Software Architecture and Engineering

Modularity

Peter Müller

Chair of Programming Methodology

Spring Semester 2017

ETH zürich

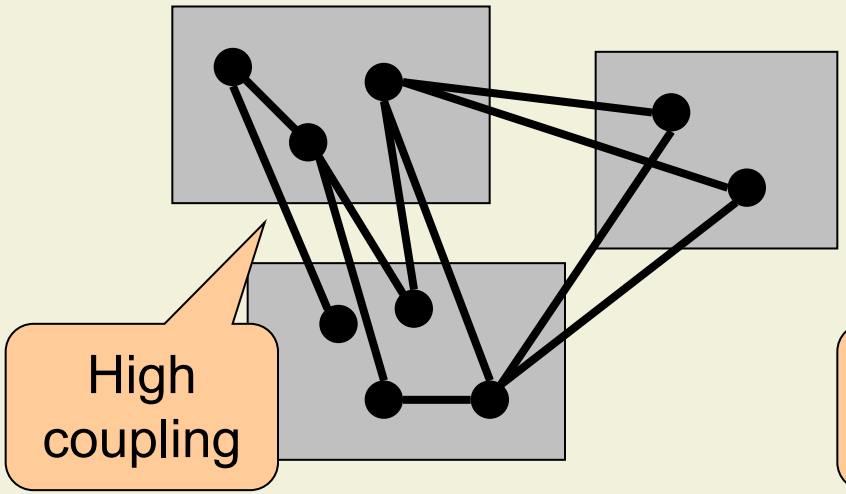
Mastering Complexity

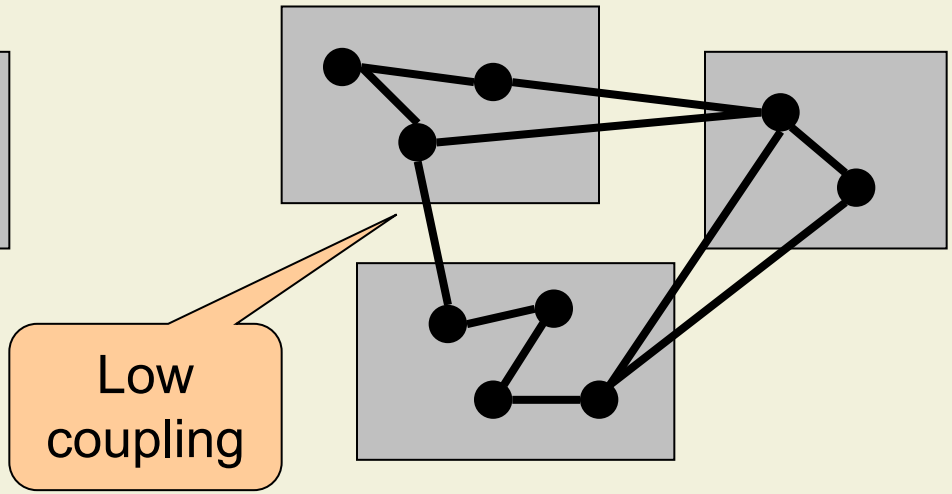
- *The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule).* [Dijkstra, 1965]

- Benefits of decomposition
  - Partition the overall development effort
  - Support independent testing and analysis
  - Decouple parts of a system so that changes to one part do not affect other parts
  - Permit system to be understood as a composition of mind-sized chunks with one issue at a time
  - Enable reuse of components

Coupling

- Coupling measures interdependence between different modules





- Tightly-coupled modules cannot be developed, tested, changed, understood, or reused in isolation

4. Modularity

4.1 Coupling
4.1.1 Data Coupling
4.1.2 Procedural Coupling
4.1.3 Class Coupling
4.2 Adaptation

Representation Exposure

- Modules that expose their internal data representation become tightly coupled to their clients

```
class Coordinate {
  public double radius, angle;

  public double getX( ) {
    return Math.cos( angle ) * radius;
  }
}
```

```
class Item {
  private int id;
  protected static int nextId;

  public Item( … ) {
    id = Item.nextId;
    Item.nextId++; … }
}
```

Problems of Representation Exposure

- Data representation is difficult to change during maintenance
- Modules cannot maintain strong invariants
- Concurrency requires complex synchronization

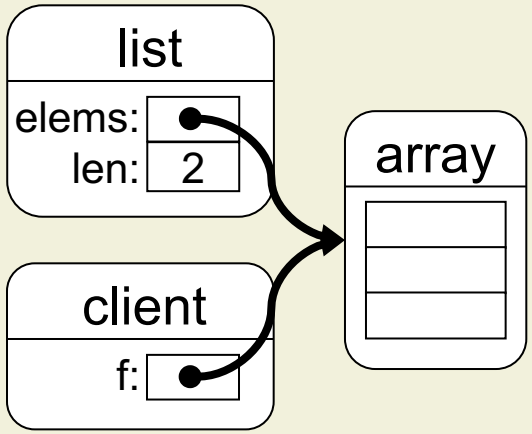
```
class Coordinate {
  public double radius, angle;
  invariant 0 <= radius;

  public double getX( ) {
    synchronized( this ) {
      return Math.cos( angle ) * radius;
    }
  }
}
```

```
class Coordinate {
  public double x,y;

  public double getX( ) { return x; }
}
```

Representation Exposure (cont'd)

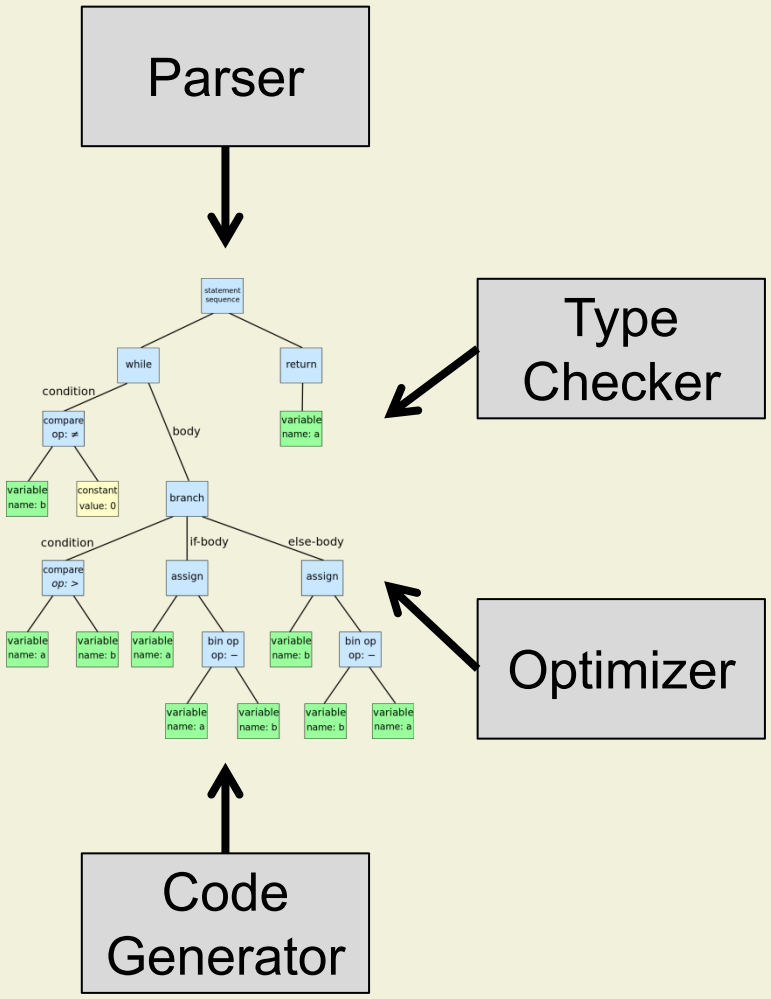
- The data representation often includes sub-objects or entire sub-object structures
- In addition to the problems above, exposing sub-objects may lead to unexpected side effects

```
class List<E> {
  E[ ] elems;
  int len;

  void set( int index, E e )
  { elems[ index ] = e; }

  E[ ] toArray( )
  { return elems; }
}
```



Shared Data Structures

- Modules get coupled by operating on shared data structures
  - Including databases and files
- Problems caused by
  - Changes in data structure
  - Unexpected side effects
  - Concurrency



Approach 1: Restricting Access to Data

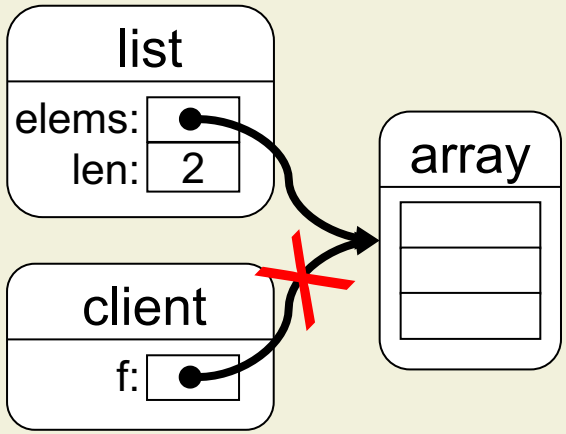
- Force clients to access the data representation through a narrow interface
- Information hiding: Hide implementation details behind interface
- Use interface for necessary checks

```
class Coordinate {
  private double radius, angle;
  invariant 0 <= radius;

  public void setRadius( double r )
    requires 0 <= r;
  { synchronized( this ) { radius = r; } }

  public double getX( ) {
    synchronized( this ) {
      return Math.cos( angle ) * radius;
    }
  }
}
```

