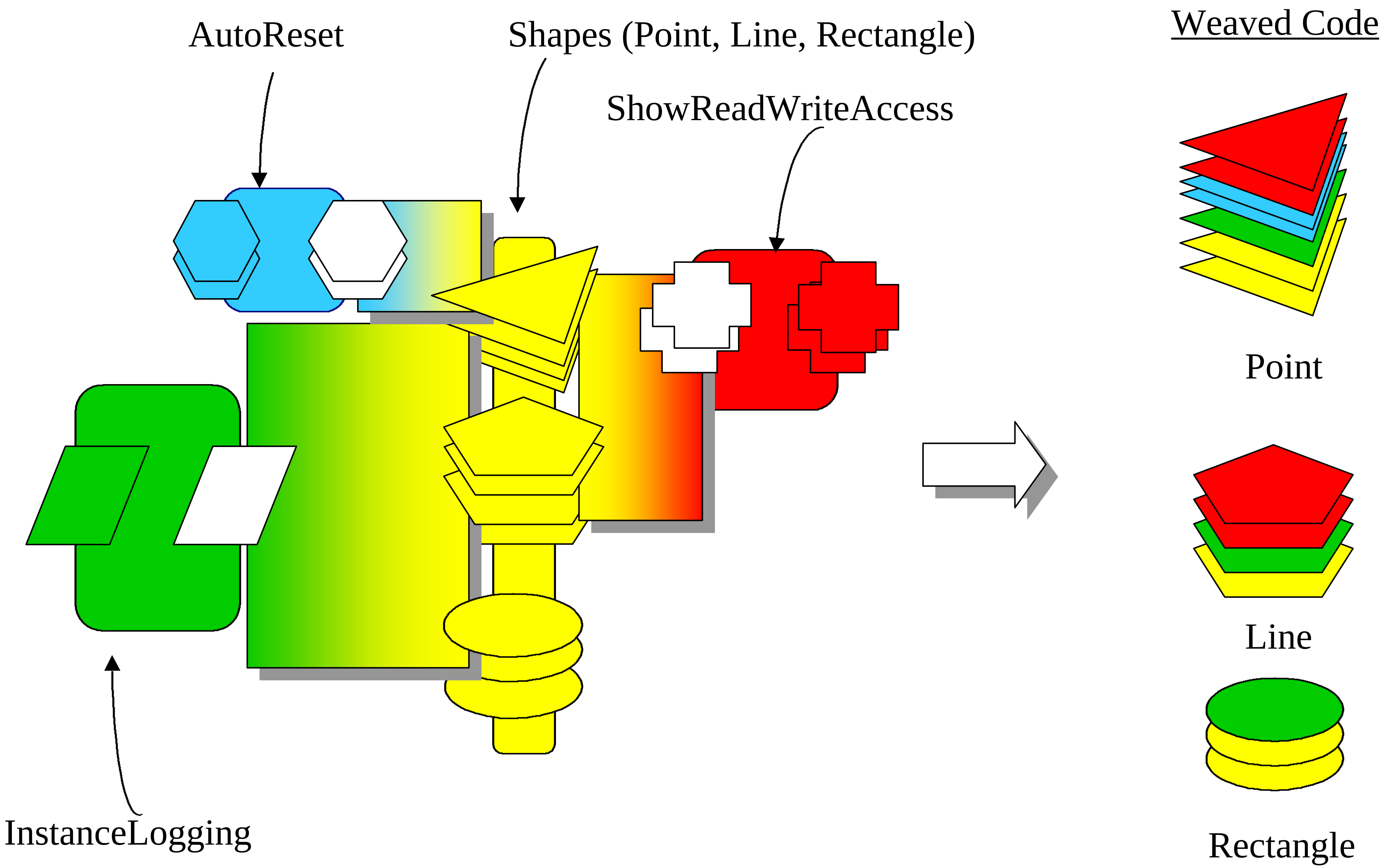
```
connector  PersistenceConn1 {
  ClassGraph g = … ; // from Company …
  Company is Persistence.Source;
  Nodes(g) is Persistence.Target;
  g is Persistence.(Source,Target);
  with {writeOp = write*};
  // must be the same writeOp for both
  // Source and Target
}
```

Generate deployment descriptor

- Connector contains information about ContainerManagedFields
- Connector localizes information; it is not spread through several classes

Composition example

- Use three aspects simultaneously with three classes.
- Three aspects:
  - ShowReadWriteAccess
  - InstanceLogging
  - AutoReset
- Three classes: Point, Line, Rectangle


