

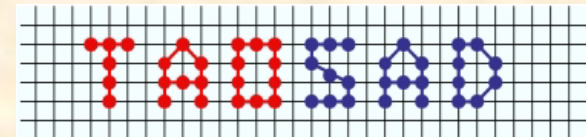
Separating Software Engineering Concerns



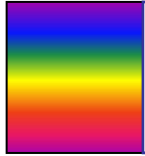
Introduction to AOSD

Bedir Tekinerdoğan

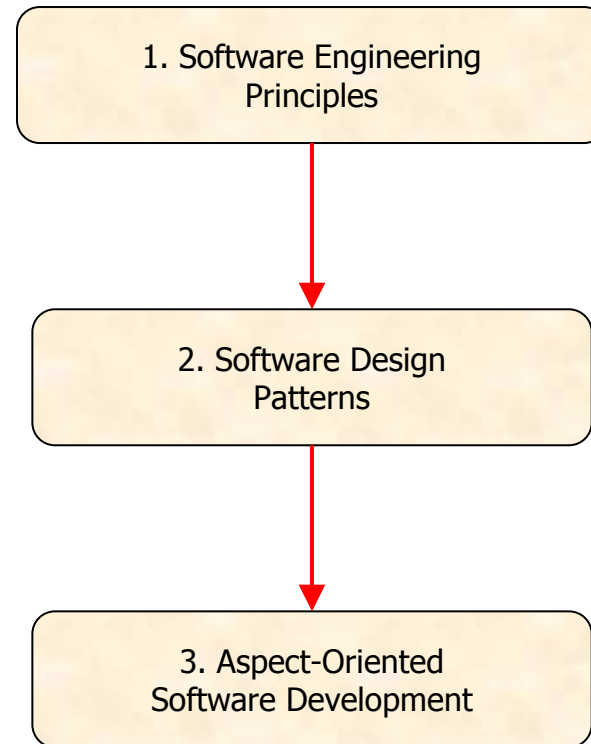
University of Twente
Department of Computer Science
Enschede, The Netherlands
e:mail – bedir@cs.utwente.nl
<http://www.cs.utwente.nl/~bedir/>



<http://trese.cs.utwente.nl/taosad/>



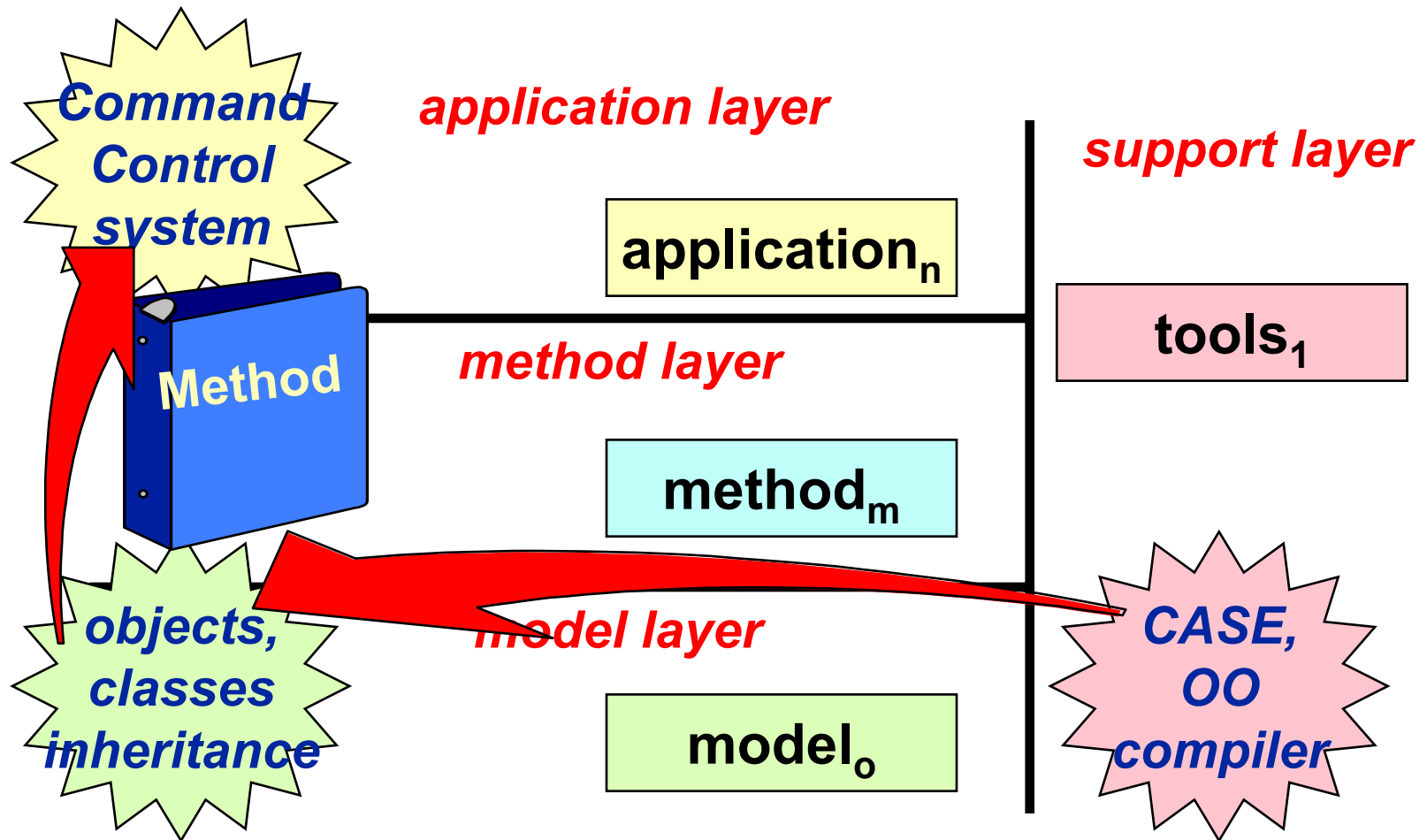
Roadmap of presentation





Software Engineering Principles

Software engineering

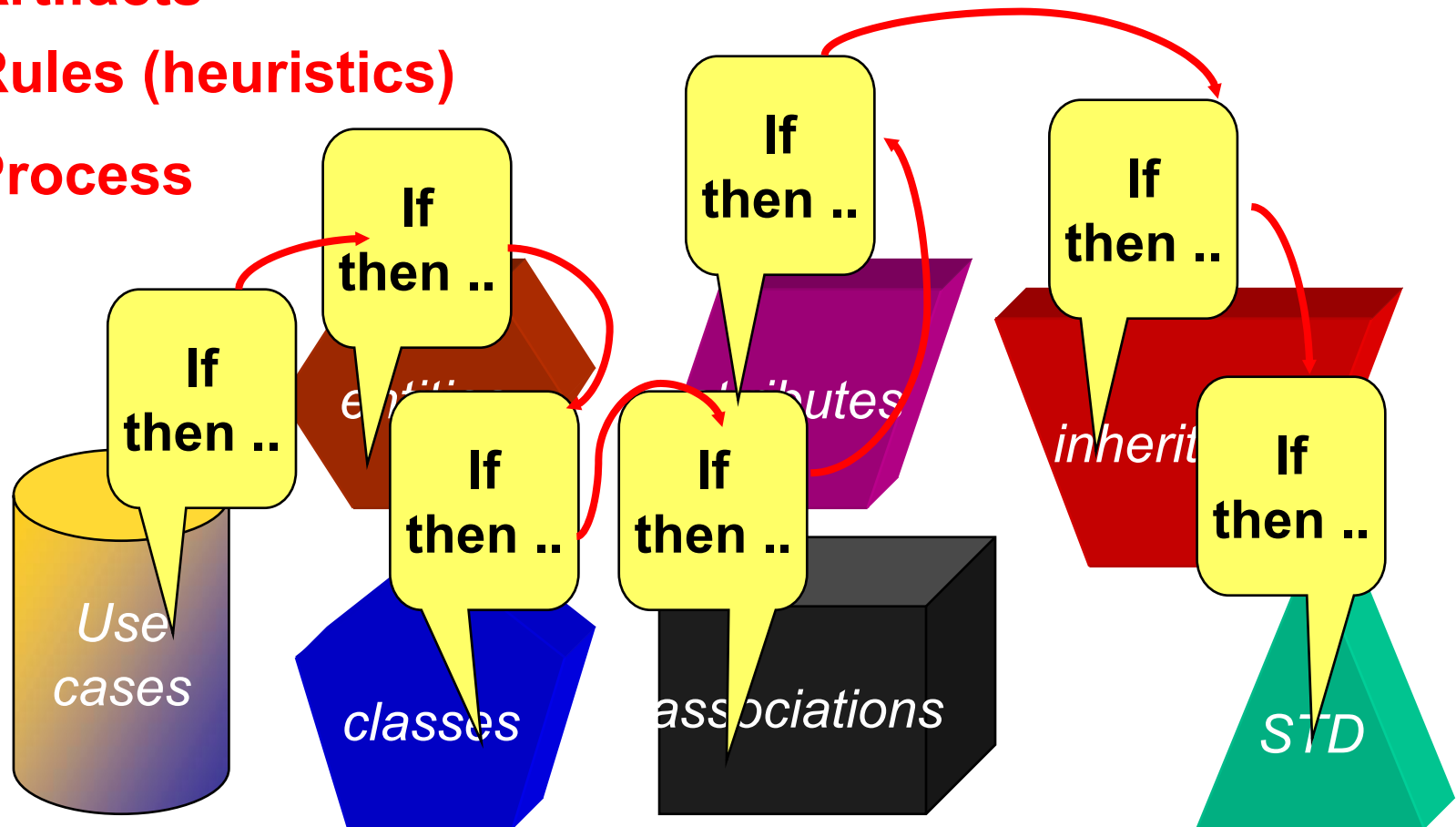


What is a method?

■ Artifacts

■ Rules (heuristics)

■ Process



Example OO Heuristic Rules

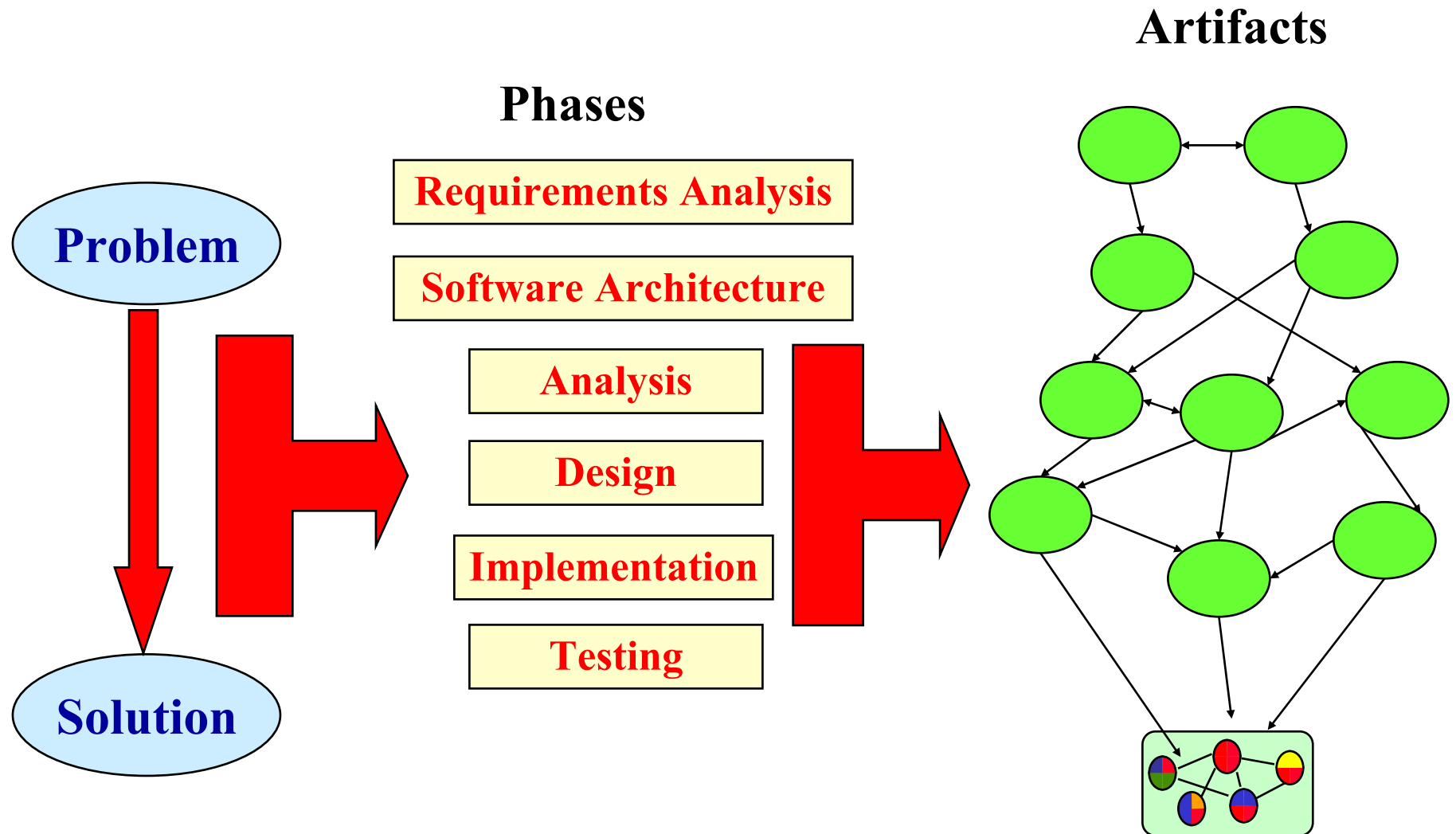
- **Tentative Class identification**
- *Extract* nouns from the problem statement.
- *Select* nouns as tentative classes.
- *Extract* tentative classes from application domain knowledge.
- *Extract* tentative classes from general knowledge.

- **Class identification**
- **Class**
- The identified tentative classes are used to identify classes:
- **IF** tentative class <isRedundant>
THEN eliminate tentative class.
- **IF** tentative class <isIrrelevant>
THEN eliminate tentative class.
- **IF** tentative class <isVague>
THEN eliminate tentative class.
- **IF** tentative class <isAttribute>
THEN select tentative class as Attribute.
- **IF** tentative class <isOperation>
THEN eliminate tentative class.
- **IF** tentative class <isRole>
THEN eliminate tentative class.
- **IF** tentative class <isImplementationConstruct>
THEN eliminate tentative class.
- Select remaining tentative classes as right classes.

- **Association**
- Right associations are derived from the tentative associations and from the application domain:
- **IF** a tentative association <includesEliminatedClass>
THEN eliminate tentative association **or** restate tentative association.
- **IF** a tentative association <isIrrelevant>
THEN eliminate tentative association.
- **IF** a tentative association <isImplementationConstruct>
THEN eliminate tentative association.
- **IF** a tentative association <isTransientEvent>
THEN eliminate tentative association.
- **IF** a tentative association <isDerivedAssociation> **and** <multiplicityConstraintNotImportant > **and not** <useful>
THEN eliminate tentative association.
- **IF** a tentative association <representsConditionOnAttribute>
THEN eliminate tentative association.
- Select remaining tentative associations as right associations.
- Add missing overlooked tentative associations from application domain.
-

B. Tekinerdogan and M. Aksit, *Providing Automatic Support for Heuristic Rules of Methods*, in Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), LNCS 1543, ECOOP'98 Workshop Reader, Springer Verlag, pp. 496-498, July 1998.