Restricting Access to Data (cont'd)

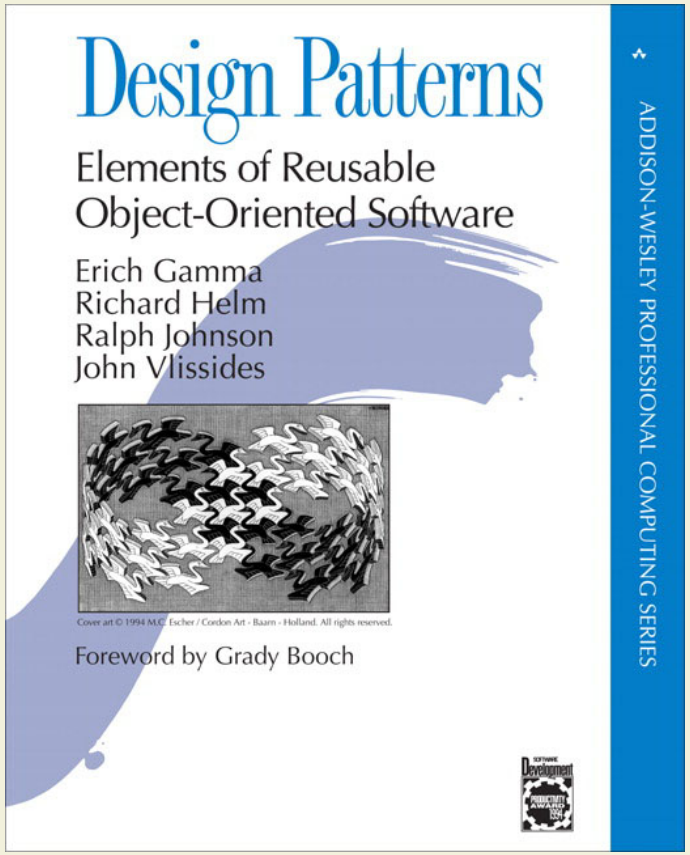
- Avoid exposure of sub-objects
- No leaking: Do not return references to sub-objects
- No capturing: Do not store arguments as sub-objects
- Clone objects if necessary

```
class List<E> {
  E[ ] elems;
  int len;

  E[ ] toArray( ) {
    E[ ] res = new E[ len ];
    System.arraycopy( … );
    return res;
  }
}
```



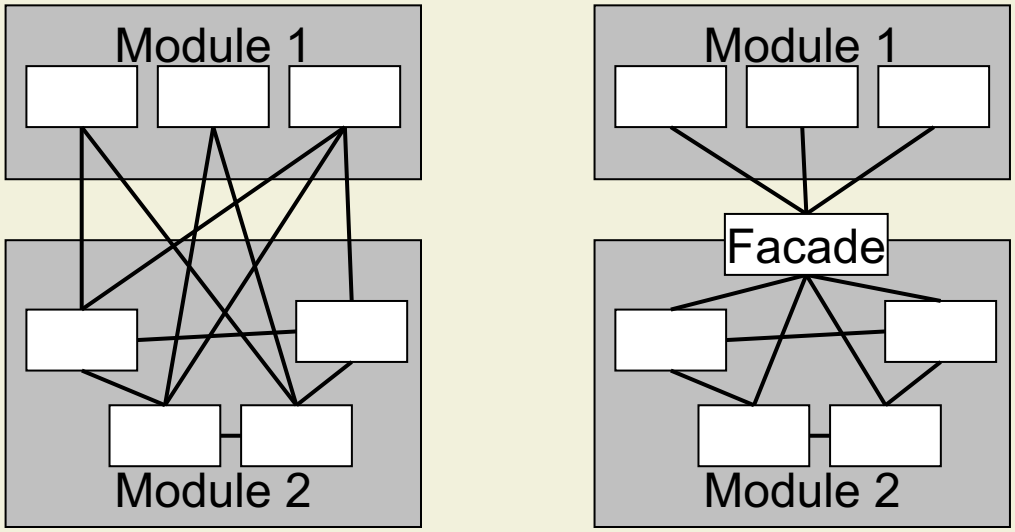
Design Patterns

- Design patterns are general, reusable solutions to commonly occurring design problems
- They capture best practices in detailed design
- They describe relationships and interactions among classes and objects



Facade Pattern

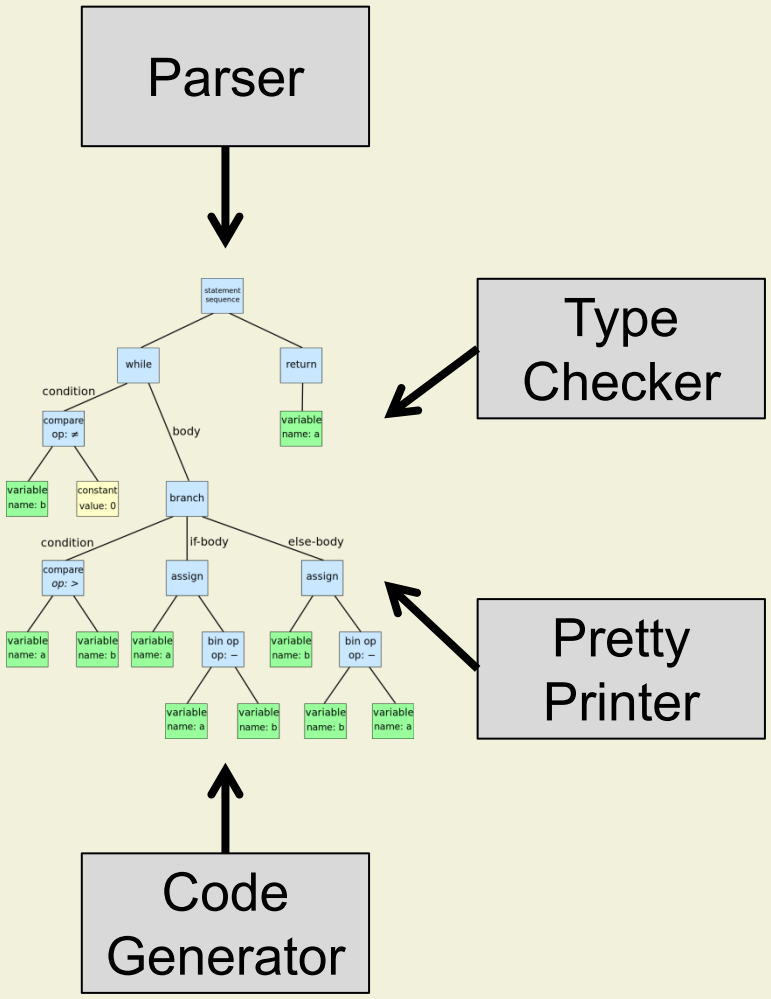
- Facade objects provide a single, simplified interface to the more general facilities of a module without hiding the details completely



- Example: Access scanner, parser, AST node, etc. through a compiler object

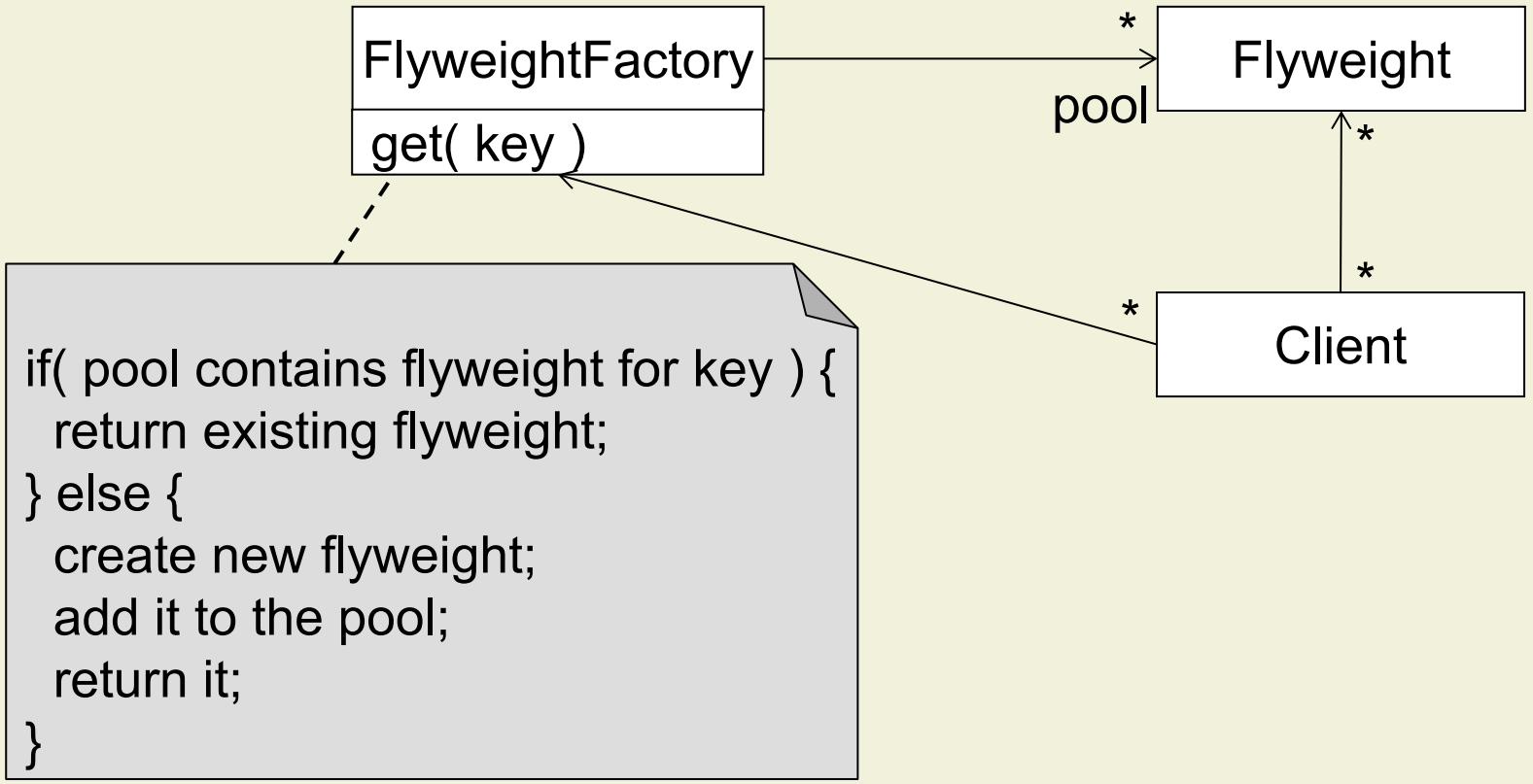
Approach 2: Making Shared Data Immutable