AutoReset
Shapes (Point, Line, Rectangle)
ShowReadWriteAccess
InstanceLogging
Weaved Code
Point
Line
Rectangle

Inheritance between components

```
component ShowReadWriteAccess extends
  ShowReadAccess {
  participant DataToAccess {
     expect void writeOp(Object[] args);
     replace void writeOp(Object[] args){
        System.out.println(
        "Write access on " +
            this.toString());
        expected(args);}}
}
```

InstanceLogging component (first part)

```
component InstanceLogging {
  participant DataToLog {
    expect public DataToLog(Object[] args);
    replace public DataToLog(Object[] args) {
      expected(args);
      long time = System.currentTimeMillis();
      try {
        String class =  this.class.getName() + " ";
        logObject.writeBytes(""New instance of " + class +
        at "" " + time + "" " \n");
      } catch (IOException e)
            {System.out.println(e.toString());}
    }
  }
```

InstanceLogging component (second part)

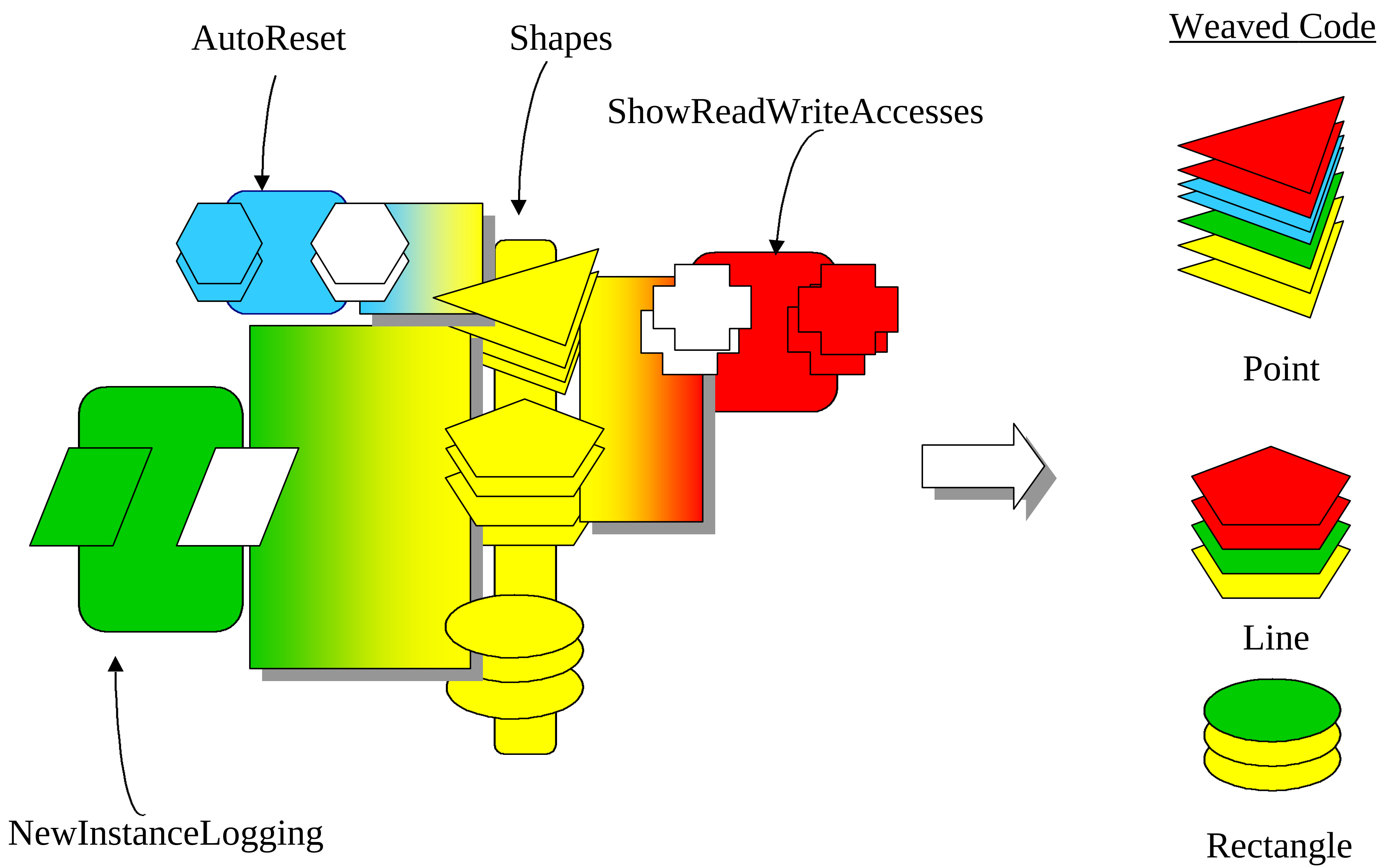
```
  protected DataOutputStream logObject = null;
  public init() {
      try {logObject = new DataOutputStream(
              new FileOutputStream(log));}
      catch (IOException e)
        {System.out.println(e.toString());}
     }
}
```