Artifacts in Software Engineering



Monolithic software

- Large program
 - consisting of one module
 - difficult to understand
 - difficult to reuse
 - difficult to adapt
 - difficult to ...
- Lack of Modularity

```
/**-----  
import InventoryItem;  
import java.util.StringTokenizer;  
import java.io.*;  
  
public class Inventory  
{  
    //-----  
    // Reads data about a store inventory from an input file,  
    // creating an array of InventoryItem objects, then prints them.  
    //-----  
    public static void main (String[] args)  
    {  
        final int MAX = 100;  
        InventoryItem[] items = new InventoryItem[MAX];  
        StringTokenizer tokenizer;  
        String line, name, file="inventory.dat";  
        int units, count = 0;  
        float price;  
  
        try  
        for (int scan = 0; scan < count; scan++)  
            System.out.println (items[scan]);  
        }  
        catch (FileNotFoundException  
        {  
            System.out.println  
        }  
        catch (IOException  
        {  
            System.out.println  
        }  
    }  
}
```



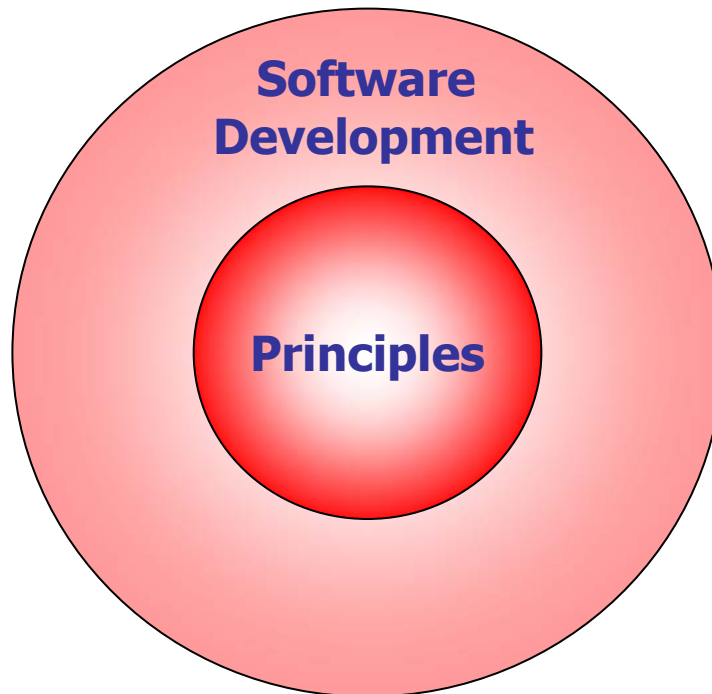
Quality Software

How do you
develop modular software?





Principles of Software Engineering



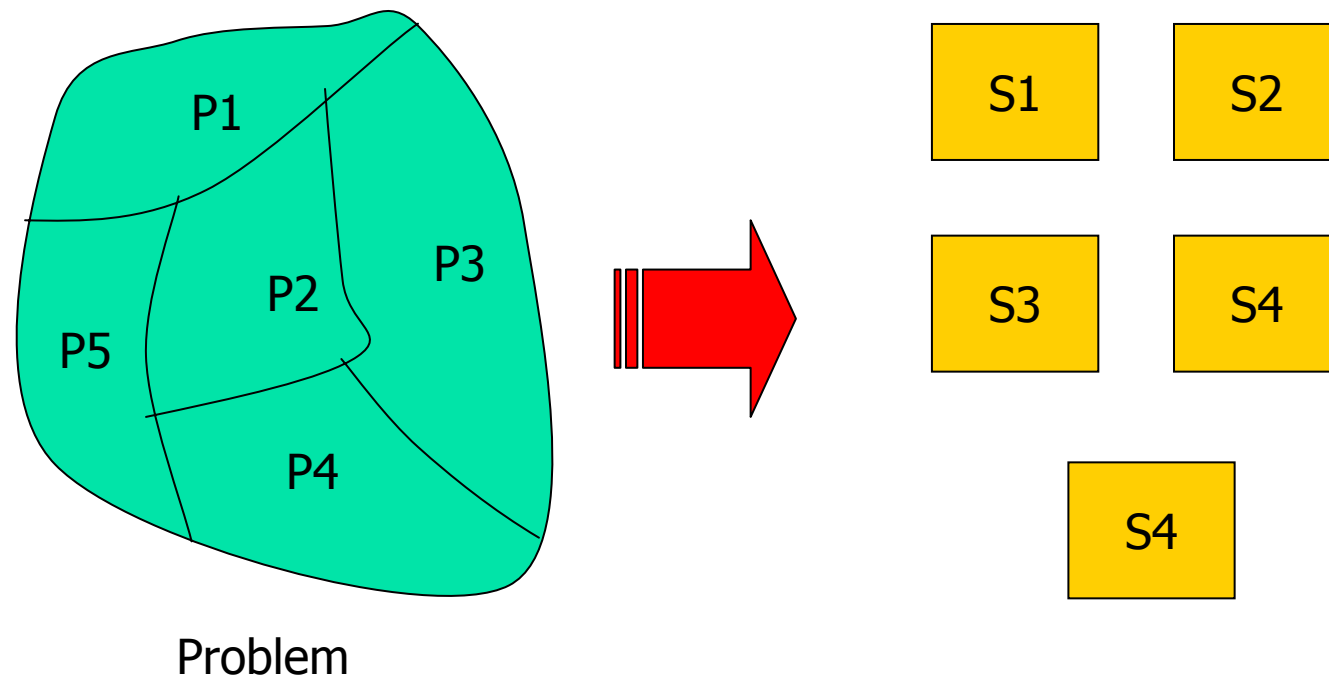
Key Principles

- Decomposition
- Abstraction
- Encapsulation
- Information Hiding
- Separation of Concerns

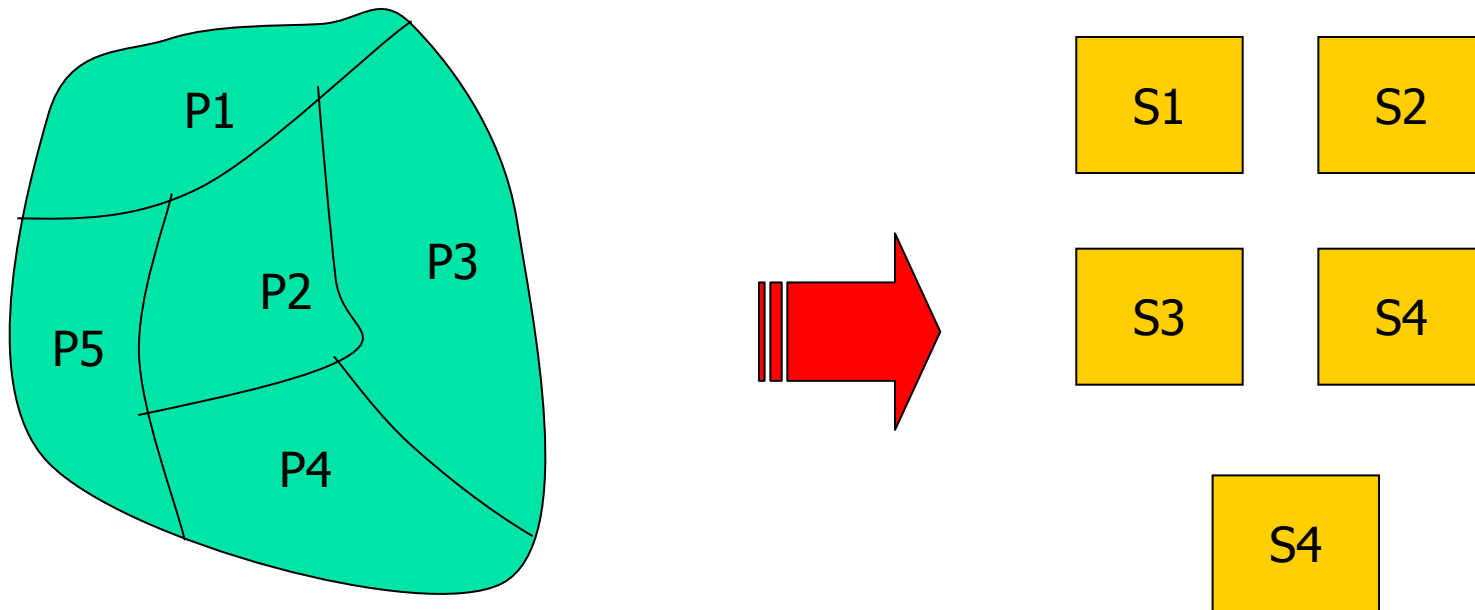
→ Modularity

Decomposition

- 'Divide and conquer'
- Divide software into separately named and addressable components or *modules*.
- Problem decomposition → Solution Decomposition
 - $P \rightarrow P1..Pn \rightarrow S1..Sn \rightarrow S$



Decomposition - Example



Problem: Atomic Transaction Design

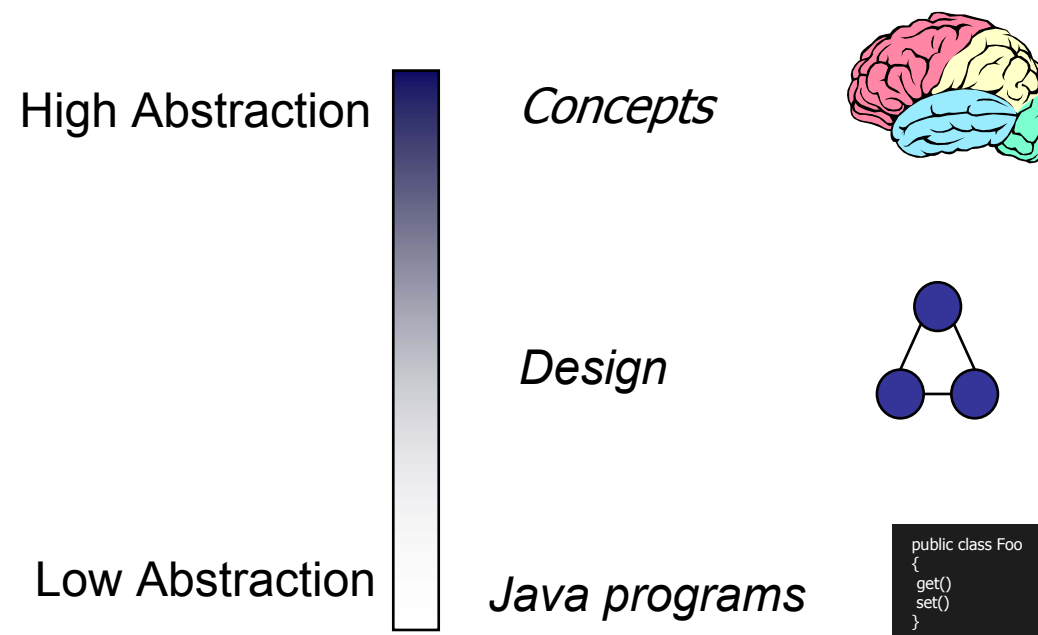
- P1. Transaction Management
- P2. Data Management
- P3. Adapting Transaction Protocols
- P4. Concurrency Control
- P5. Failure Management

Solution: Atomic Transaction Design

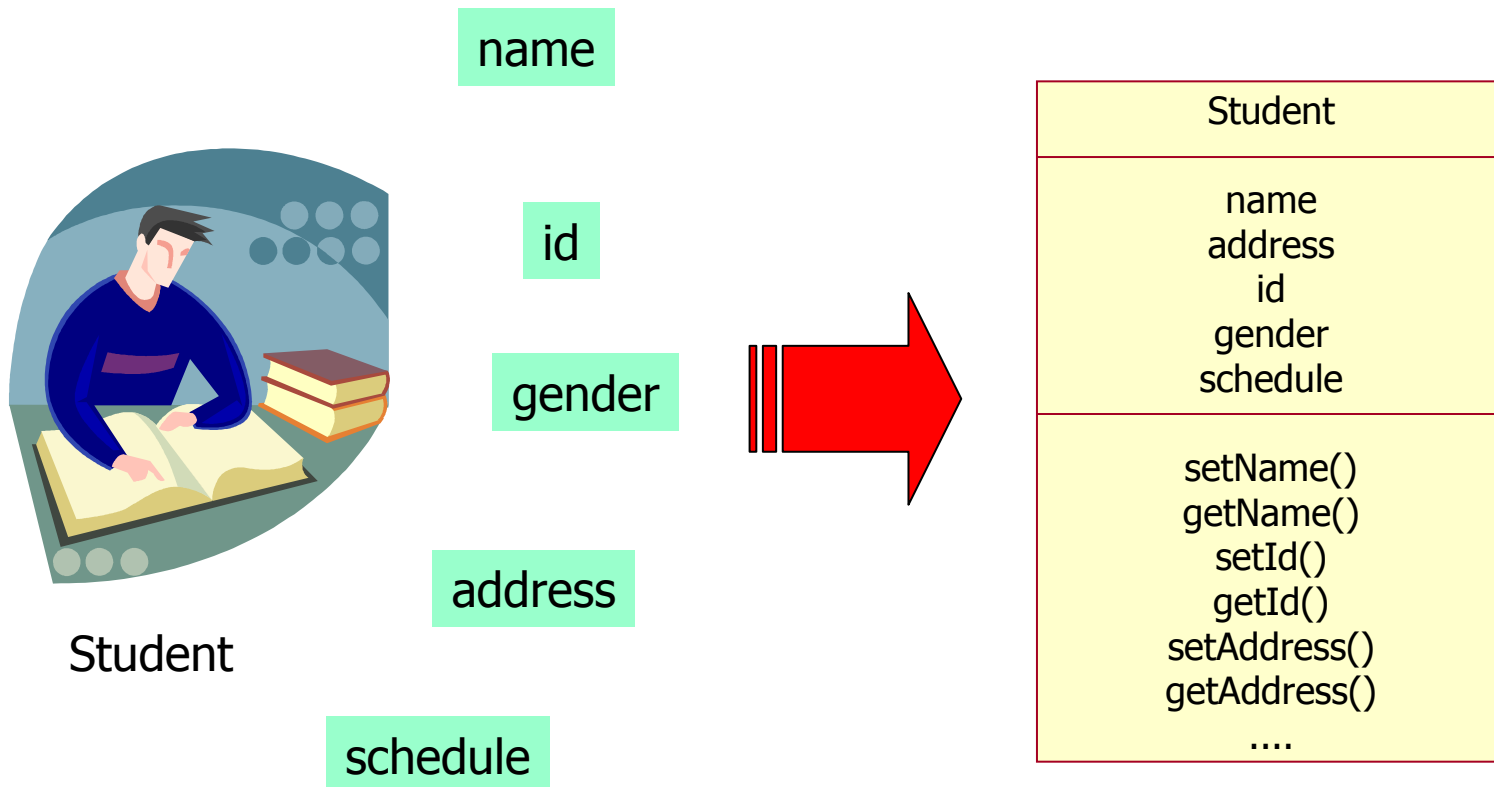
- S1. Transaction Manager
- S2. Data Manager
- S3. PolicyManager
- S4. Scheduler
- S5. RecoveryManager

Abstraction

- Focus only on relevant properties of the problem
- 'Ignore' details.

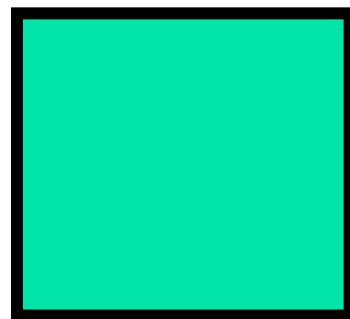


Encapsulation



Information Hiding

- Module implementation details is inaccessible from outside.