- Some drawbacks of shared data apply only to mutable shared data
  - Maintaining invariants
  - Thread synchronization
  - Unexpected side effects
- Changing the data representation remains a problem
- Copies can lead to run-time and memory overhead



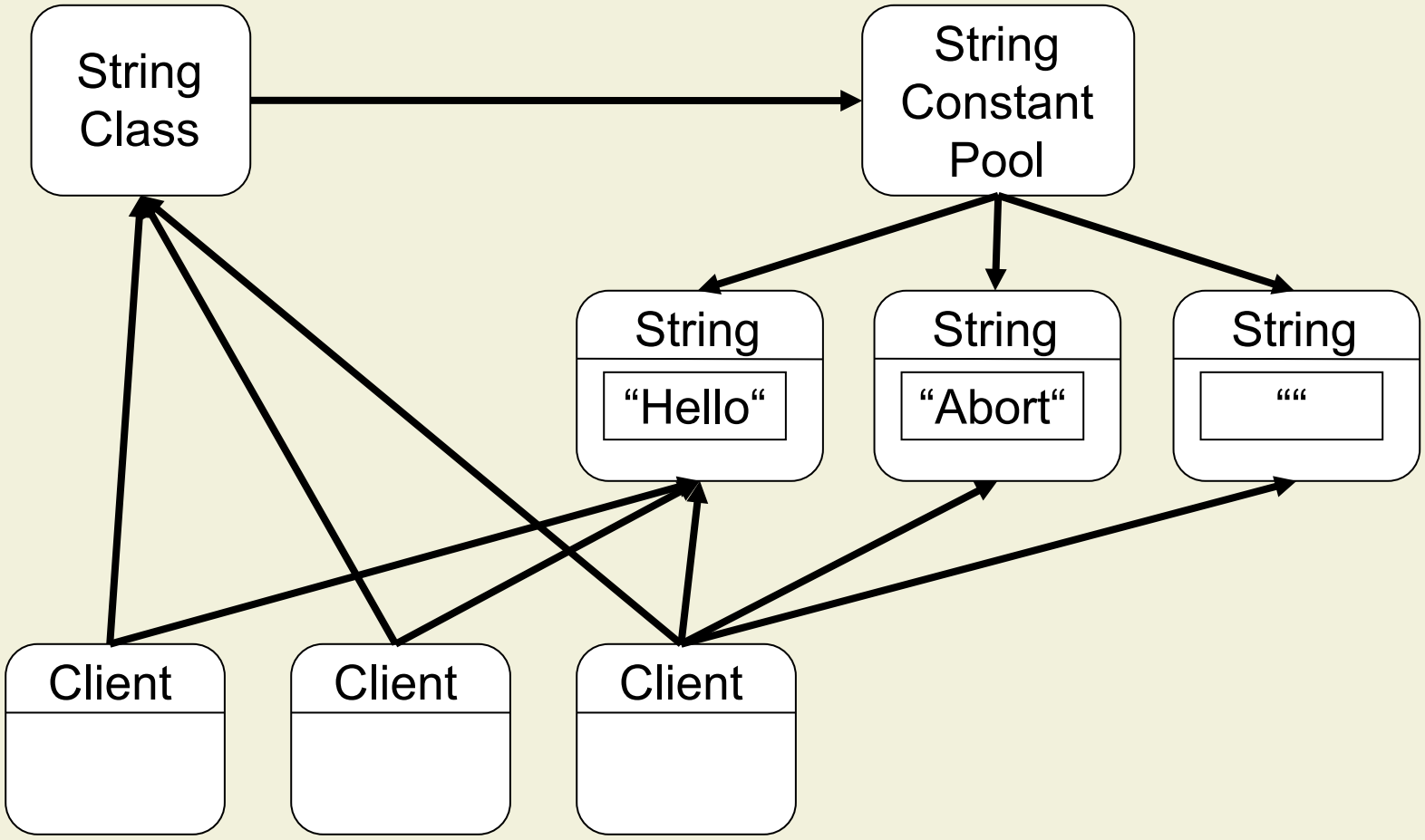
Flyweight Pattern

- The flyweight pattern maximizes sharing of immutable objects



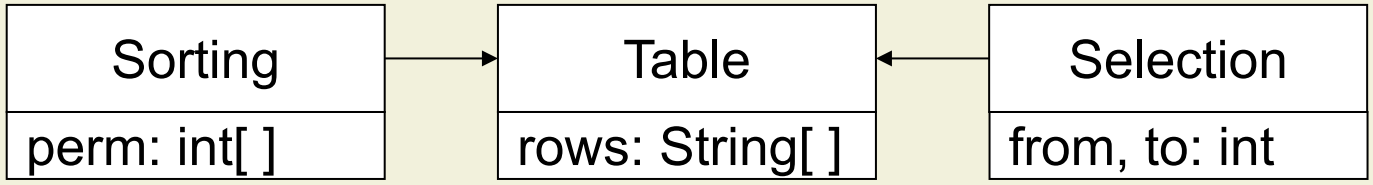
Flyweight Pattern: Example

- Java uses the flyweight pattern for constant strings



Approach 3: Avoiding Shared Data

- Working with immutable shared data is often cumbersome



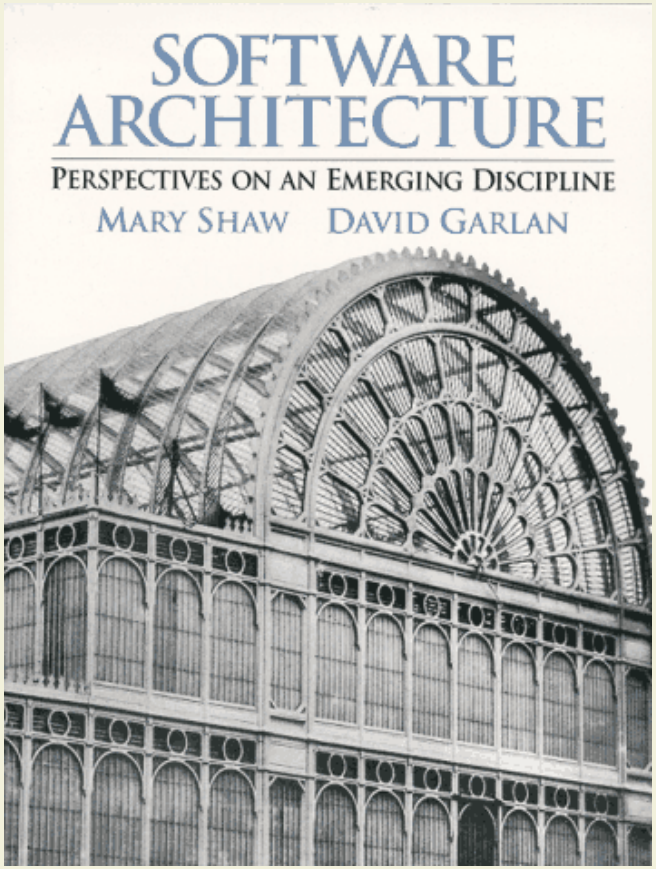
- Sorting a selection is not supported

Sorting	Table	Selection
sort(String[ ]): String[ ]	rows: String[ ]	select(String[ ], int, int): String[ ]