AutoReset component

```
component AutoReset {
  participant DataToReset {
    expect void setOp(Object[] args);
    expect void reset();
    protected int count = 0;
    replace void setOp(Object[] args) {
     if ( ++count >= 100 ) {
        expected(args);
        count = 0;
        reset();
      }}
   }
}
```

Composition of components

```
connector CompositionConn1 {
  {Line, Point} is
  ShowReadWriteAccess.DataToAccess with
     { readOp = get*; writeOp = set*;};
  Point is AutoReset.DataToReset with {
         setOp = set*;
         void reset() { x = 0; y = 0; }
      };
  {Line, Point, Rectangle} is
     InstanceLogging.DataToLog;}
```


AutoReset
Shapes
ShowReadWriteAccesses
NewInstanceLogging
Weaved Code
Point
Line
Rectangle

Composition of components

```
Connector graph CompositionConn1
                                    Line, Point, Rectangle
ShowReadWriteAccess.DataToAccess      *      *
AutoReset.DataToReset                        *
InstanceLogging.DataToLog             *      *         *
```

Modified composition

```
connector CompositionConn2 extends
   CompositionConn1 {
    Line is AutoReset.DataToReset with {
        setOp = set*;
        void reset() {init();}
        };
}
```

Composition of components

Connector graph CompositionConn1

	Line,	Point,	Rectangle
ShowReadWriteAccess.DataToAccess	*	*	
AutoReset.DataToReset		*	
InstanceLogging.DataToLog	*	*	*

Connector graph CompositionConn2

	Line,	Point,	Rectangle
ShowReadWriteAccess.DataToAccess	*	*	
AutoReset.DataToReset	*	*	
InstanceLogging.DataToLog	*	*	*

Modify existing connection statements

```
connector CompositionConn3 extends CompositionConn1 {
  Point is AutoReset.DataToReset with {
      { setOp = set;
        void reset() {
            x = 0; y = 0; }}
      { setOp = setX;
        void reset() { x = 0;}}
      {
        setOp = setY;
        void reset() { y = 0;}}
   };
}
```

Composition of components

```
Connector graph CompositionConn3
                                    Line, Point, Rectangle
ShowReadWriteAccess.DataToAccess      *      *
AutoReset.DataToReset                        ***
InstanceLogging.DataToLog             *      *         *
```