Module

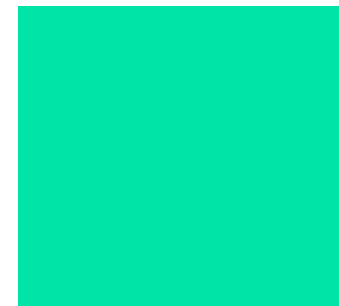
=



Interface

externally visible

+

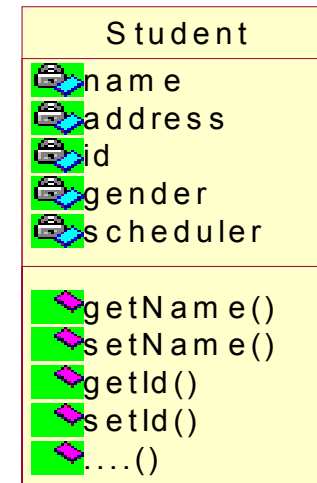


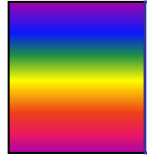
Implementation

only internally visible

Information Hiding - Example

```
public abstract class Student
{
    private Name name;
    private Address address;
    private int id;;
    private String gender;
    private Schedule sch;
}
```





Separation of concerns

- Identification of the various distinct concerns
 - concerns: properties or areas of interest
- Assignment of single concerns to single modules
- At various levels: design and implementation

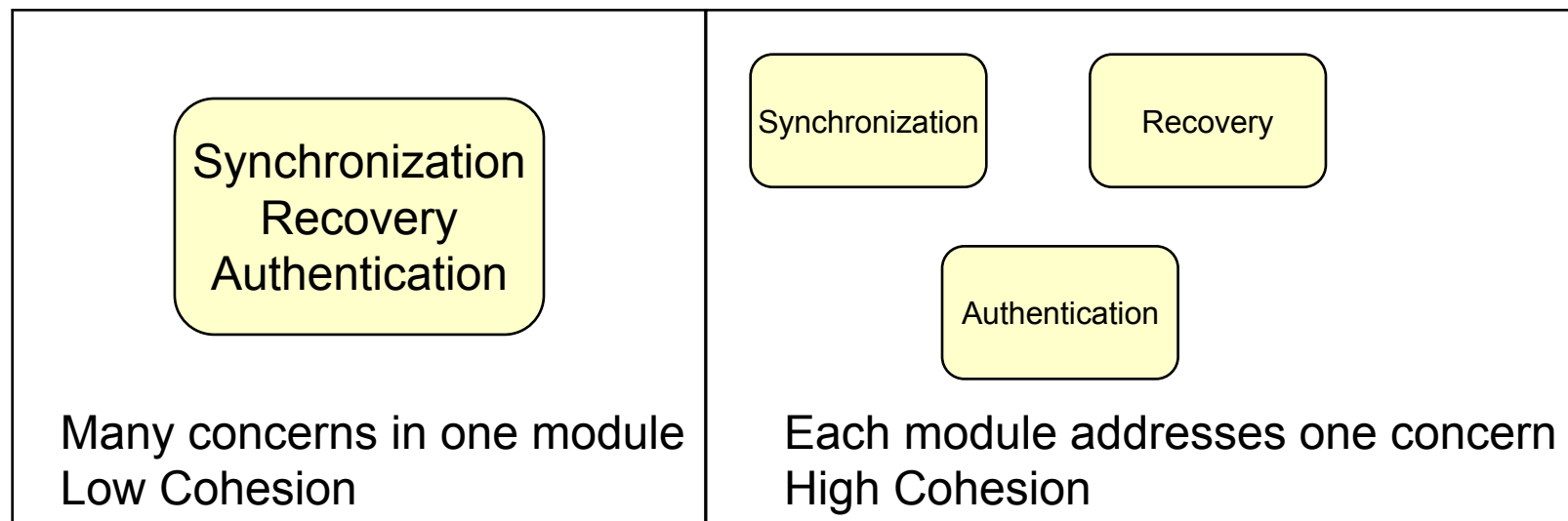


Separation of Concerns applied

- Separate software development into phases each dealing with specific activities (e.g. requirements analysis, design, implementation)
- Separation of different artifacts: class, subsystems, attributes.
- Separation of different design views (static, dynamic, implementation, ...)
- Separation of different roles
- ...

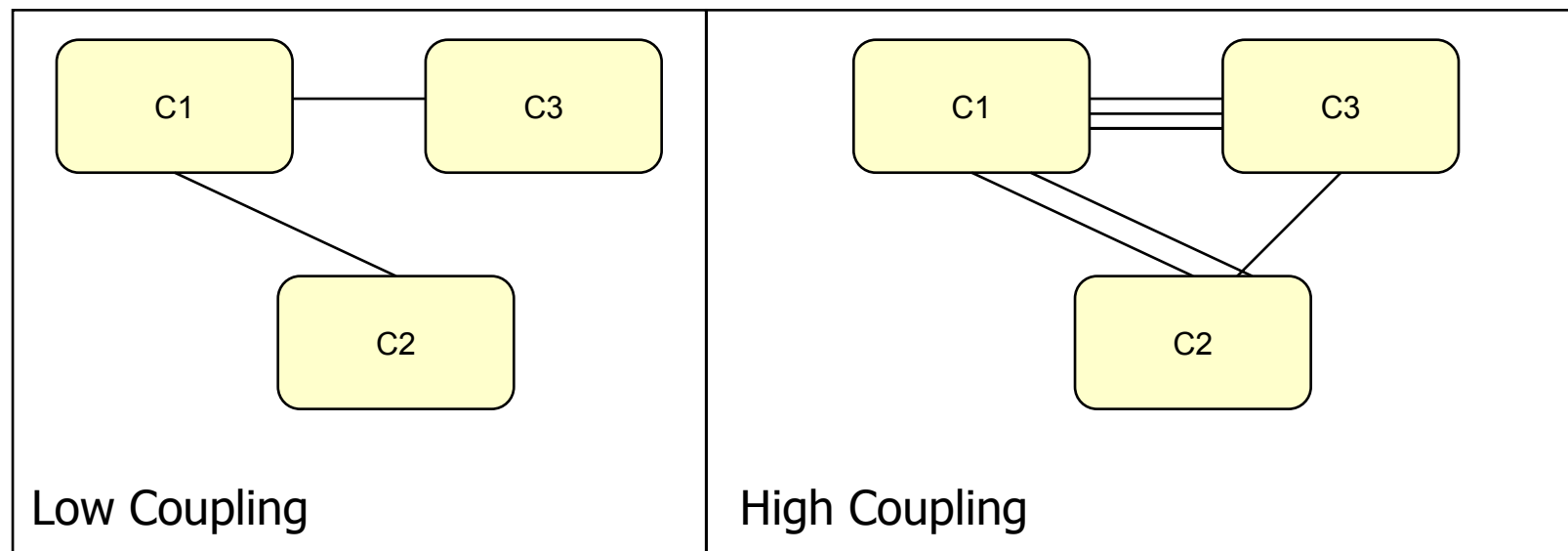
Separation of Concerns - Cohesion

- Cohesive component performs only **one concern/task**
- Maximize cohesion within a component
 - required changes can be easily localized and will not propagate

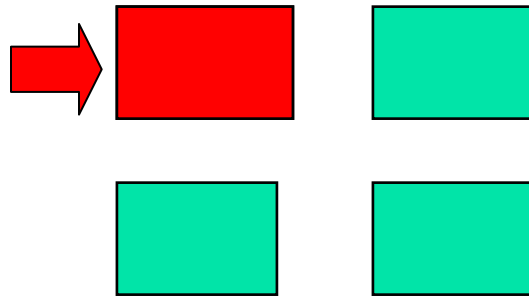


Separation of Concerns - Coupling

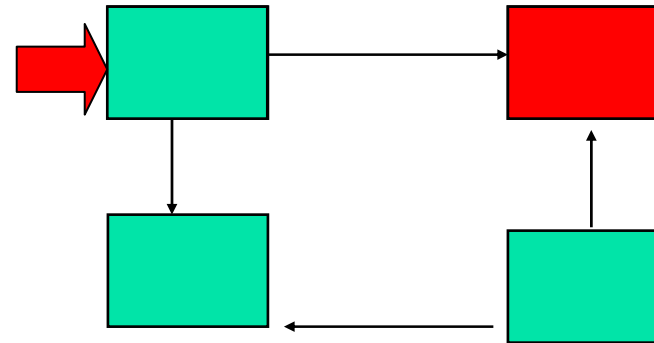
- Two components are independent if they do not have interactions
- Highly coupled components have many dependencies/interactions
- Minimize coupling between components
 - reduces complexity of interactions
 - reduces 'ripple' effect



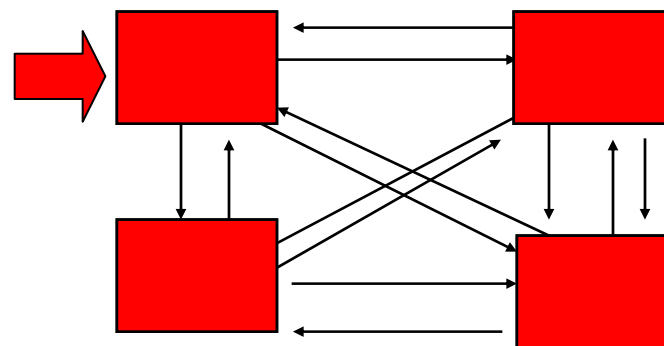
Coupling and Dependency



Uncoupled



Loosely Couple: Some Dependencies



'ripple' effect

Highly Couple: Many Dependencies



Advantages of separation of concerns

- Understandability
- Maintainability
- Extensibility
- Reusability
- Adaptability

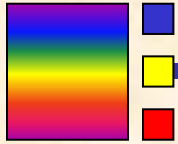
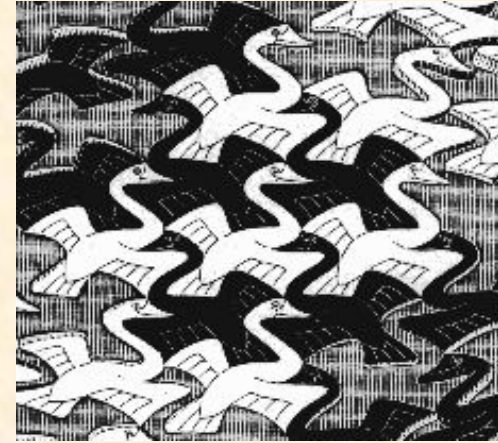
Separation of Concerns directly supports quality factors.

Lack of Separation of Concerns negatively impacts quality factors.



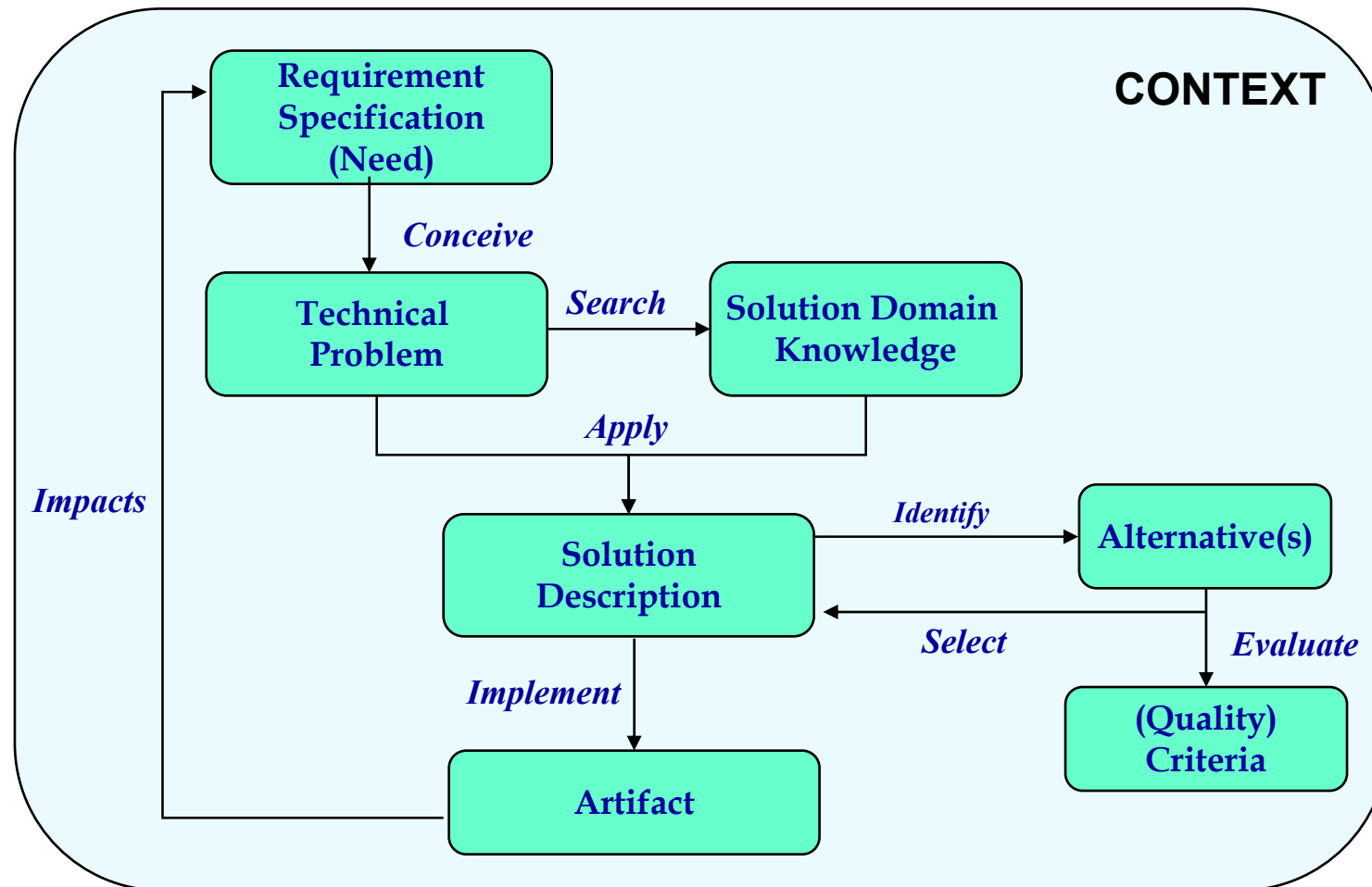
Design Principles → Modularity

- Abstraction
 - Focus only on relevant properties
- Decomposition
 - Divide software into separately named and addressable modules
- Encapsulation
 - Group related things together.
- Information Hiding
 - Hide implementation details from the outside
- Separation of Concerns
 - Ensure that each module only deals with one concern
 - Low Coupling
 - aim for low coupling among the modules
 - High Cohesion
 - aim for high cohesion within one module



Object-Oriented Design Patterns

Engineering=Problem Solving



Mature Engineering adheres to this problem solving model

See: B. Tekinerdogan, Chapter 2, *On the Notion of Software Engineering: A Problem Solving Perspective*, PhD Thesis, University of Twente, 2000.

Observations...

- Many *problems recur*.
- Many problems have the *same solution structure*.
- Exact solution is dependent on the *context*
- A more experienced person can solve new problems faster and better.



Problem Solving - Novice

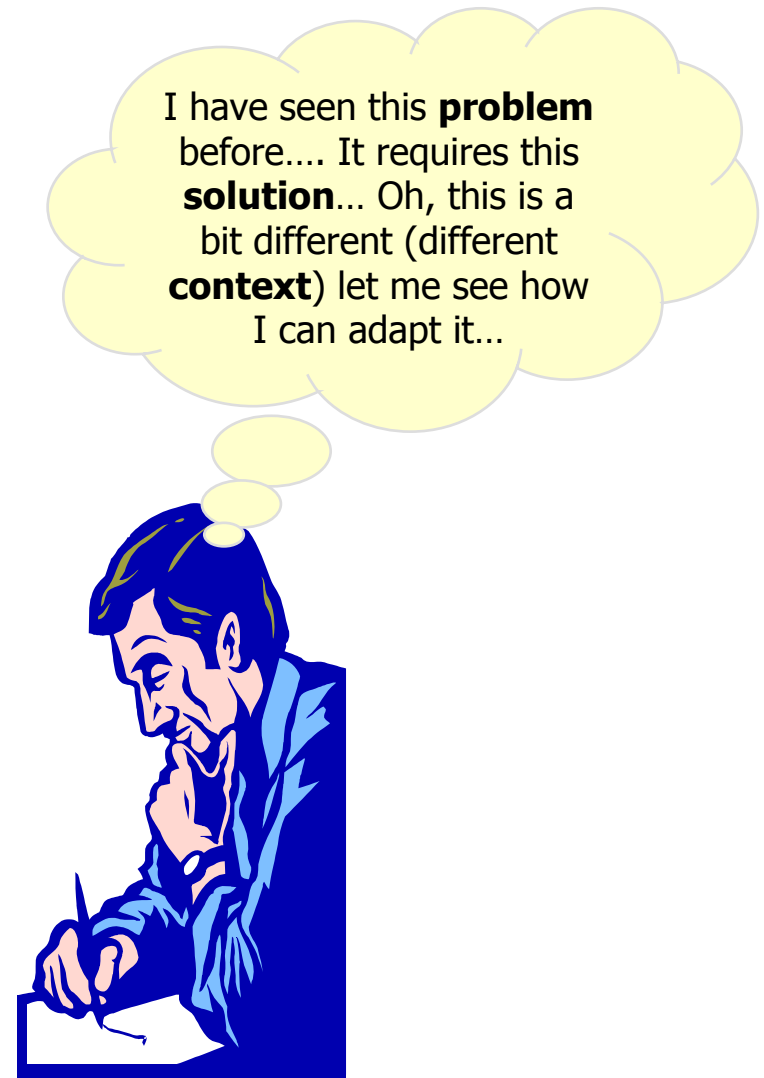
- Knows a few problems
- Knows a few solutions
- Invents new solutions for each problem
- Has seen/operated a few contexts.
- Cannot easily match problems to solutions in different contexts
- Solution is based on trial-and-error



Problem Solving - Expert

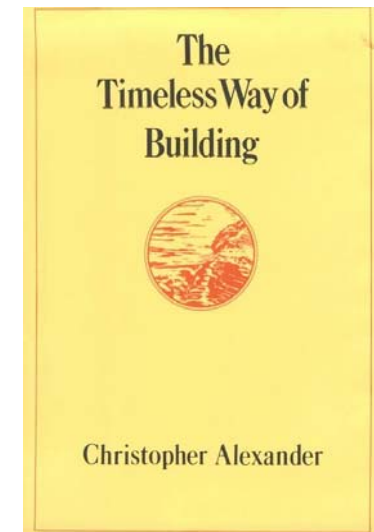
- Knows many problems.
- Knows solutions to these problems.
- Recalls generic solution when encountering new problems
- Knows various different contexts
- Knows how to apply this knowledge