- Sorting and Selection can be combined flexibly

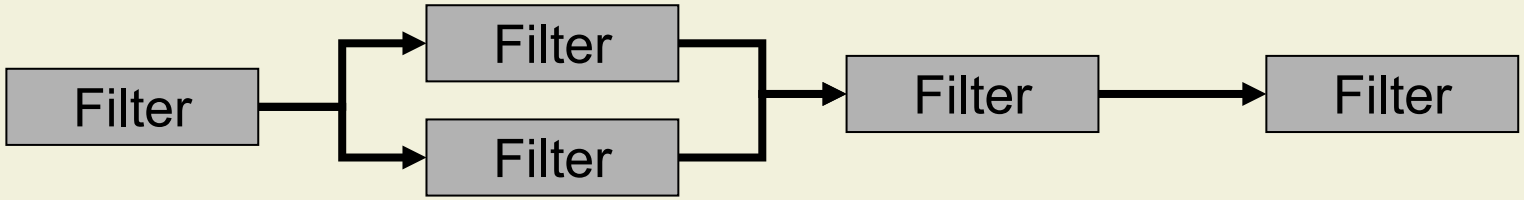
Architectural Styles

- Architectural styles are idiomatic patterns of system organization
- They capture best practices in system design
- They describe the components and connectors of a software architecture



Pipe-and-Filter Style

- Data flow is the only form of communication between components
  - No shared state
- Components (Filters)
  - Read data from input ports, compute, write data to output ports
- Connectors (Pipes)
  - Streams (typically asynchronous first-in-first-out buffers)
  - Split-join connectors



Pipe-and-Filter Style: Properties

- Data is processed incrementally as it arrives
- Output usually begins before input is consumed
- Filters must be independent, no shared state
- Filters don't know upstream or downstream filters

- Examples
  - Unix pipes

grep search-text file | **sort**

sort file | **grep** search-text

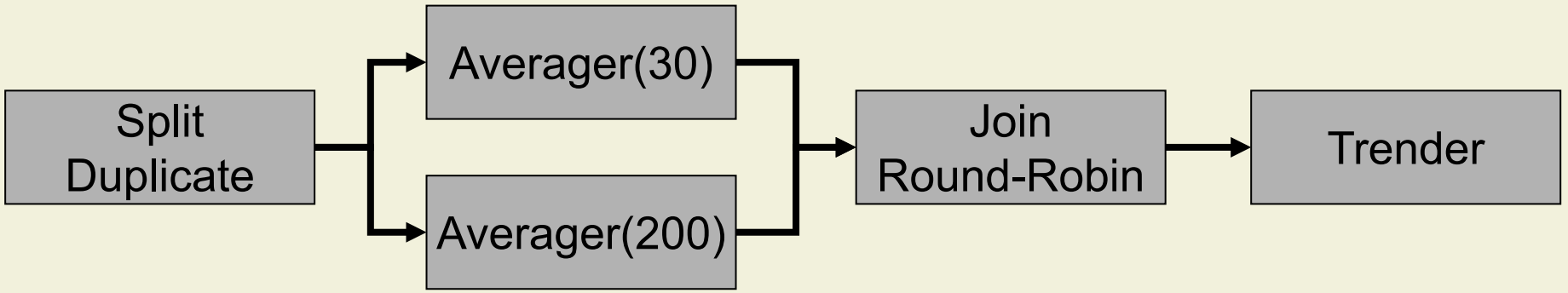
  - Stream processing

Pipe-and-Filter Style: Example

- For a stream of stock quotes, compute the 30-days and 200-days simple moving average (SMA) and determine trend

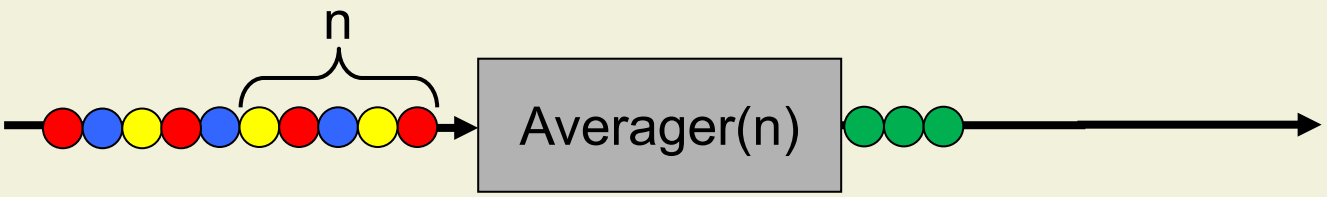
Pipe-and-Filter Style: Example (cont'd)

- Architecture



- We sketch an implementation in MIT's StreamIt language
  - http://groups.csail.mit.edu/cag/streamit/

StreamIt Language: Filters



Input and output streams are typed

Parameter is provided when filter is instantiated

```
int->int filter Averager( int n ) {
  work pop 1 push 1 peek n {
    int sum = 0;
    for ( int i = 0; i < n; i++ )
      sum += peek( i );
    push( sum/n );
    pop( );
  }
}
```