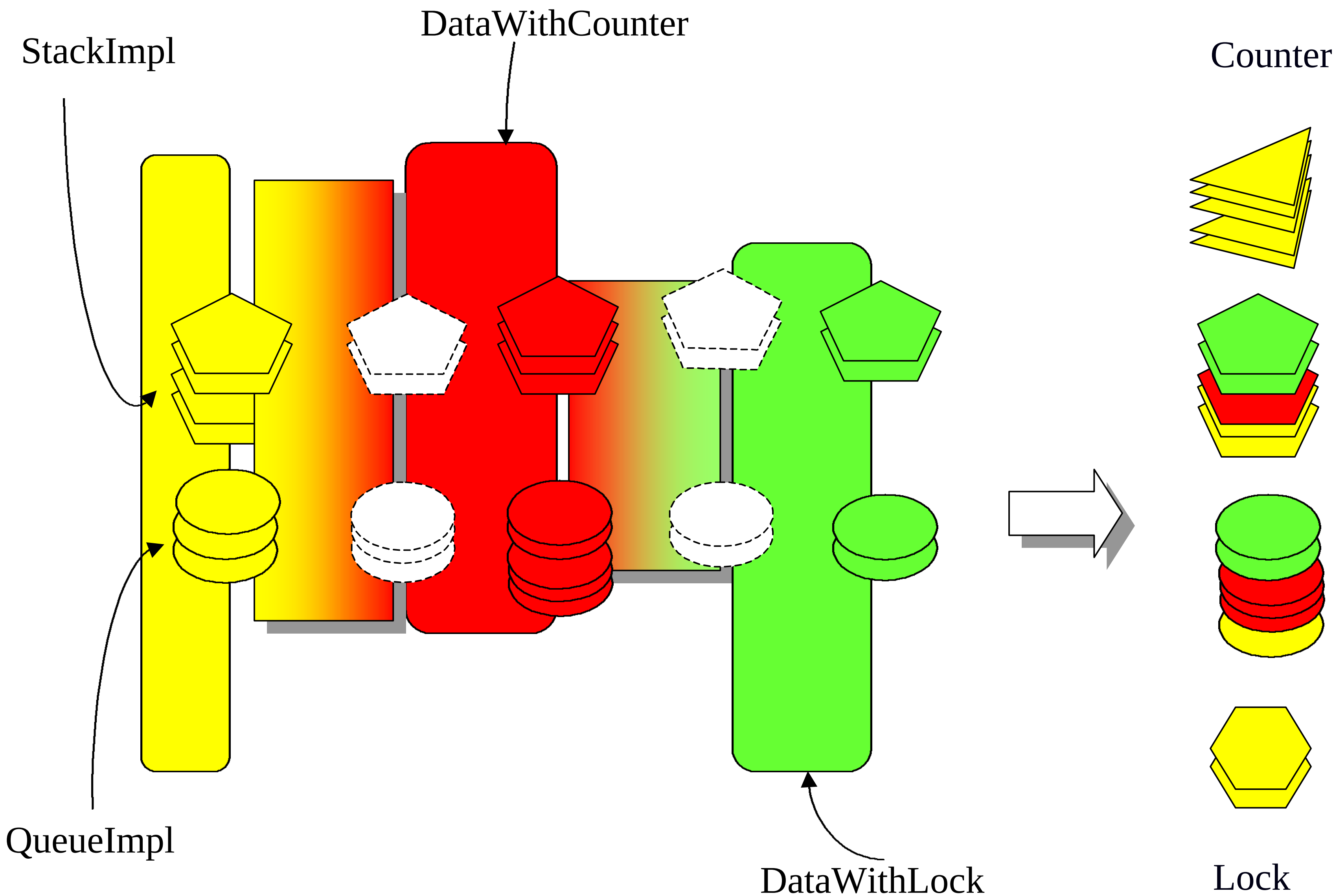
overridden: ***

DataWithCounter component pairwise interaction Data/Counter

```
component DataWithCounter {
   private participant Counter { int i=0;
       void reset(){i=0;}; void inc(){…}; void dec(){…};}
   participant DataStructure {
       protected Counter counter;
       expect void initCounter();
       expect void make_empty();
       expect void push(Object a);
       expect void pop();
       replace void make_empty(){counter.reset();expected();}
       replace void push(Object a){counter.inc(); expected(a);}
       replace void pop() {counter.dec();expected();}
     }
}
```