- In short: Expert knows many *patterns...*

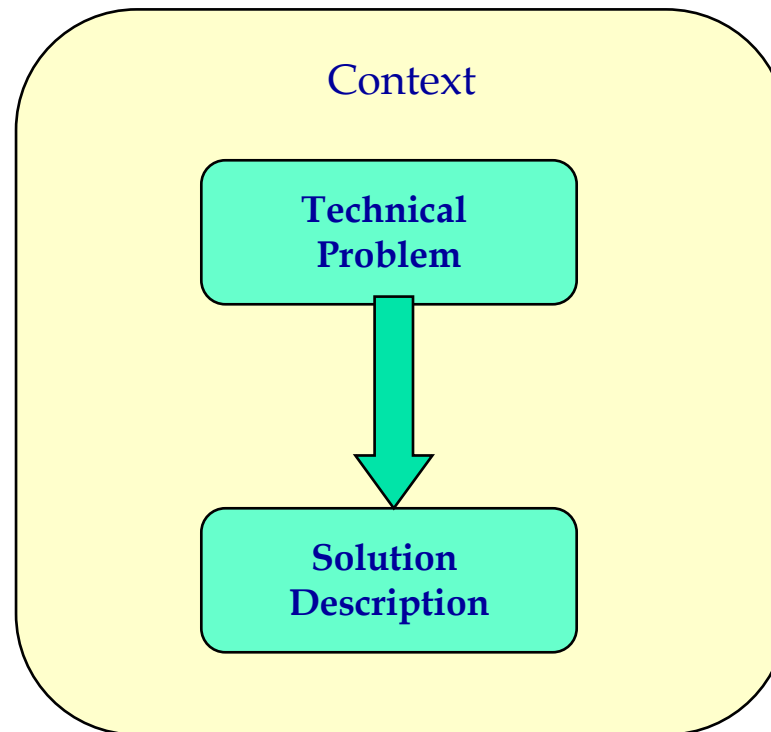


Alexander's Timeless Way of Building

*When a person is faced with an act of design, what he does is governed entirely by the pattern language which he has in his mind at that moment. ... the pattern languages in each mind are evolving all the time, as each person's experience grows. ... His act of design, whether humble or gigantically complex, is governed entirely by the **patterns** ... and his ability to combine these **patterns** to form a new design.*



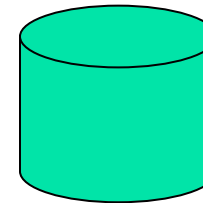
Pattern: Problem-Solution-Context



Christopher Alexander, *Timeless Way of Building*:
a **pattern** is a three-part rule, which expresses a relation
between a certain **context**, a **problem**, and a **solution**.

Describing Patterns

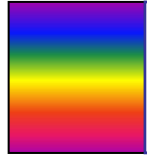
- Name
 - meaningful name
- Problem
 - statement of the intent/goal
- Context
 - preconditions and the pattern's applicability
- Forces
 - description of relevant forces and constraints
- Solution
 - a structure
- Example
 - sample application of the pattern
- Consequences
 - state of the system after applying the pattern
- Rationale
- Related Patterns
 - static and dynamic relations to other patterns



Pattern Base

This is the pattern

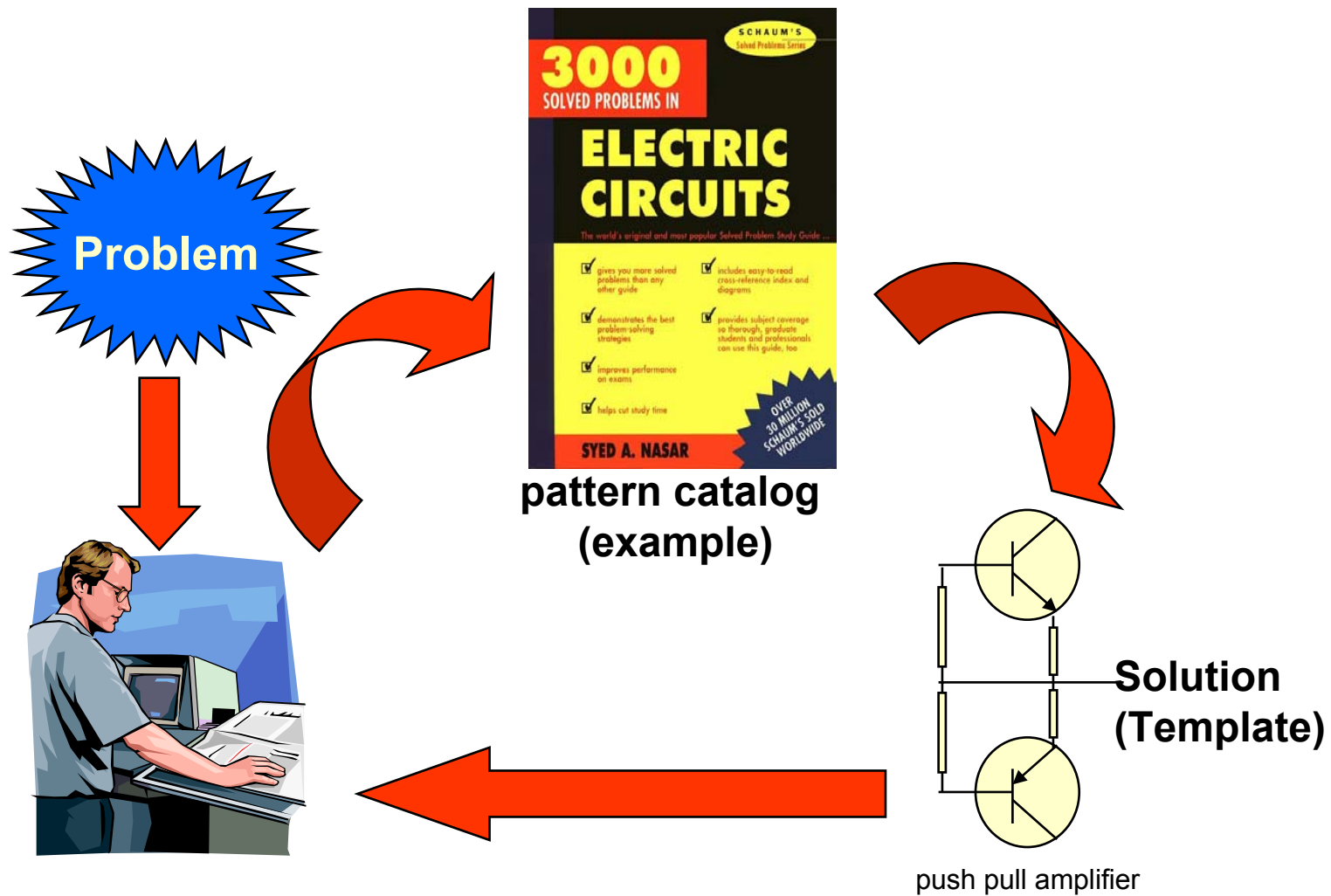




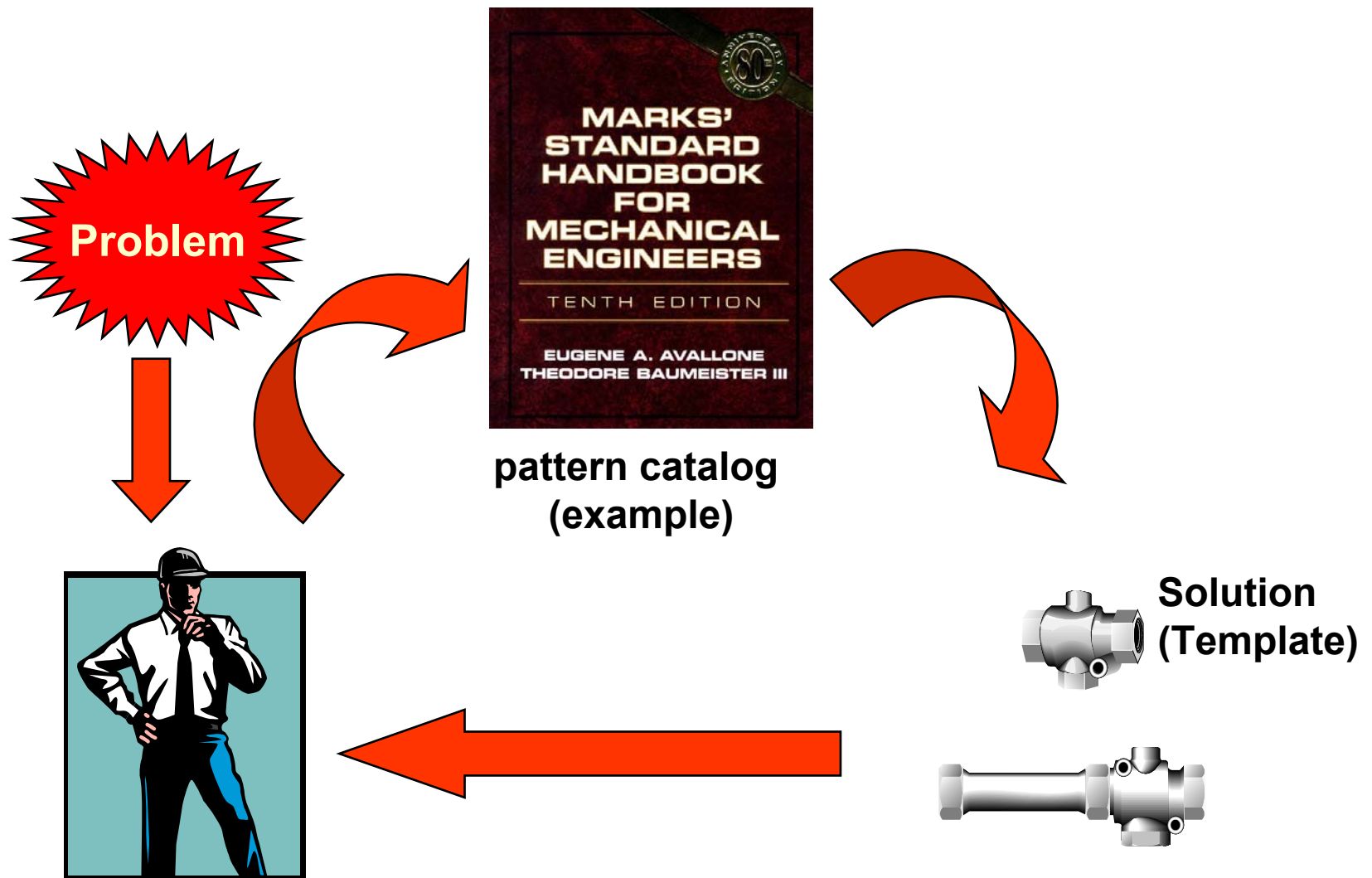
Patterns in Engineering

- All mature engineering disciplines have pattern catalogs
- with patterns that have been discovered by experts
- and which show proven quality solutions
- to recurring problems