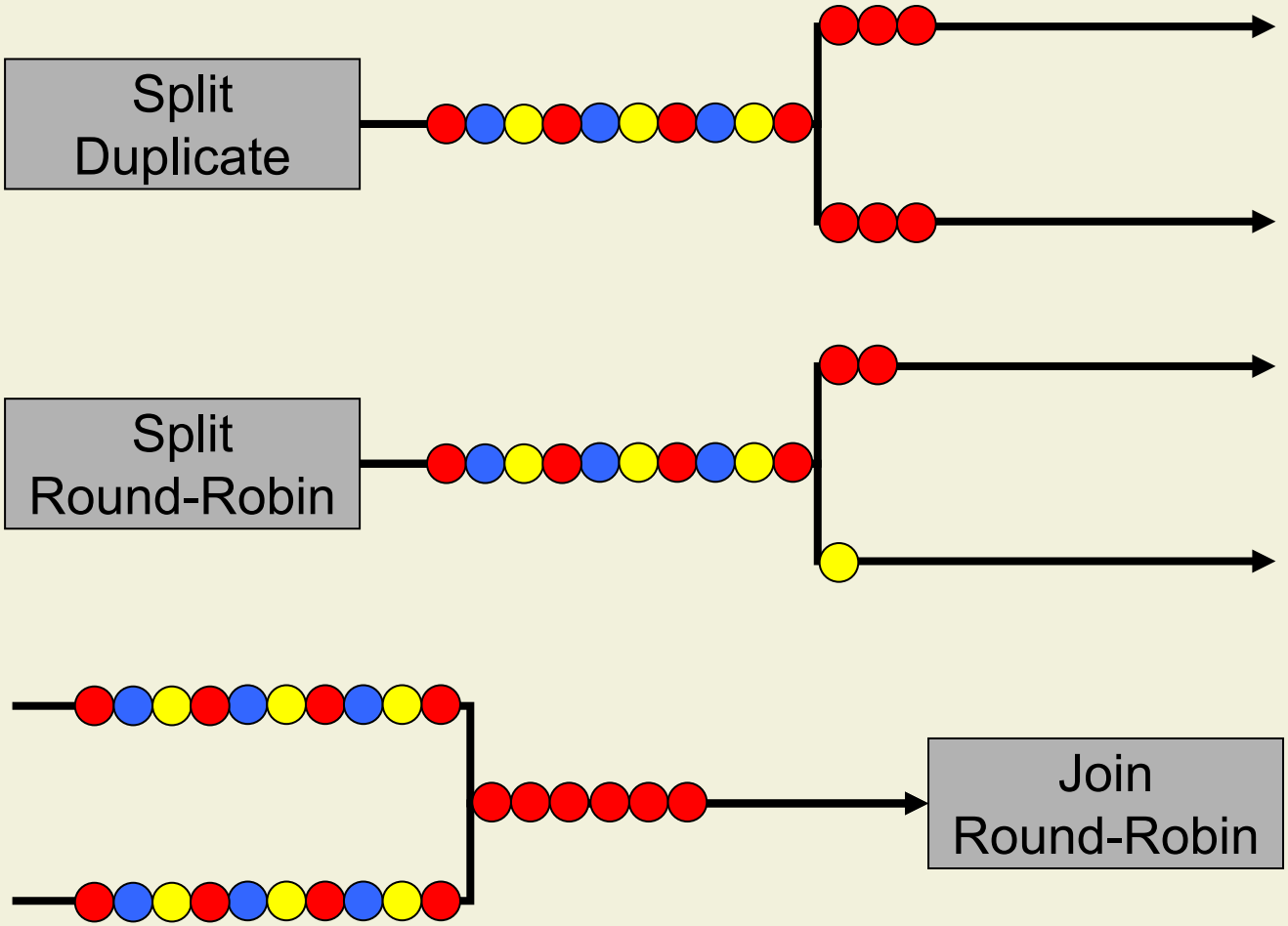
work function is executed repeatedly

Each work function declares its data rate

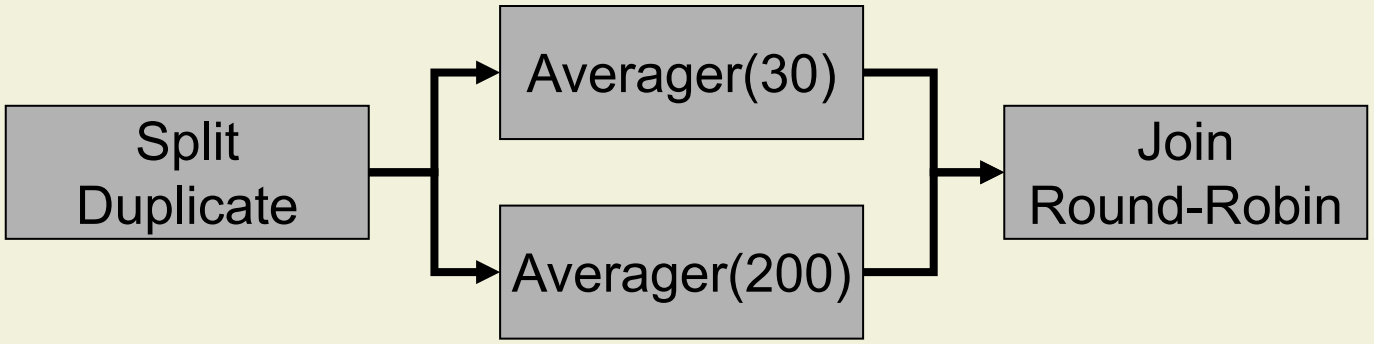
Filters may have local state

Filters may look ahead

StreamIt Language: SplitJoins

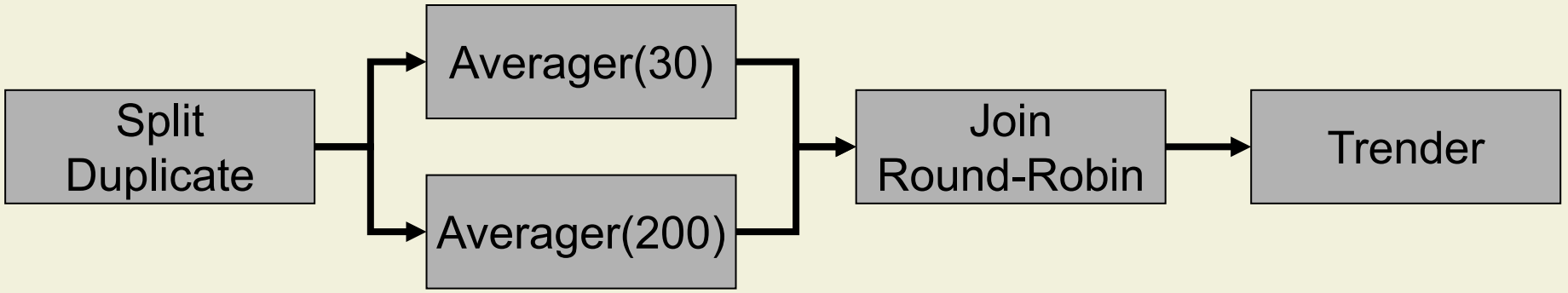


StreamIt Language: SplitJoin Example




```
int -> int splitjoin
dualAverager( int n, int m ) {
  split duplicate;
  add Averager( n );
  add Averager( m );
  join roundrobin;
}
```

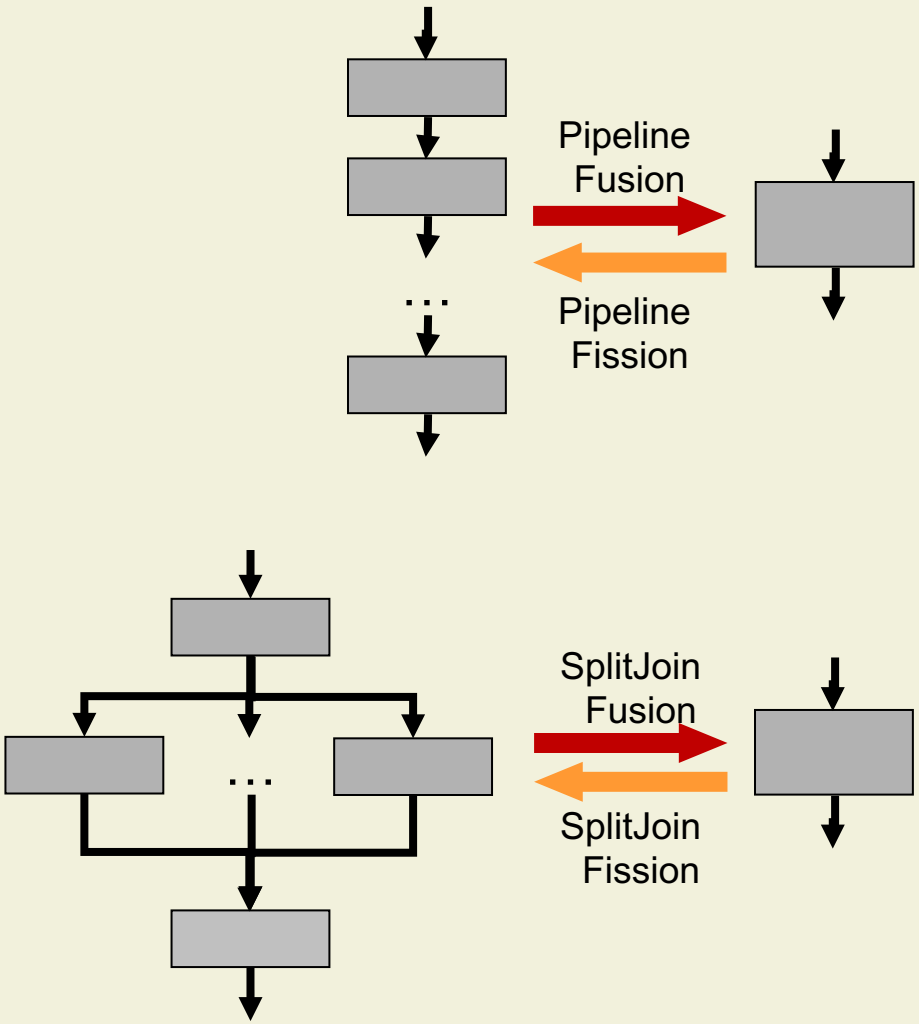
StreamIt Language: Composition Example




```
int -> int filter Trender {
  work pop 2 push 1 {
    int a = pop( );
    int b = pop( );
    if(a > b) { push( 1 ); }
    else { push( 0 ); }
  }
}
```

```
int -> int pipeline System {
  add dualAverager( 30, 200 );
  add Trender;
}
```

Fusion and Fission



- Fusion reduces communication cost at the expense of parallelism
- Fission is profitable if the benefits of parallelization outweigh the overhead introduced by fission

Pipe-and-Filter Style: Discussion

Strengths

- Reuse: any two filters can be connected if they agree on the data format that is transmitted
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

Weaknesses

- Sharing global data is expensive or limiting
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling is Achilles heel, e.g., some intermediate filter crashes
- Often smallest common denominator on data transmission, e.g., ASCII in Unix pipes

4. Modularity

4.1 Coupling
4.1.1 Data Coupling
4.1.2 Procedural Coupling
4.1.3 Class Coupling
4.2 Adaptation

Problems of Procedural Coupling: Reuse

- Modules are coupled to other modules whose methods they call
- Callers cannot be reused without callee modules

```
class Controller {
  Sensor sensor;

  public boolean selfTest( ) {
    List<LogEntry> log = sensor.log( );
    for( LogEntry e: log )
      if( e.isError( ) ) return false;
    return true;
  }
}
```

```
class LogEntry {
  …

  boolean isError( ) { … }
}
```

```
class Sensor {
  List<LogEntry> logData;

  List<LogEntry> log( ) { return logData; }
}
```

Problems of Procedural Coupling: Adaptation

- When modules are procedurally coupled, any change in the callees may require changes in the caller
  - Change in signatures
  - Adding or removing callees
- Example: Display stack trace when breakpoint is reached

```
class Editor {
  void showContext( … ) { … }
}
```

```
class Debugger {
  Editor editor;
  …
  void processBreakPoint( … ) {
    …
    editor.showContext( … );
  }
}
```

```
class StackViewer {
  void showStackTrace( … ) { … }
}
```

Approach 1: Moving Code

Loop does not use data from Controller

```
class Controller {
  Sensor sensor;

  boolean selfTest( ) {
    List<LogEntry> log = sensor.log( );
    for( LogEntry e: log ) {
      if( e.isError( ) ) return false;
    return true;
  }
}
```

```
class LogEntry {
  …

  boolean isError( ) { … }
}
```

```
class Sensor {
  List<LogEntry> logData;

  List<LogEntry> log( ) { return logData; }
}
```

Approach 1: Moving Code (cont'd)

- Moving code may reduce procedural coupling
- It is common to even duplicate functionality to avoid dependencies on code from other projects or companies

```
class Controller {
  Sensor sensor;

  boolean selfTest( )
  { return sensor.noError( ); }
}
```