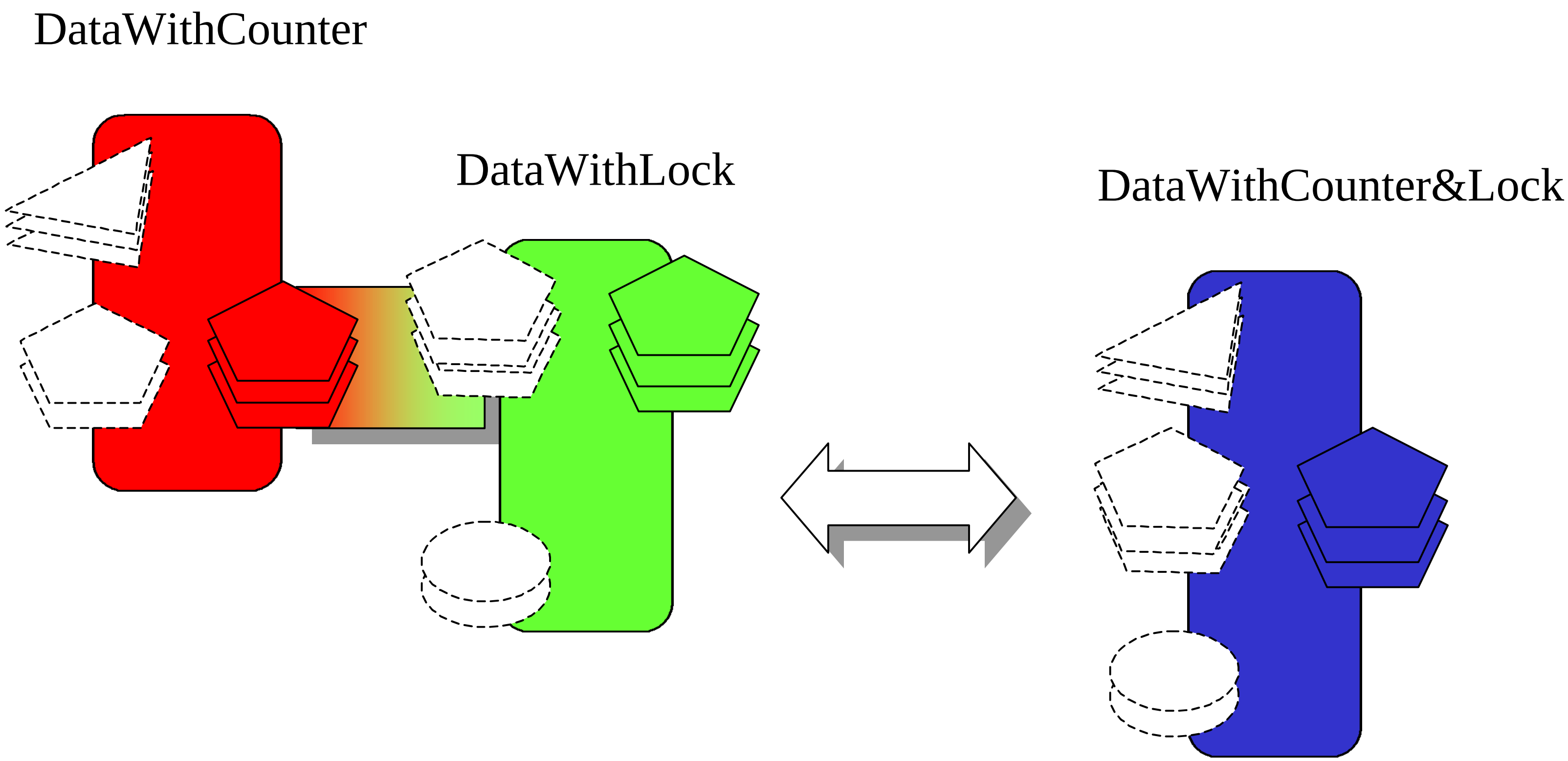
DataWithLock Component pairwise interaction Data/Lock

```
component DataWithLock {
   participant Data {
        Lock lock;
        expect void initLock();
        expect AnyType method_to_wrap(Object[] args);
        replace AnyType method_to_wrap(Object[] args) {
          if (lock.is_unlocked()) {
               lock.lock();
               expected(Object[] args);
               lock.unlock(); }}}
   private participant Lock {boolean l = true;
     void lock(){…};
     void unlock(){…};
     boolean is_unlocked(){return l};}
```


StackImpl
DataWithCounter
Counter
QueueImpl
DataWithLock
Lock

First connector

```
connector addCounter&Lock {
  StackImpl is DataWithCounter.DataStructure
  with {
    void initCounter() {counter = new Counter();}
    void push(Object obj) {push(obj));} // use name map instead
    Object top() {return top();}
      ...
  } is DataWithLock.Data
    with {
      method_to_wrap = {pop, push, top, make_empty, initCounter};
     };
  QueueImpl is DataWithCounter.DataStructure with {
     ... } is DataWithLock.Data with { ... };
}
```


DataWithCounter
DataWithLock
DataWithCounter&Lock

Create composed aspects prior to deployment

```
component DataWithCounterAndLock {
  participant Data =
    DataWithCounter.DataStructure is
      DataWithLock.Data with {
        method-to-wrap =
          {make_empty, pop, top, push}};
}
```

Second connector: Deploy composed component

```
connector addCounter&Lock {
  StackImpl is DataWithCounterAndLock.Data with {
       void make_empty() {empty();}
       void initCounter() {
          counter = new Counter();}
      void push(Object obj) {push(obj);}
      ...
    };
  QueueImpl is DataWithCounterAndLock.Data with
  {...};
}
```

END

Inheritance between components

```
component ShowReadWriteAccess extends
  ShowReadAccess {
  participant DataToAccess {
      expect void writeOp(Object[] args);
      replace void writeOp(Object[] args){
         System.out.println(
         "Write access on " +
             this.toString());
         expected(args);}}
}
```

Inheritance between connectors

```
connector ShowReadWriteAccessConn2
  extends ShowReadAccessConn3  {
   {Point,Line,Rectangle}
    is DataToAccess with {
      writeOp = set*;
   }
```