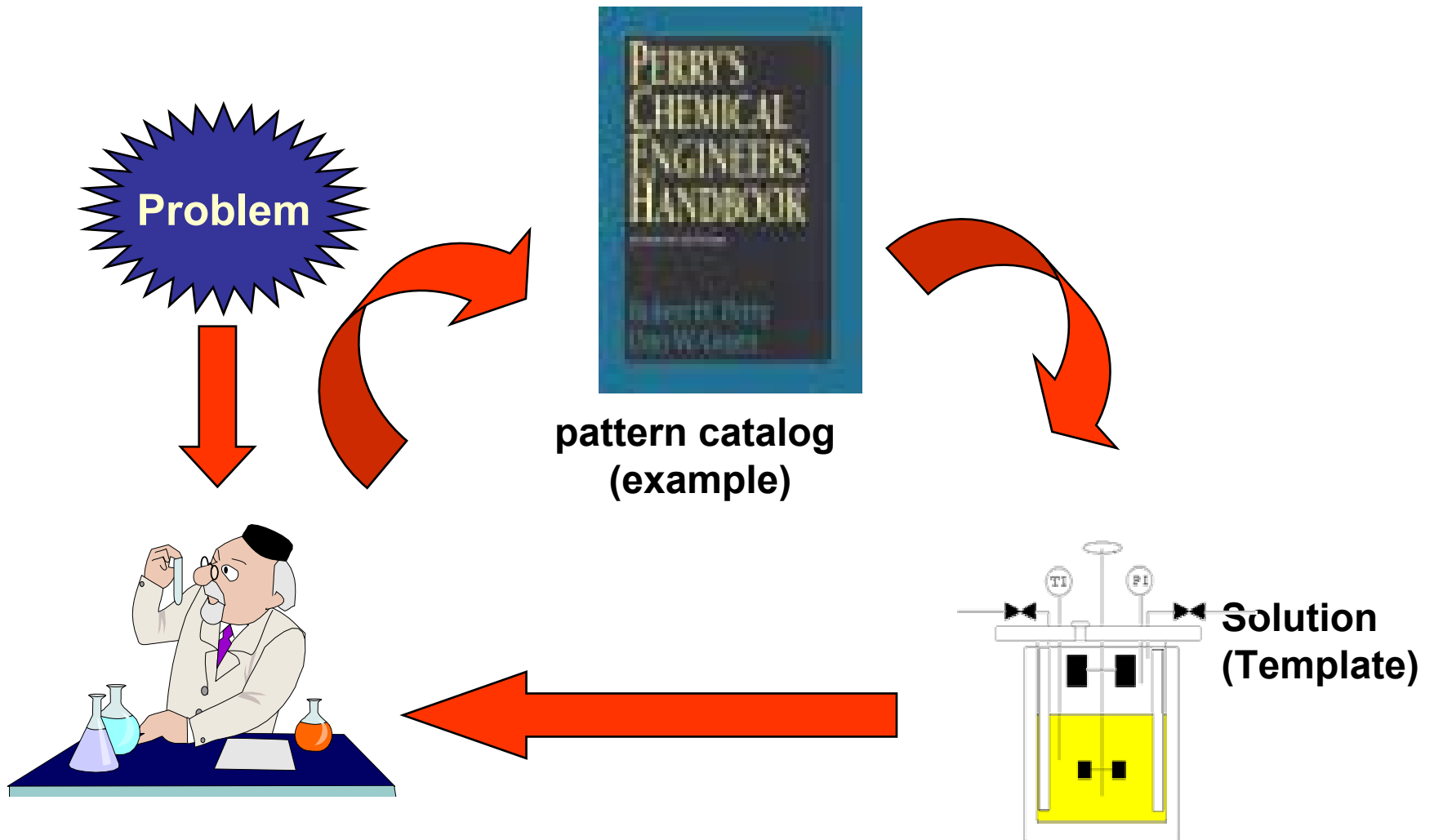
Electrical Engineering Patterns



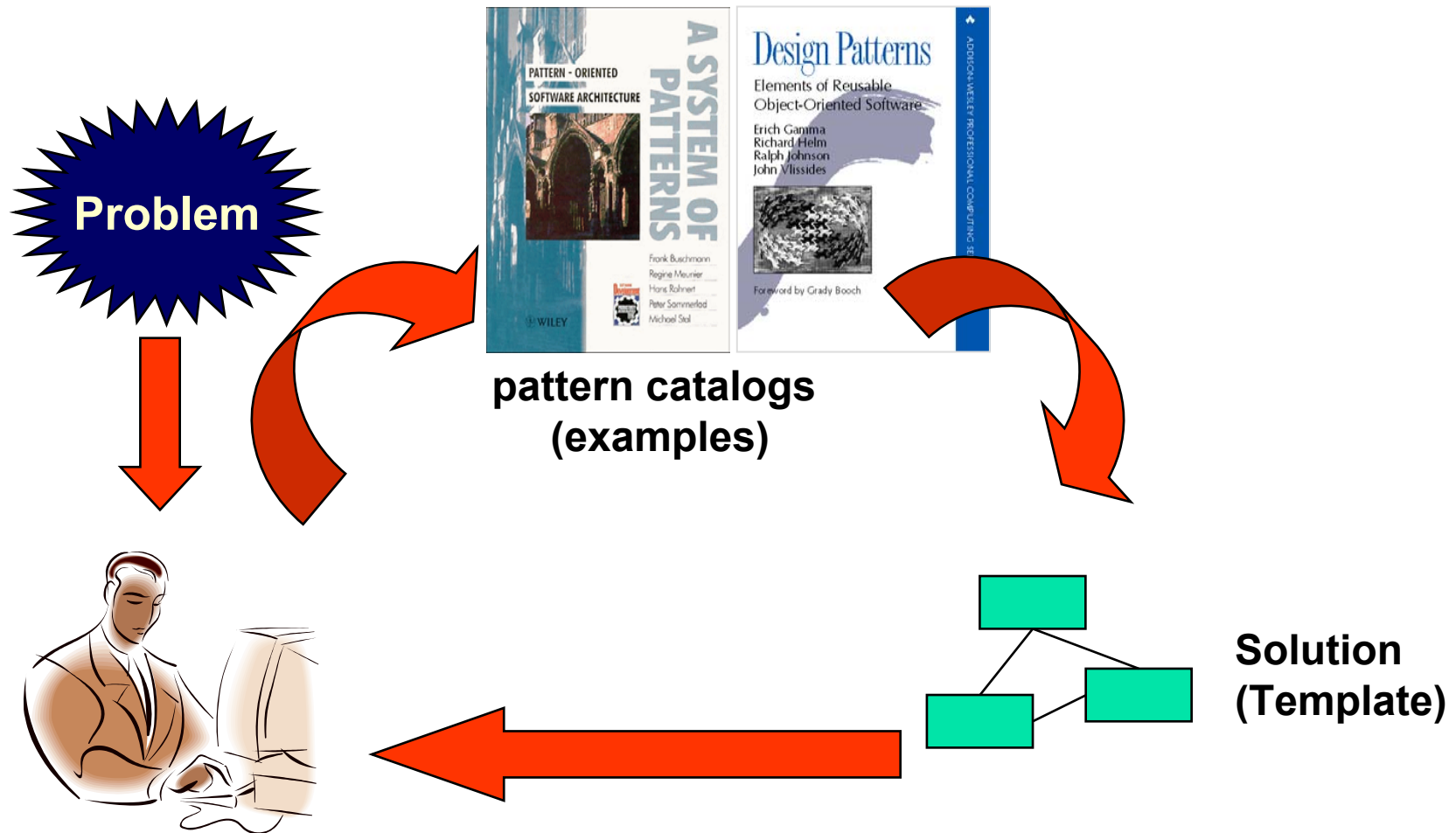
Mechanical Engineering Patterns



Chemical Engineering Patterns

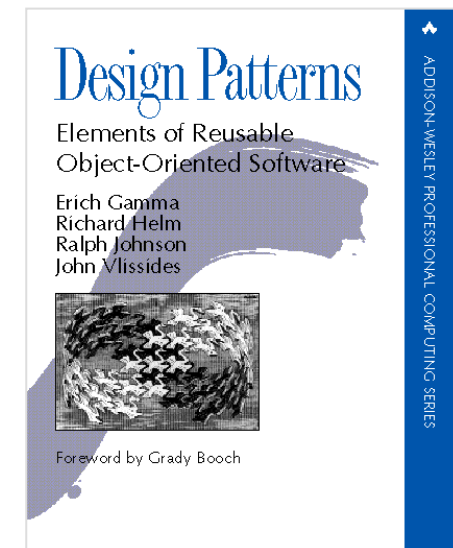


Software Engineering Patterns

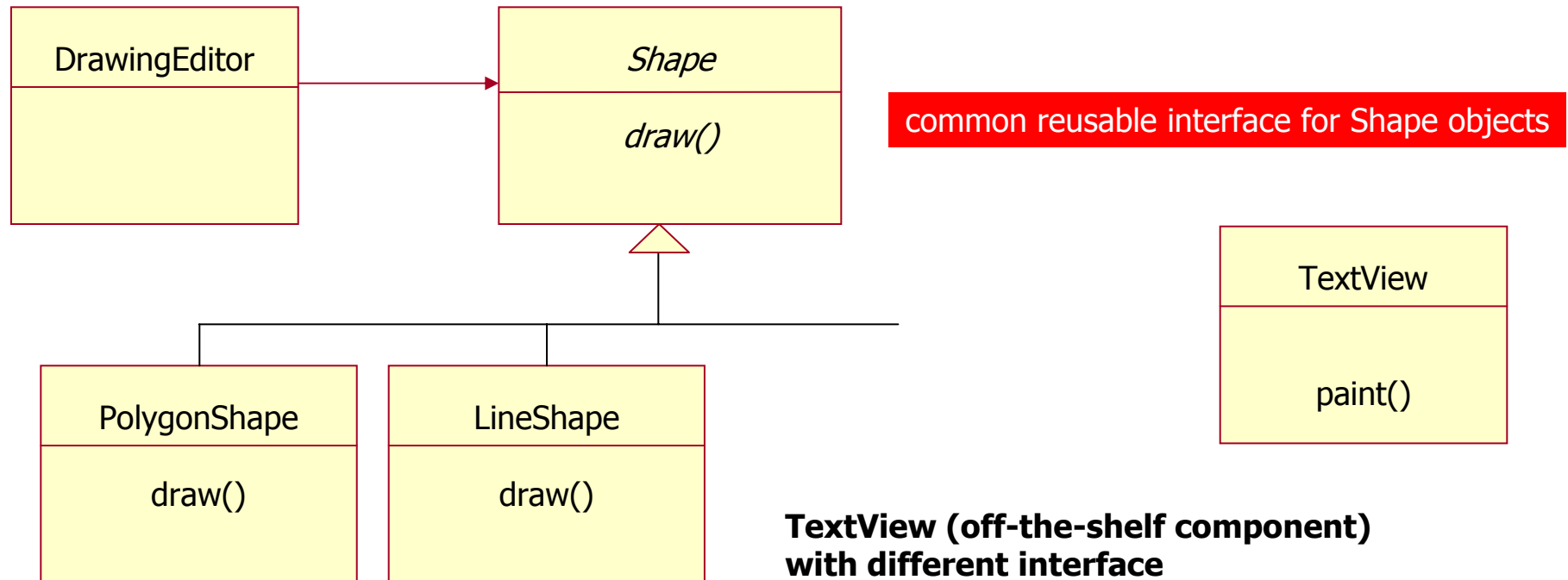


Software Design Patterns

- Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al., 1995
- A *design pattern* provides a scheme for refining the architectural entities of a software system. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- 23 object-oriented design patterns
- Design pattern community
 - <http://hillside.net/patterns/>
- PLOP conferences/workshops



Example - Adapter Design Pattern - Problem

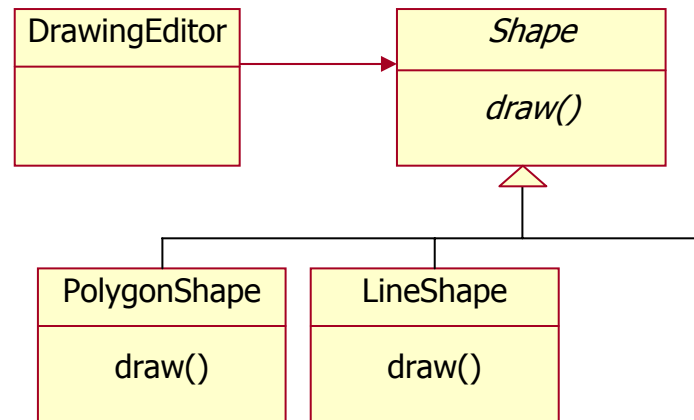


**TextView (off-the-shelf component)
with different interface**

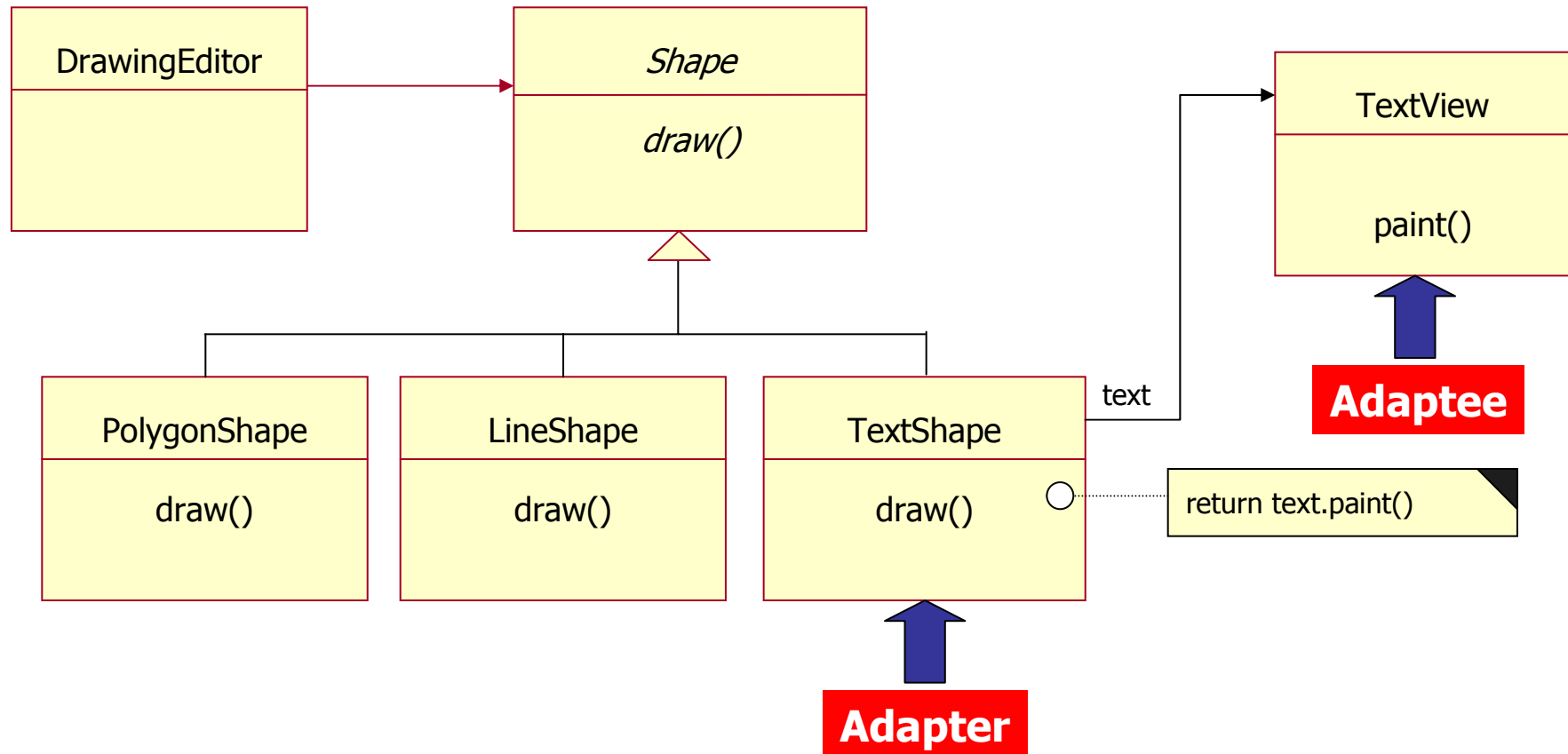
**How to use TextView using the common
interface?**

Adapter – Concerns

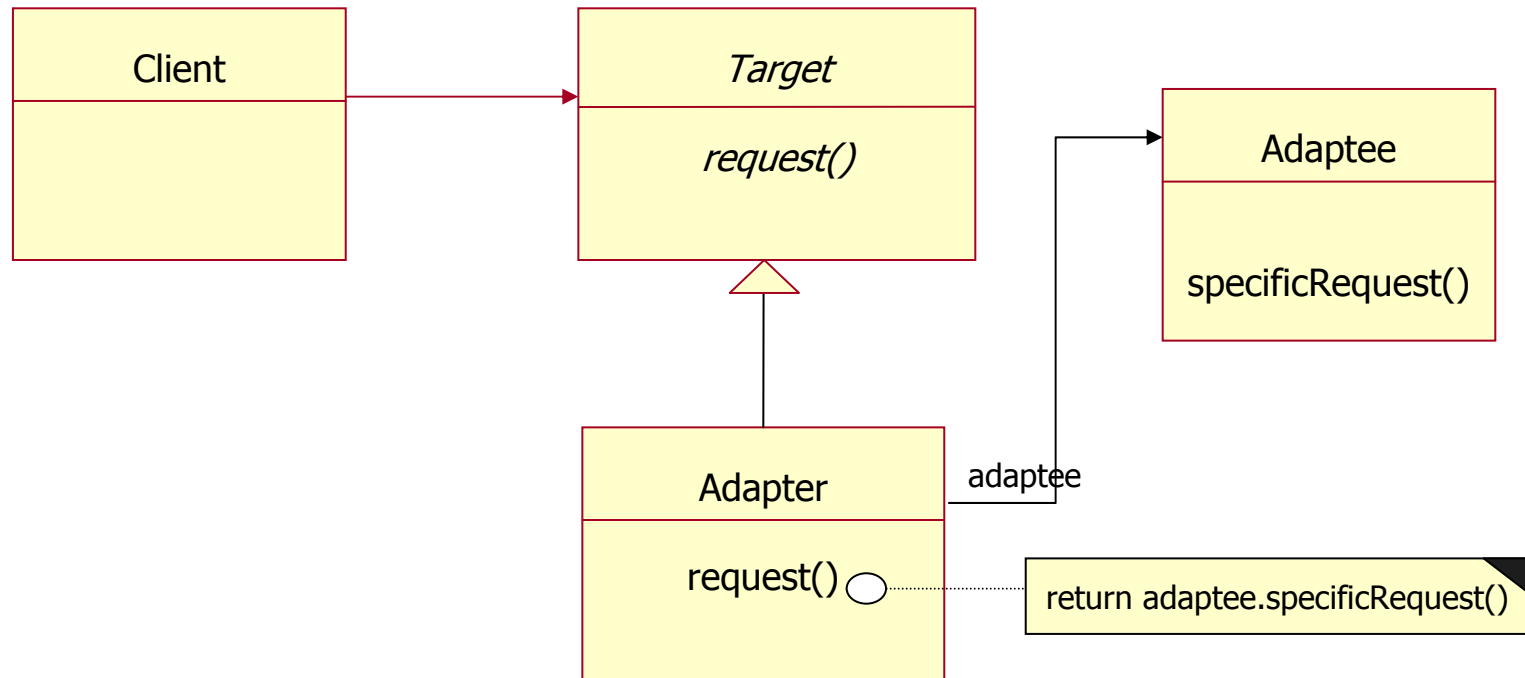
- You want to use an existing class, but its interface does not match the one you need.
- You want to create a reusable class that cooperates with classes that do not necessarily have compatible interfaces.



Adapter (Object) – Solution - Example



Adapter (Object) – Solution - Structure





Catalog of 23 Design Patterns

Creational

Abstract Factory

Builder

Factory Method

Prototype

Singleton

Structural

Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Behavioral

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Observer

State

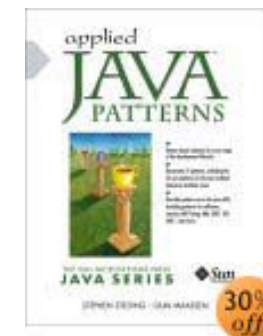
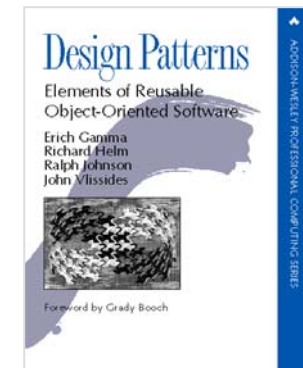
Strategy

Template Method

Visitor

Categorization of Software Patterns

- Architectural Patterns (Styles)
 - Patterns for gross level structure of the system
- Design Patterns
 - provides schemes for **refining** the architecture.
- Programming Patterns (Idioms)
 - provides schemes for mapping the design to a specific programming language





Conclusions

- For designing quality software several design principles must be followed.
- The key design principles are abstraction, decomposition, encapsulation, information hiding and separation of concerns.
- Object-Oriented design patterns apply these principles in the best possible way using the OO concepts, and the given problem.
- Application of patterns will improve modularity
- But...