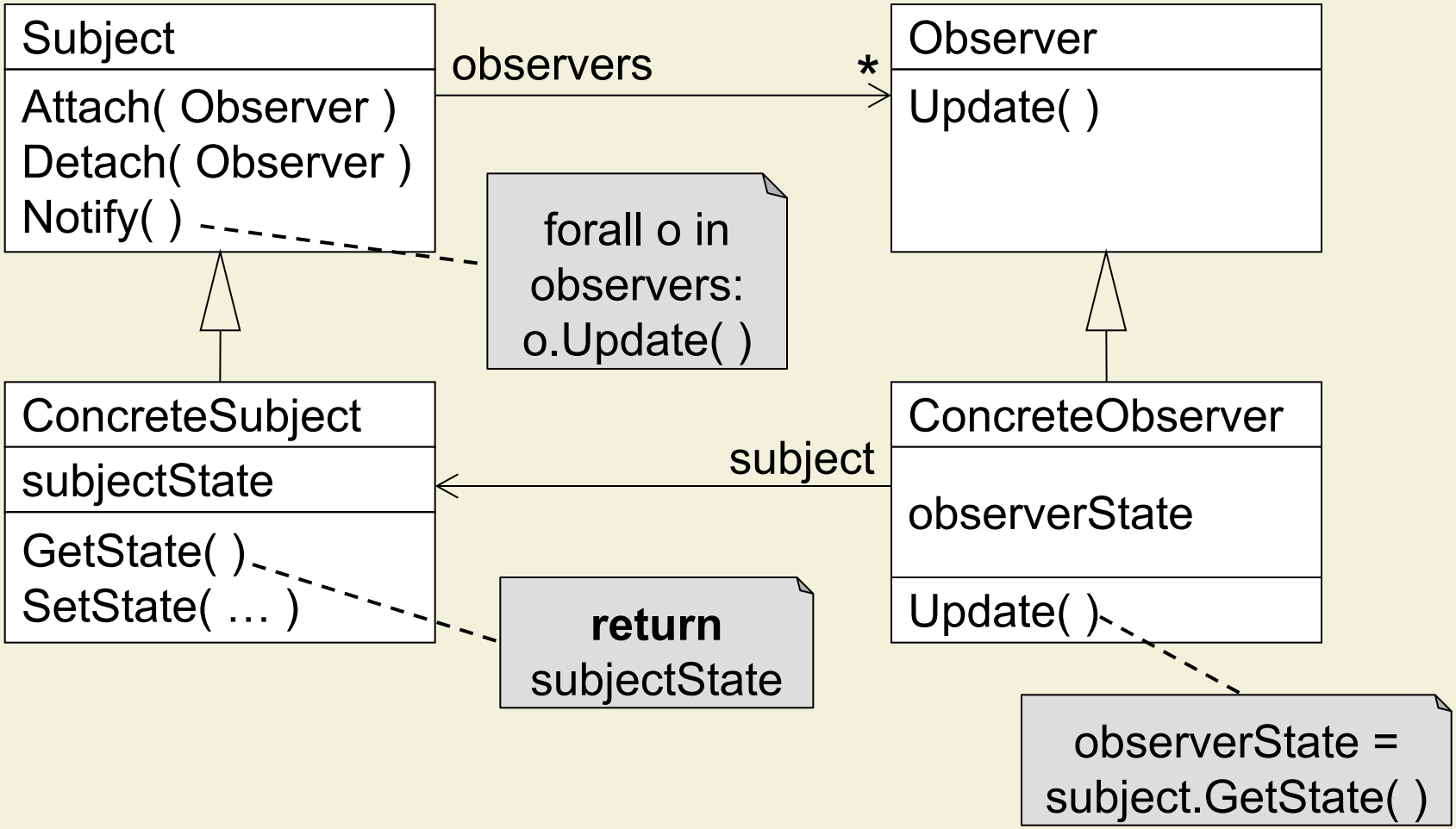
```
class Sensor {
  List<LogEntry> logData;

  boolean noError( ) {
    for( LogEntry e: logData )
      if( e.isError( ) ) return false;
    return true;
  }
}
```

```
class LogEntry {
  …

  boolean isError( ) { … }
}
```

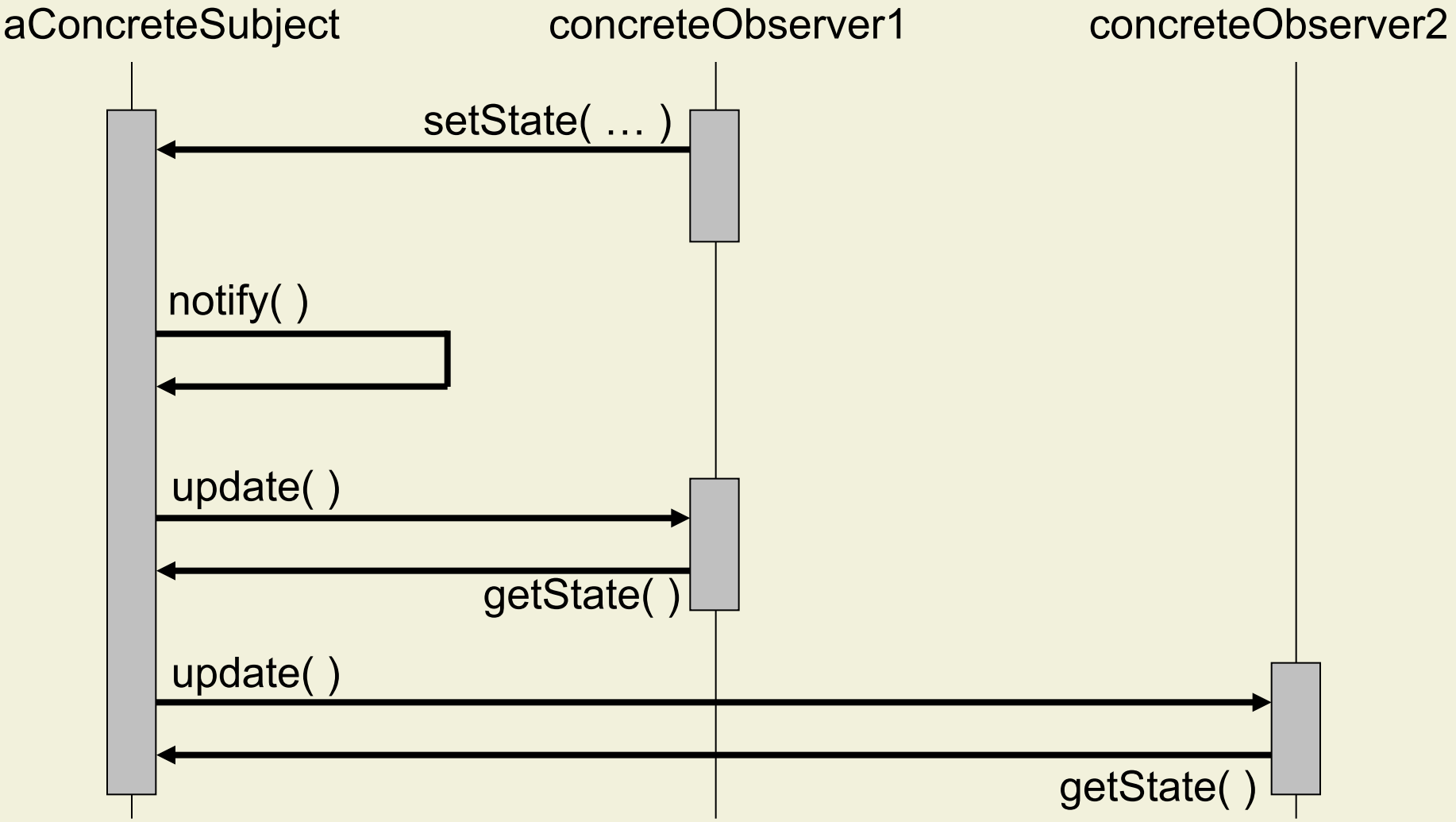
Approach 2: Event-Based Style

- Components may
  - Generate events
  - Register for events of other components with a callback
- Generators of events do not know which components will be affected by their events
- Examples
  - Programming environment tool integration
  - User interfaces, web sites, Android

Observer Pattern: Structure



Observer Pattern: Collaborations



Observer Pattern: Example

- Debugger has a generic list of observers
- Debugger generates event when breakpoint is reached
- Observers decide how to handle this event (no control by debugger)

```
class Debugger extends Subject {
  …
  void processBreakPoint( … ) {
    …
    notify( … );
  }
}
```

```
class Editor
            implements Observer {
  void showContext( … ) { … }
  void update ( … ) {
    showContext( … );
  }
}
```

Adaptation: Add StackViewer

- New requirement: Display stack trace when breakpoint is reached
- StackViewer is just another observer
- Debugger does not have to be adapted

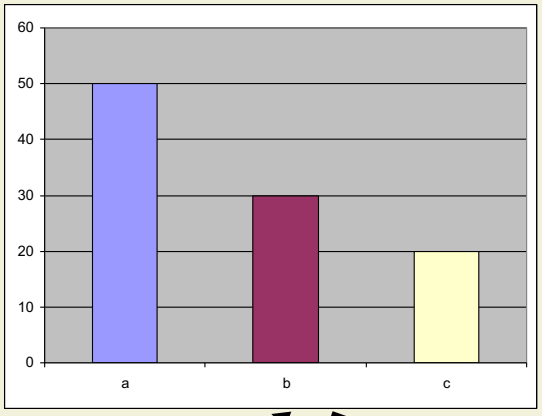
```
class StackViewer
            implements Observer {
  …
  void showStackTrace( … )
    { … }

  void update ( … ) {
    showStackTrace( … );
  }
}
```

Model-View-Controller Architecture

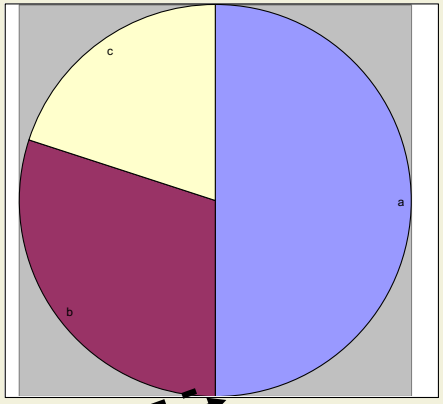
- Components
  - Model contains the core functionality and data
  - One or more views display information to the user
  - One or more controllers handle user input

- Communication
  - Change-propagation mechanism via events ensures consistency between user interface and model
  - If the user changes the model through the controller of one view, the other views will be updated automatically

Model-View-Controller Example



	a	b	c
X	60	30	10
Y	50	30	20
Z	80	10	10

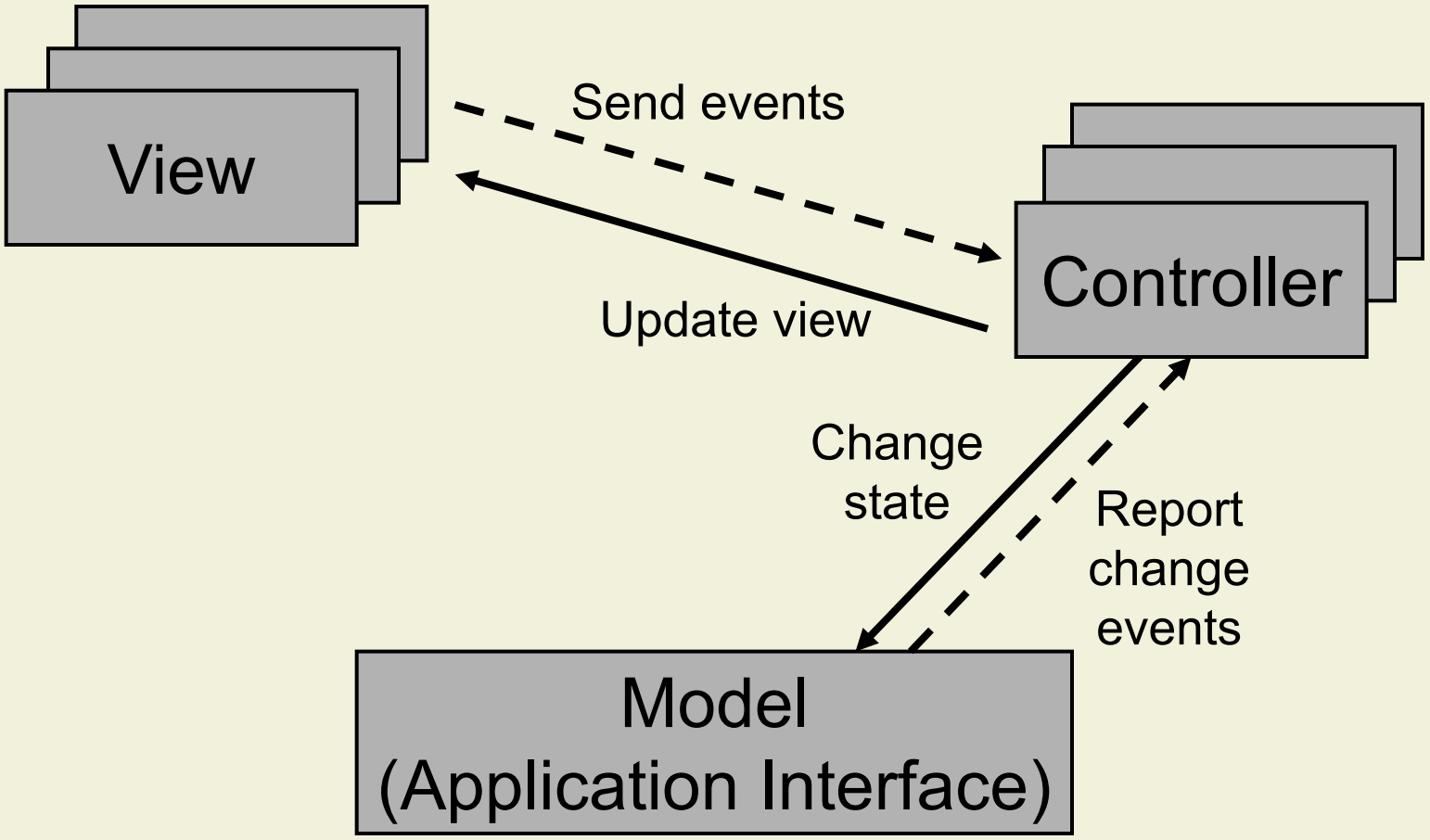


a = 50%
b = 30%
c = 20%

Change notification

Requests, modifications

Model-View-Controller Architecture



Event-Based Style: Discussion

Strengths

- Strong support for reuse: plug in new components by registering it for events
- Adaptation: add, remove, and replace components with minimum effect on other components in the system

Weaknesses

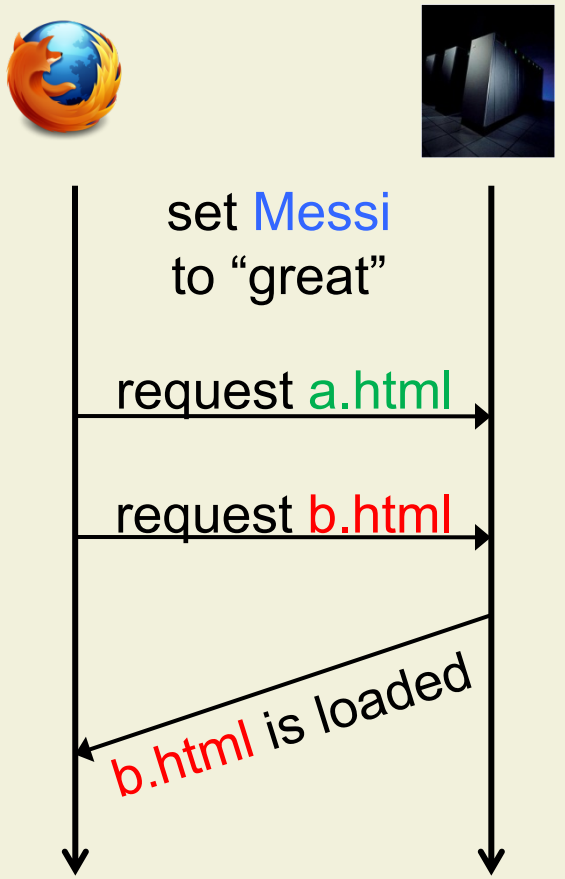
- Loss of control
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Ensuring correctness is difficult because it depends on context in which invoked

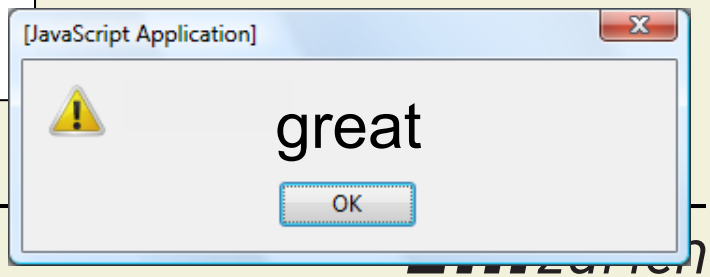
Loss of Control: Example

```
<html><body>
<script>document.Messi = “great”;</script>
<iframe src="a.html" ></iframe>
<iframe src="b.html" ></iframe>
</body></html>
```

```
<html><body>
<script>parent.document.Messi = “poor”;
    </script>
</body></html>
```

```
<html><body>
<script>alert(parent.document.Messi);</script>
</body></html>
```



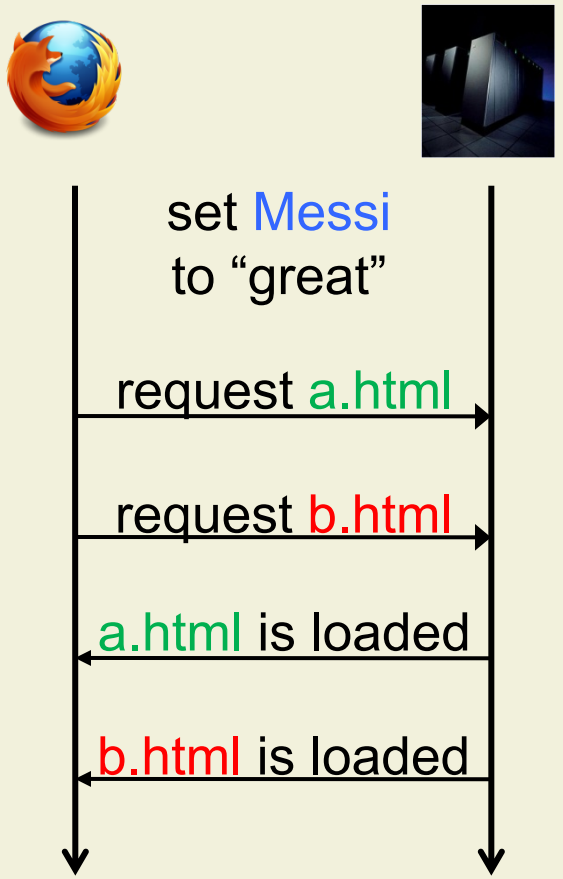


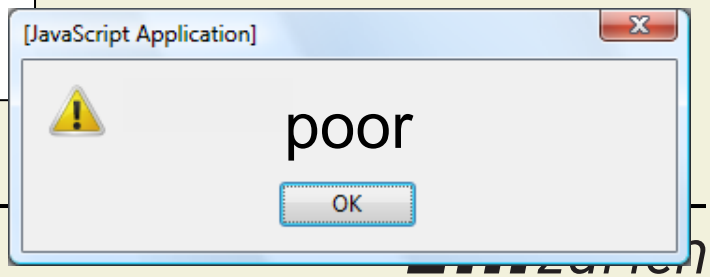
Loss of Control: Example (cont'd)

```
<html><body>
<script>document.Messi = “great”;</script>
<iframe src="a.html" ></iframe>
<iframe src="b.html" ></iframe>
</body></html>
```

```
<html><body>
<script>parent.document.Messi = “poor”;
    </script>
</body></html>
```