Aspect-Oriented Software Development

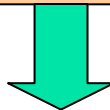
Crosscutting

Scattering and Tangling

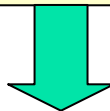
Aspect

Principles, Design Patterns, Aspects

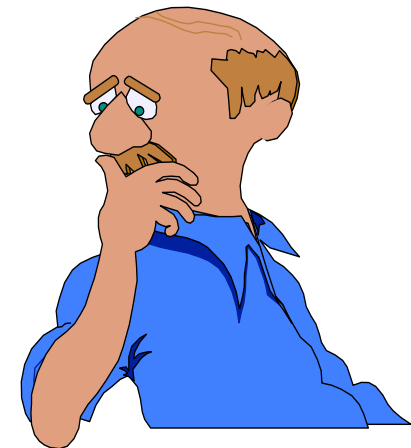
1. Apply Design Principles

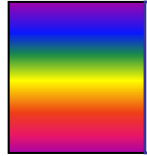


2. Apply Design Patterns



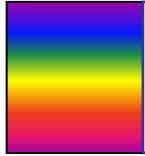
3. Identify Aspects/
Apply AO techniques





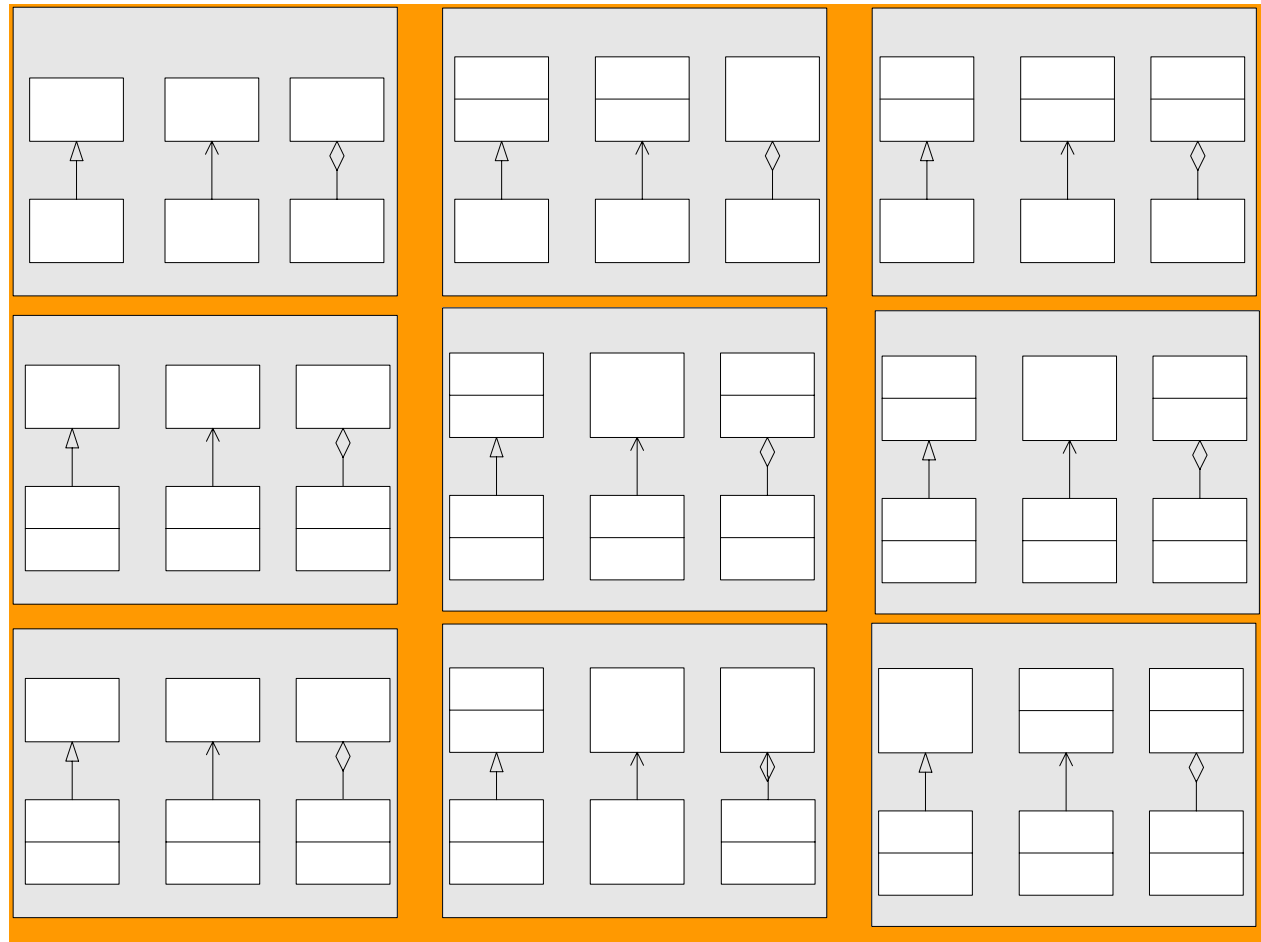
Example - Figure Editor

- A *figure* consists of several *figure elements*. A figure element is either a *point* or a *line*. Figures are drawn on *Display*. A point includes X and Y coordinates. A line is defined as two points.
- Provide object-oriented design in UML.
- Apply principles of abstraction, separation of concerns, modularity, encapsulation. Ensure that the components are cohesive and loosely coupled.

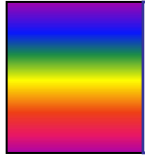


Object-Oriented Model

- Concepts
 - Class
 - Operation
 - Attribute
- Relations
 - Association
 - Aggregation
 - Inheritance



B.Tekinerdogan, Chapter 3, Synthesis-Based Software Architecture Design, PhD Thesis

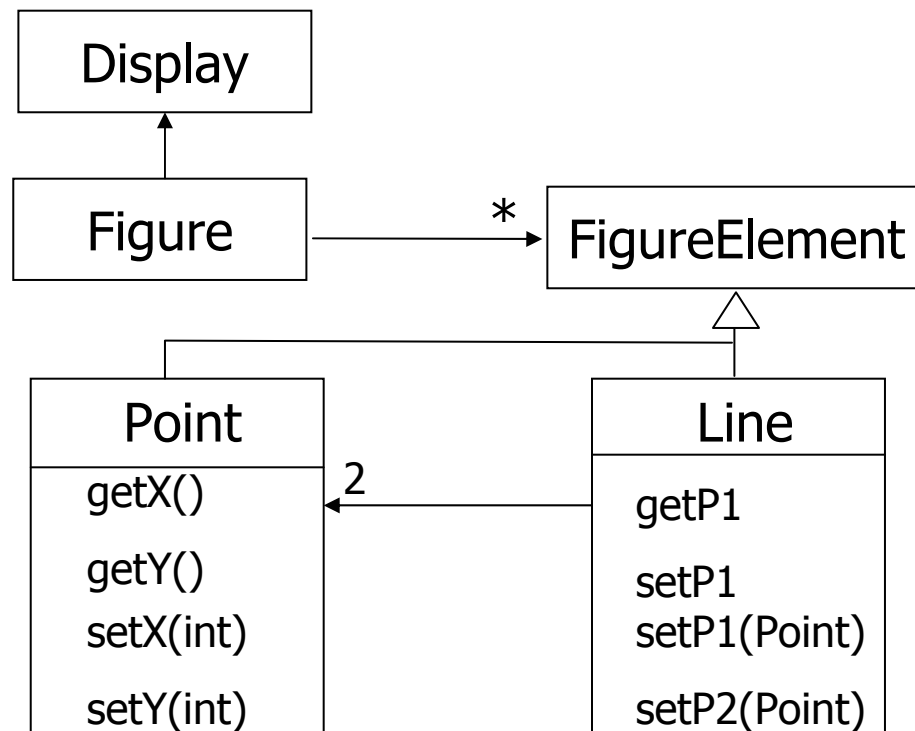


Example - Figure Editor - Requirements

Requirements:

- A *figure* consists of several *figure elements*. A figure element is either a *point* or a *line*. Figures are drawn on *Display*. A point includes X and Y coordinates. A line is defined as two points.

Example - Figure Editor - Design

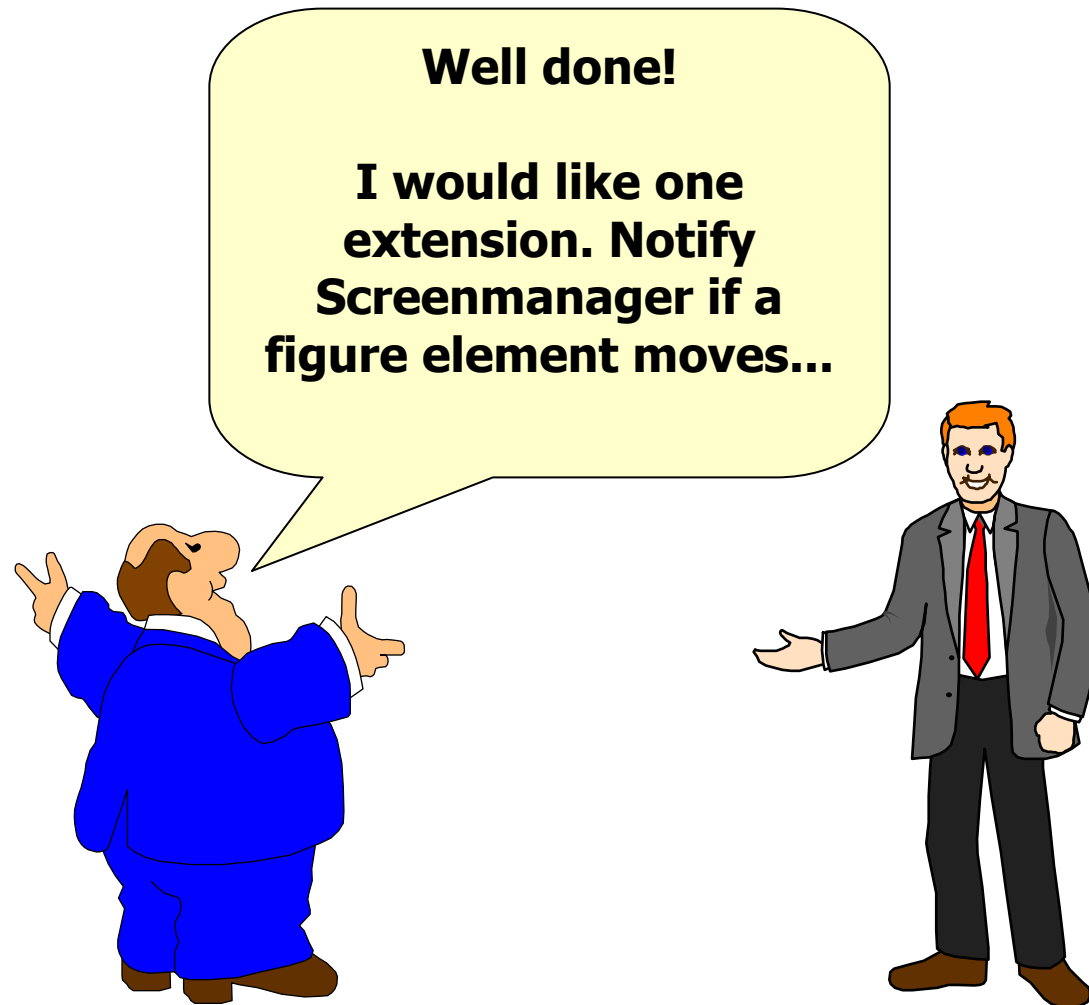


Components are

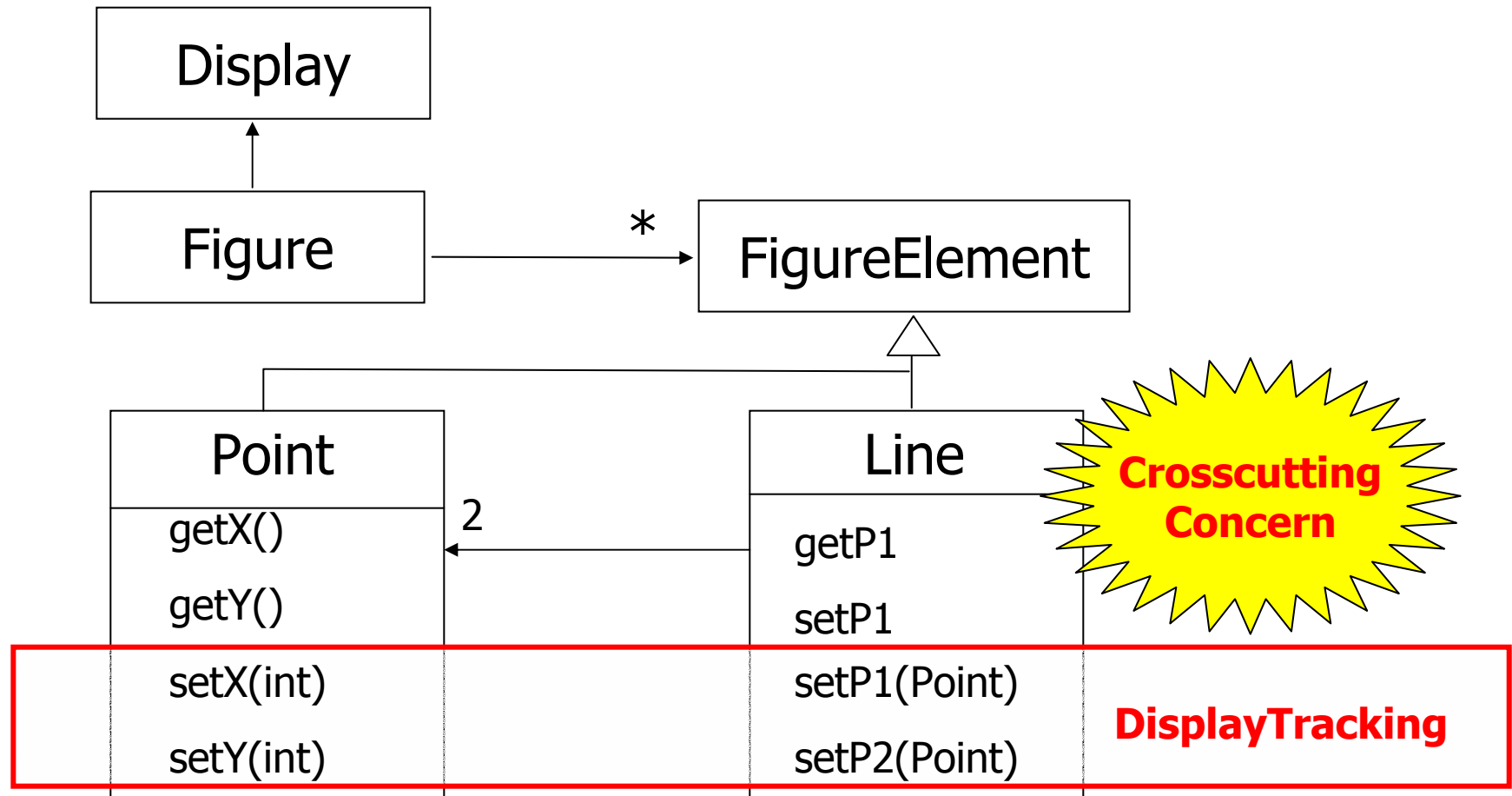
- Cohesive
- Loosely Coupled
- Have well-defined interfaces (abstraction, encapsulation)

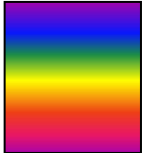
Nice Modular Design!

Exercise - Figure Editor Example – cont'd

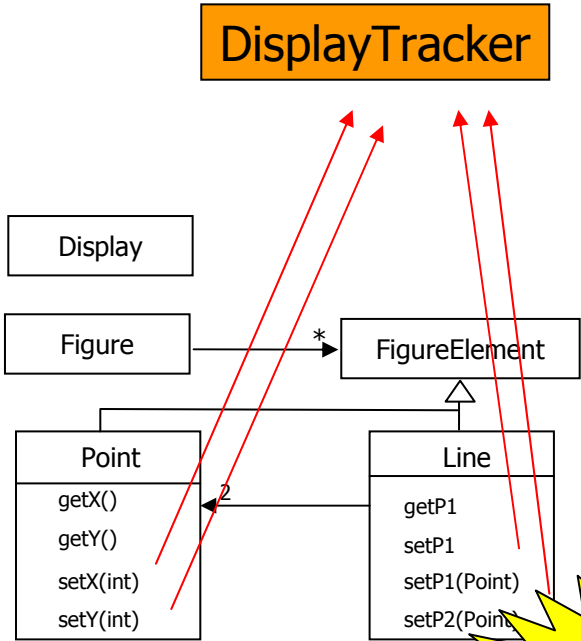


Crosscutting Concern - Example





Example: Display Tracking



```

class DisplayTracker {
    static void updatePoint(Point p)
    {
        this.display(p);
        ....
    }
    static void updateLine(Line l)
    {
        this.display(l);
        ....
    }
}
  
```

Tangling Code

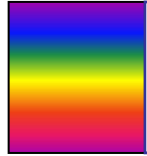
Scattered Concern

```

class Point {
    void setX(int x) {
        DisplayTracker.updatePoint(this);
        _x = x;
    }
}
  
```

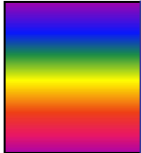
```

class Line {
    void setP1(Point p1) {
        DisplayTracker.updateLine(this);
        _p1 = p1;
    }
}
  
```

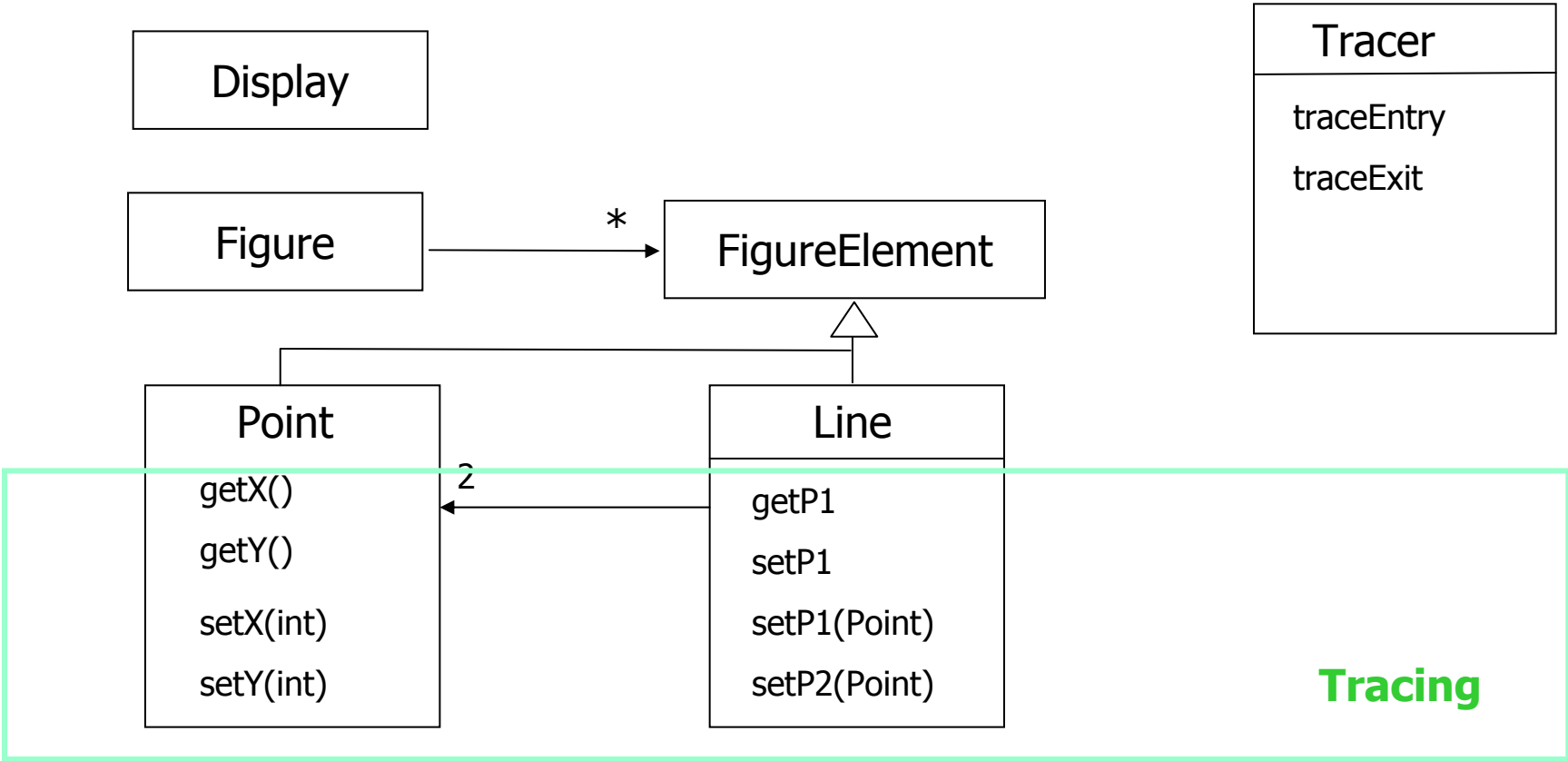


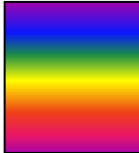
Example - Tracing

- Enhance your design to trace the execution of all operations...

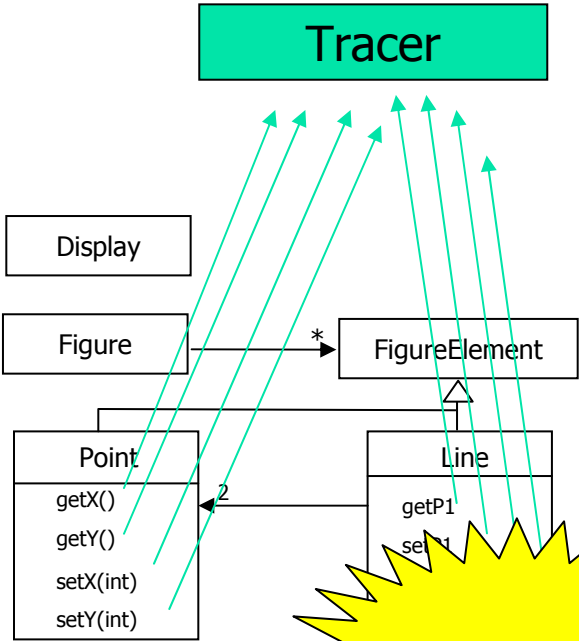


Example - Tracing - Design





Example - Tracing



```

class Tracer {

  static void traceEntry(String str)
  {
    System.out.println(str);
  }
  static void traceExit(String str)
  {
    System.out.println(str);
  }
}
  
```

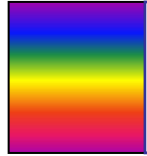


```

class Point {
  void setX(int x) {
    Tracer.traceEntry("Entry Point.set");
    _x = x;
    Tracer.traceExit("Exit Point.set");
  }
}
  
```

```

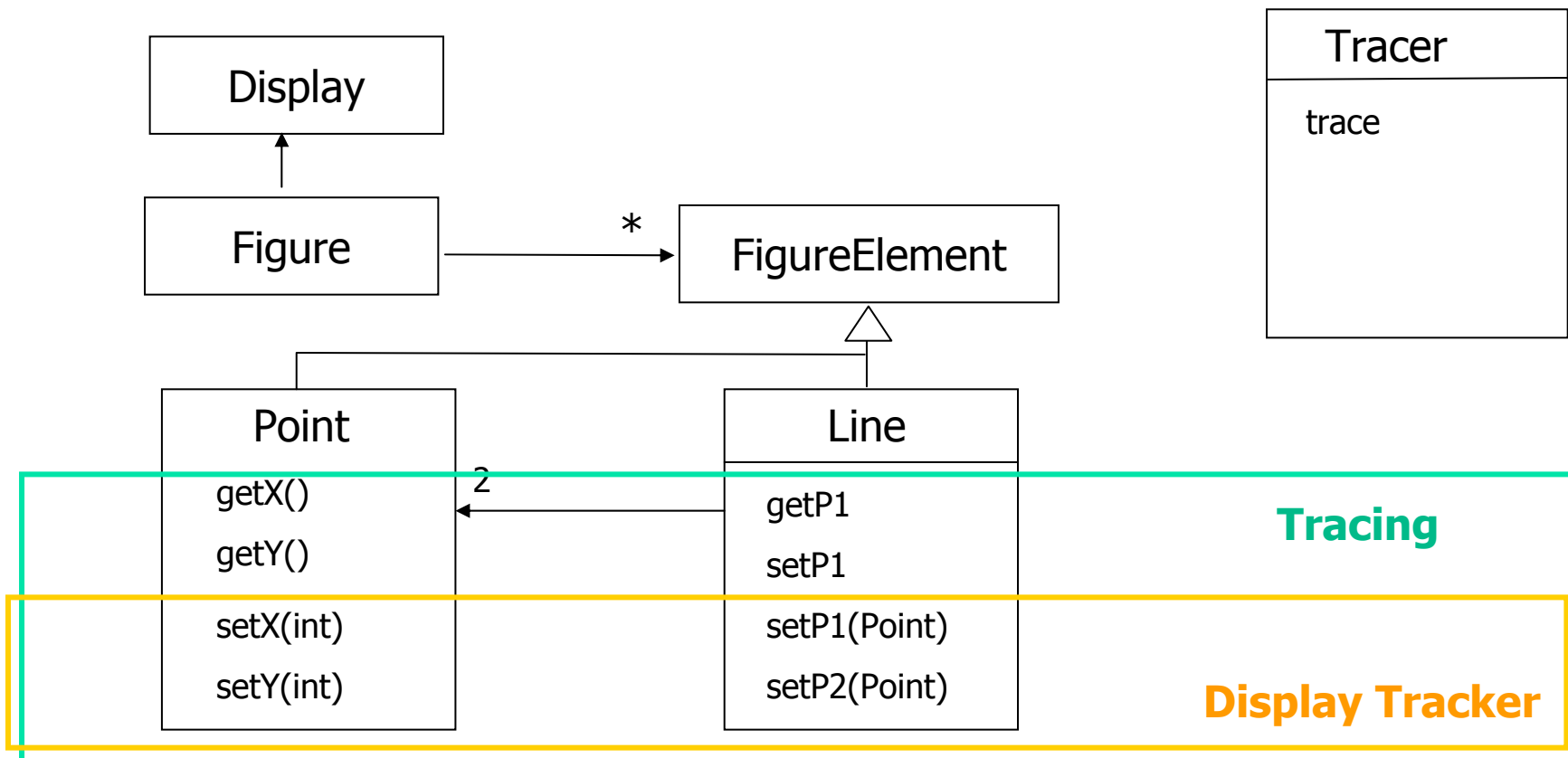
class Line {
  void setP1(Point p1 {
    Tracer.traceEntry("Entry Line.set");
    _p1 = p1;
    Tracer.traceExit("Exit Line.set");
  }
}
  
```



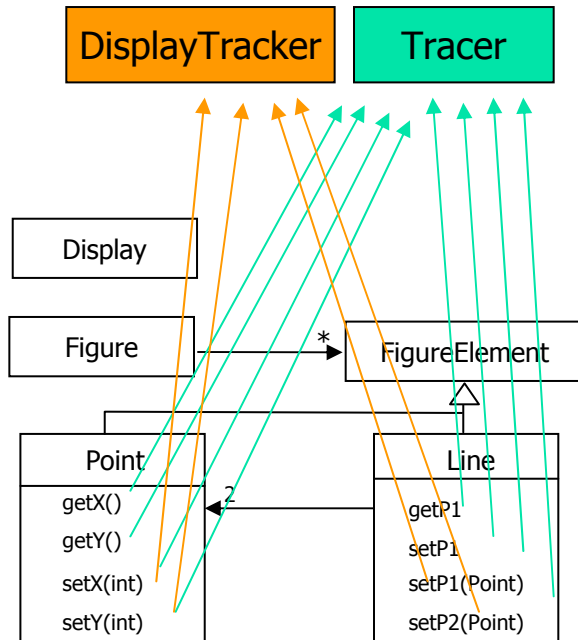
Example – Tracing and Displaytracking

- Enhance your design to trace the execution of all operations
- and update display when moving figure elements...

Example – Tracing and Display Tracking



Example: Tracing and Display Tracking



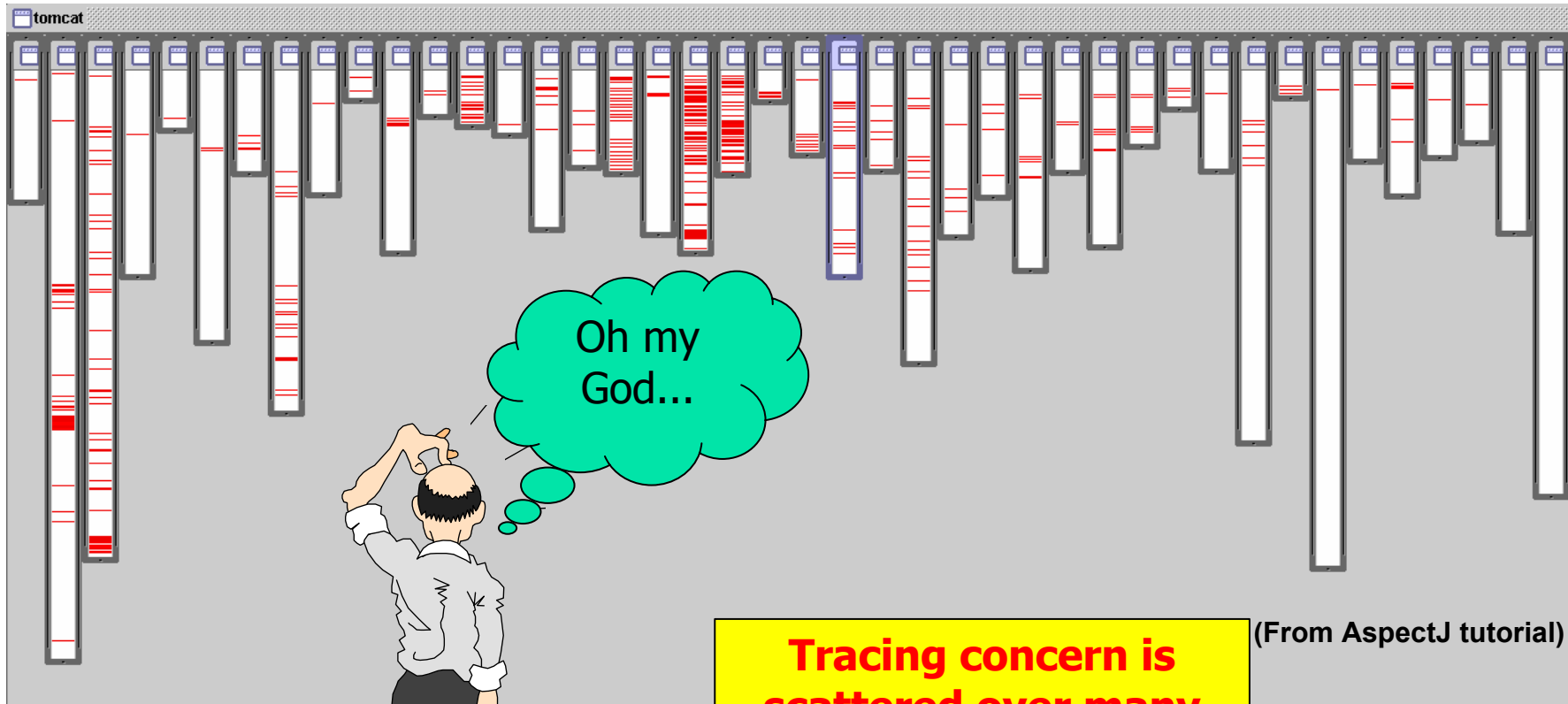
```

class Point {
    void setX(int x) {
        DisplayTracker.updatePoint(this);
        Tracer.traceEntry("Entry Point.set");
        _x = x;
        Tracer.traceExit("Exit Point.set");
    }
}
  
```

**High
coupling**

**Low
Cohesion**

Aspects Visualized - Tracing



Tracing concern is scattered over many modules. In every module tangled code.



Crosscutting, Scattering and Tangling

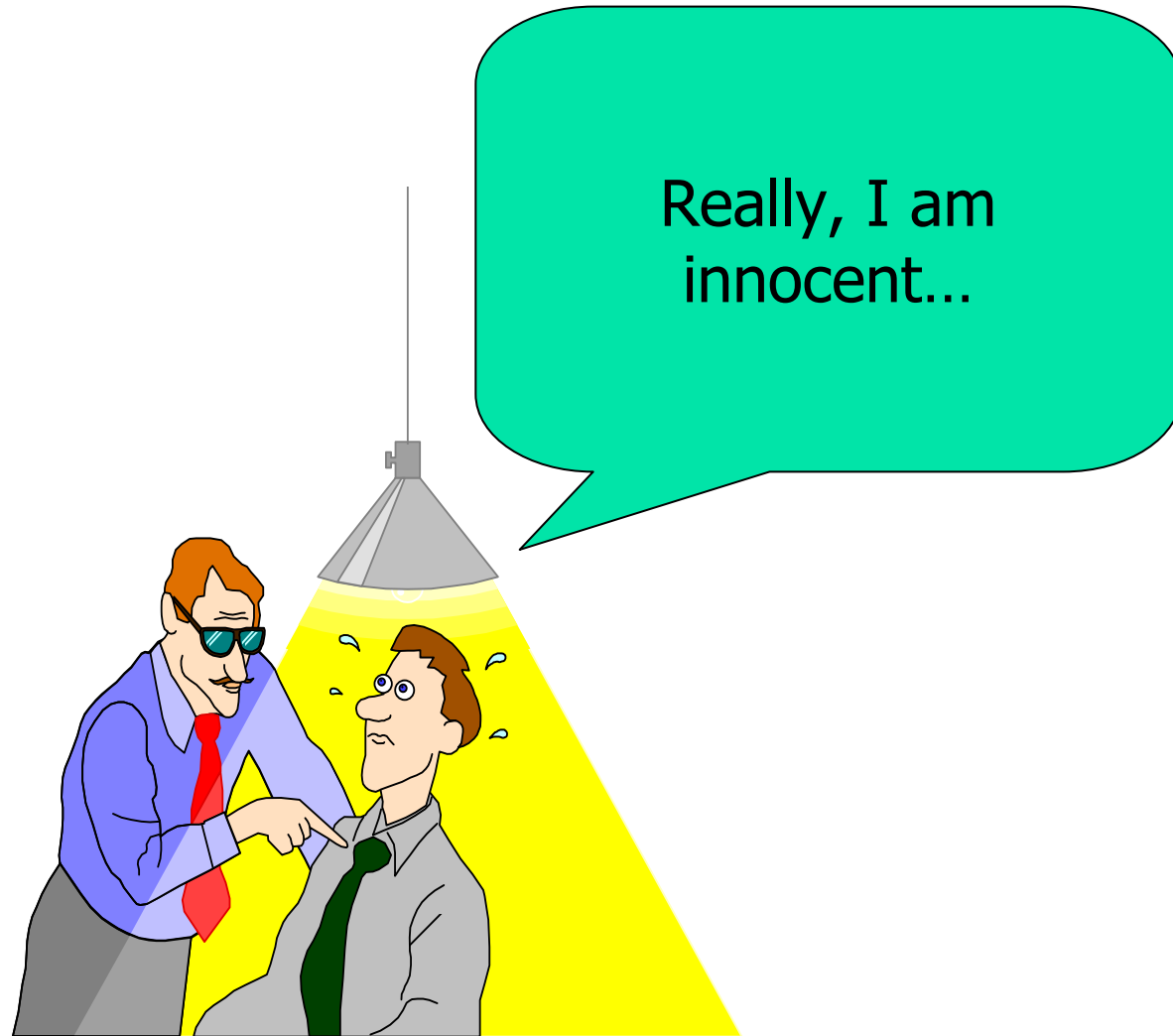
- Crosscutting
 - concern that *inherently* relates to multiple components.
 - results in scattered concern and tangled code
- Scattering
 - Single concern affects multiple modules
- Tangling
 - multiple concerns are interleaved in a single module

The Cost of Crosscutting Concerns

- Reduced understandability
 - Redundant code in many places
 - non-explicit structure
- Decreased adaptability
 - have to find all the code involved
 - and be sure to change it consistently
 - and be sure not to break it by accident
 - New concerns cannot be easily added
- Decreased reusability
 - component code is tangled with specific tangling code
- Decreased maintainability
 - 'ripple effect'



The case of the innocent designer...





Example of crosscutting concerns

- Synchronization
- Real-time constraints
- Error-checking
- Object interaction constraints
- Memory management
- Persistency
- Security
- Caching
- Logging
- Monitoring
- Testing
- Domain specific optimization
- ...

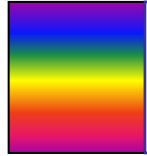


Many crosscutting concerns may appear in one system

Example:

Distributed System Design

- Component interaction
- Synchronization
- Remote invocation
- load balancing
- replication
- failure handling
- quality of service
- distributed transactions



Historical Context

- Crosscutting concerns are new type of concerns that have not been (appropriately) detected/handled before.
- No explicit management until recently at programming level
- No explicit consideration in design methods
- No explicit consideration in process
- No explicit consideration in case tools

BUT:

- Aspects are present, and severely reduce the quality of software if not appropriately managed.

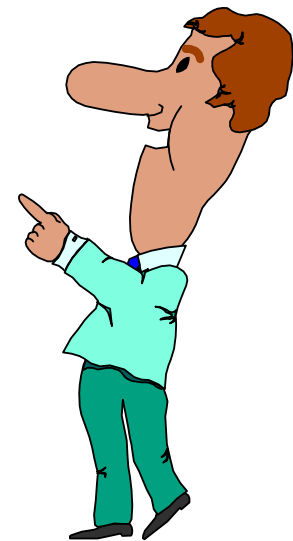


Aspect-Oriented Software Development

- Provides better separation of concerns by explicitly considering crosscutting concerns (as well)
- Does this by providing explicit abstractions for **representing** crosscutting concerns, i.e. **aspects**
- and **composing** these into programs, i.e. **aspect weaving** or **aspect composing**.
- As such AOSD **improves modularity**
- and supports quality factors such as
 - maintainability
 - adaptability
 - reusability
 - understandability
 - ...

Basic AOP technologies

- Composition Filters (since 1991)
 - University of Twente, The Netherlands
- AspectJ (since 1997)
 - XEROX PARC, US
- DemeterJ/DJ (1993)
 - Northeastern University, US
- Multi-dimensional separation of Concerns/HyperJ (1999)
 - IBM TJ Watson Research Center, USA



Example – Without AOP

```

class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        Tracer.traceEntry("entry setP1");
        _p1 = p1;
        Tracer.traceExit("exit setP1");
    }

    void setP2(Point p2) {
        Tracer.traceEntry("entry setP2");
        _p2 = p2;
        Tracer.traceExit("exit setP2");
    }
}

class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        Tracer.traceEntry("entry setX");
        _x = x;
        Tracer.traceExit("exit setX");
    }

    void setY(int y) {
        Tracer.traceEntry("entry setY");
        _y = y;
        Tracer.traceExit("exit setY");
    }
}

```

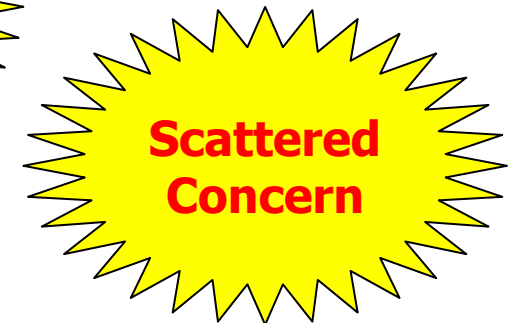
```

class Tracer {

    static void traceEntry(String str)
    {
        System.out.println(str);
    }

    static void traceExit(String str)
    {
        System.out.println(str);
    }
}

```





Example – With AOP

```

class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        _p1 = p1;
    }
    void setP2(Point p2) {
        _p2 = p2;
    }
}

class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        _x = x;
    }
    void setY(int y) {
        _y = y;
    }
}

```

```

aspect Tracing {

    pointcut traced():
        call(* Line.* ||
             call(* Point.*));

    before(): traced() {
        println("Entering:" +
               thisjoinpoint);

    void println(String str)
        {<write to appropriate stream>}

    }
}

```

Aspect is defined in a separate module
Crosscutting is localized
No scattering; No tangling
Improved modularity

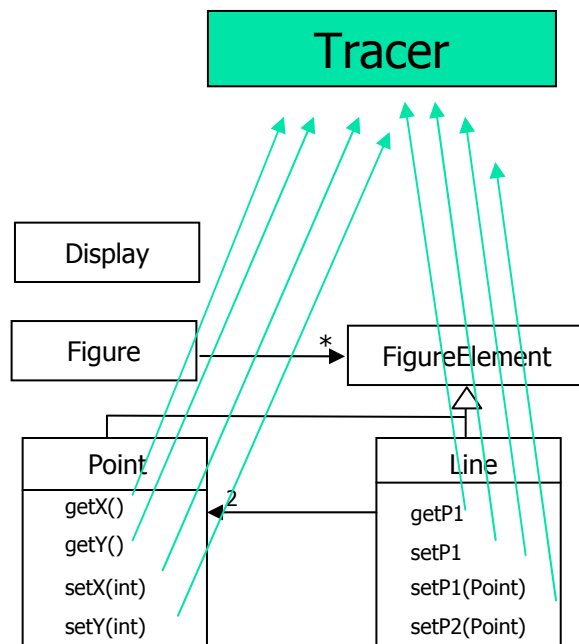


Aspect Language Elements

- join point (JP) model
 - the possible “wheres” (crosscutting points)
 - certain principled points in program execution such as method calls, field accesses, and object construction
- means of identifying JPs
 - picking out join points of interest (predicate)
 - ***pointcut*** designators
- means of specifying behavior at JPs
 - what happens
 - ***advice*** declarations

Modularizing Crosscutting

- Joinpoints: any well-defined point of execution in a program such as method calls, field accesses, and object construction
- Pointcut: predicate on joinpoints selecting a collection of joinpoints.



```
pointcut traced():
    call(* Line.*) ||
    call(* Point.);
```

**Crosscutting is
Modularized**



Example - AspectJ

```

class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        _p1 = p1;
    }
    void setP2(Point p2) {
        _p2 = p2;
    }
}

class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        _x = x;
    }
    void setY(int y) {
        _y = y;
    }
}

```

```

aspect Tracing {

```

```

    pointcut traced():
        call(* Line.* ||
            call(* Point.*));

```

```

    before(): traced() {
        println("Entering:" +
            thisjoinpoint);

```

```

    after(): traced() {
        println("Exit:" +
            thisjoinpoint);

```

```

    void println(String str)
        {<write to appropriate stream>}
    }
}

```

aspect

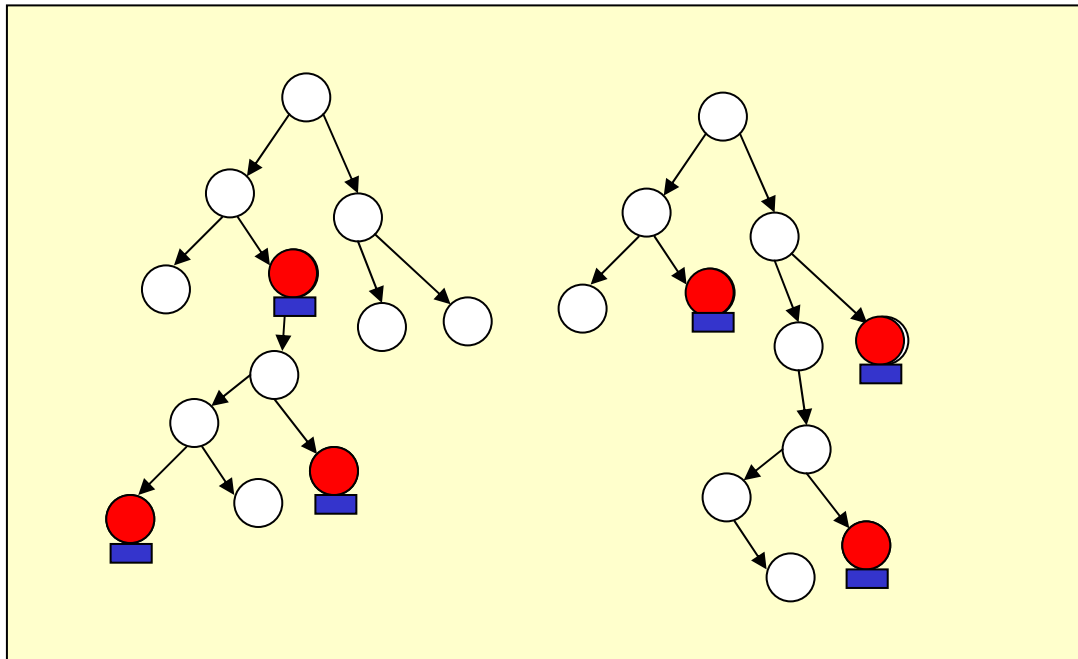
pointcut

advice

Pointcuts, Joinpoints and Advice

1. Select Joinpoints

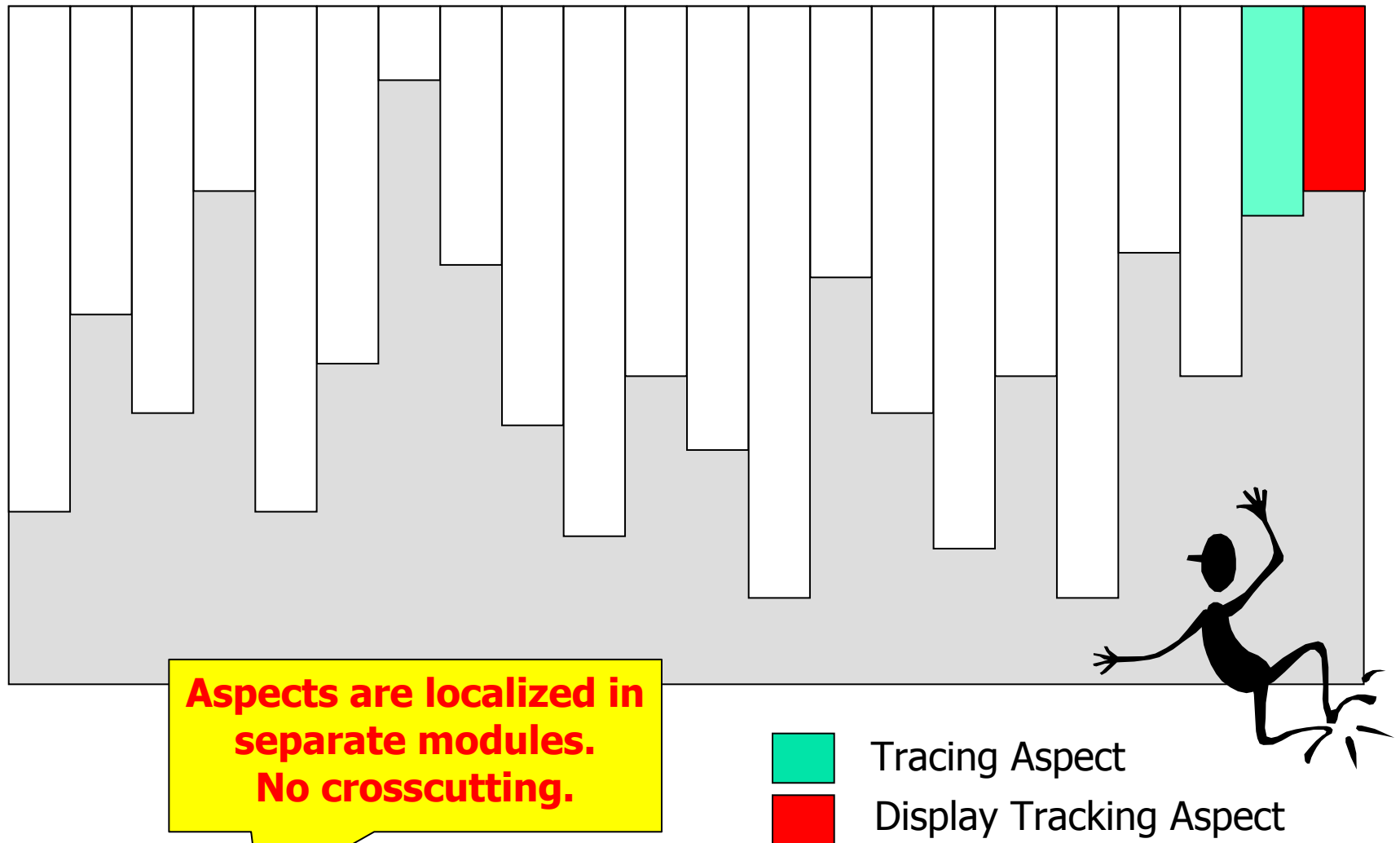
2. Inject advice code



- selected joinpoints
- injected (advice) code



Aspects Visualized





Conclusion

- For designing quality software several design principles must be followed.
- The key design principles are abstraction, decomposition, modularity, encapsulation, information hiding and separation of concerns.
- While applying these principles some designs might still not be modular using conventional abstraction mechanisms.
- This might be the case in which we are dealing with crosscutting concerns.



Conclusions

- Crosscutting concerns are typically scattered over several modules and result in tangled code.
- This reduces the modularity and as such the quality of the software system.
- AOSD provides explicit abstractions mechanisms to represent these so-called *aspects* and compose these into programs.
- increases the modularity of systems.