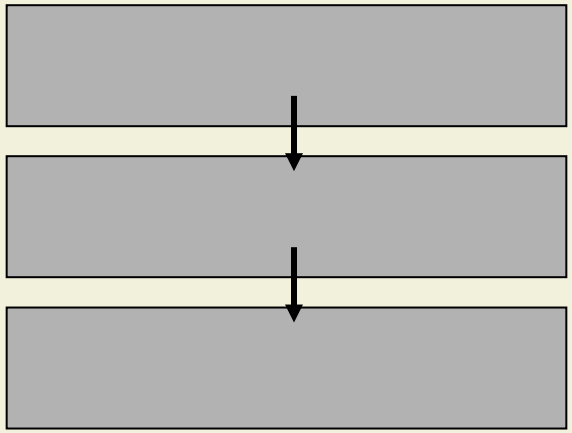
```
<html><body>
<script>alert(parent.document.Messi);</script>
</body></html>
```

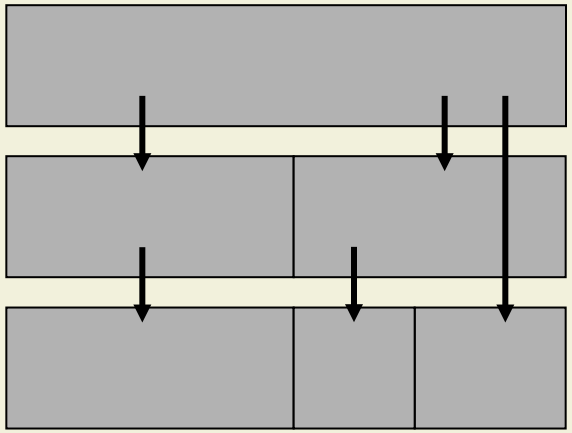




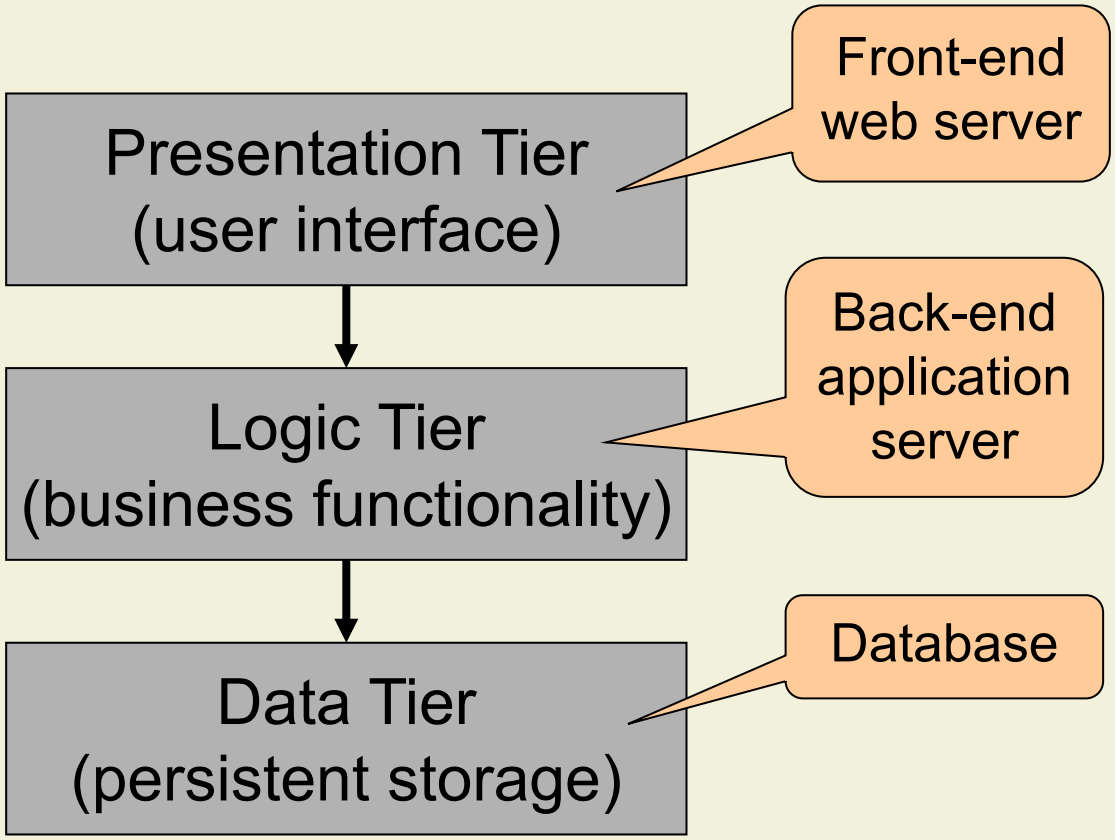
Approach 3: Restricting Calls

- Enforce a policy that restricts which other modules a module may call
- Example: Layered architectures
  - A layer depends only on lower layers
  - Has no knowledge of higher layers
  - Layers can be exchanged





Example: Three-Tier Architecture



Layered Style: Discussion

Strengths

- Increasing levels of abstraction as we move up through layers: partitions complex problems
- Maintenance: in theory, a layer only interacts with layer below (low coupling)
- Reuse: different implementations of the same level can be interchanged

Weaknesses

- Performance: communicating down through layers and back up, hence bypassing may occur for efficiency reasons

4. Modularity

4.1 Coupling
4.1.1 Data Coupling
4.1.2 Procedural Coupling
4.1.3 Class Coupling
4.2 Adaptation

Inheritance

- Inheritance couples the subclass to the superclass

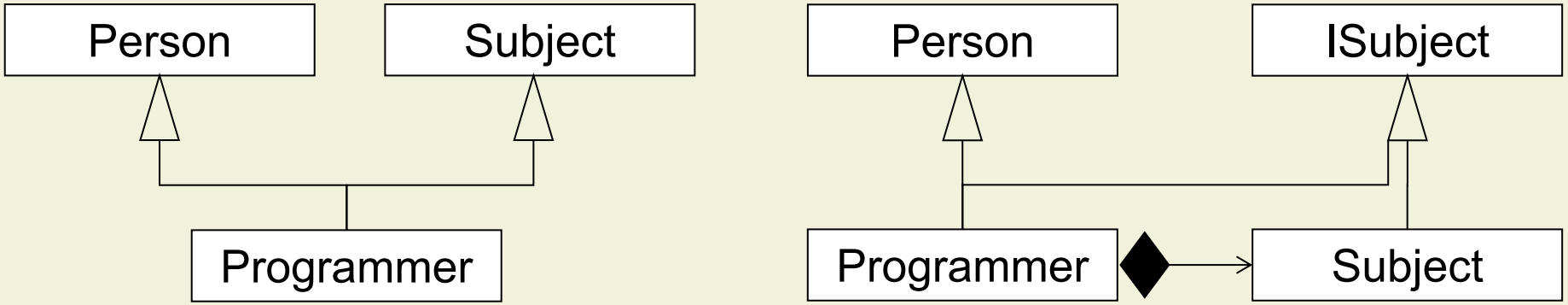
```
class SymbolTable
      extends TreeMap<Ident, Type> {
}
```

- Changes in the superclass may break the subclass
  - Fragile baseclass problem

- Limits options for other inheritance relations
  - Not possible in single-inheritance languages
  - May cause conflicts with multiple inheritance

Approach 1: Replacing Inheritance w/ Aggreg.

- Inheritance can be replaced by Subtyping, aggregation, and delegation



- The same technique can be used to avoid coupling through inheritance

```
class SymbolTable {
  TreeMap<Ident, Type> types;

  Type getType( Ident id )
  { return types.get( id ); }
}
```

Type declarations

- Using class names in declarations of methods, fields, and local variables couples the client to the used classes

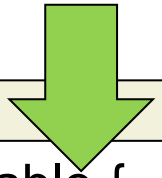
```
class SymbolTable {
  TreeMap<Ident, Type> types;

  TreeMap<Ident, Type> getTypes( ) {
    return types.clone( );
  }
}
```

- Data structures are difficult to change during maintenance

Approach 2: Using Interfaces

- Replace occurrences of class names by supertypes
- Use the most general supertype that offers all required operations
- Data structures can be changed without affecting the code

```
class SymbolTable {
  TreeMap<Ident, Type> types;

  TreeMap<Ident, Type> getTypes( ) {
    return types.clone( );
  }
}
```



```
class SymbolTable {
  Map<Ident, Type> types;

  Map<Ident, Type> getTypes( ) {
    return types.clone( );
  }
}
```

Object Allocation

- Allocations couple clients to the instantiated class

```
class SymbolTable {
  Map<Ident, Type> types;

  SymbolTable( ) {
    types = new TreeMap<Ident, Type>( );
  }
}
```

- Problem is shifted to clients
- Difficult to create objects for testing

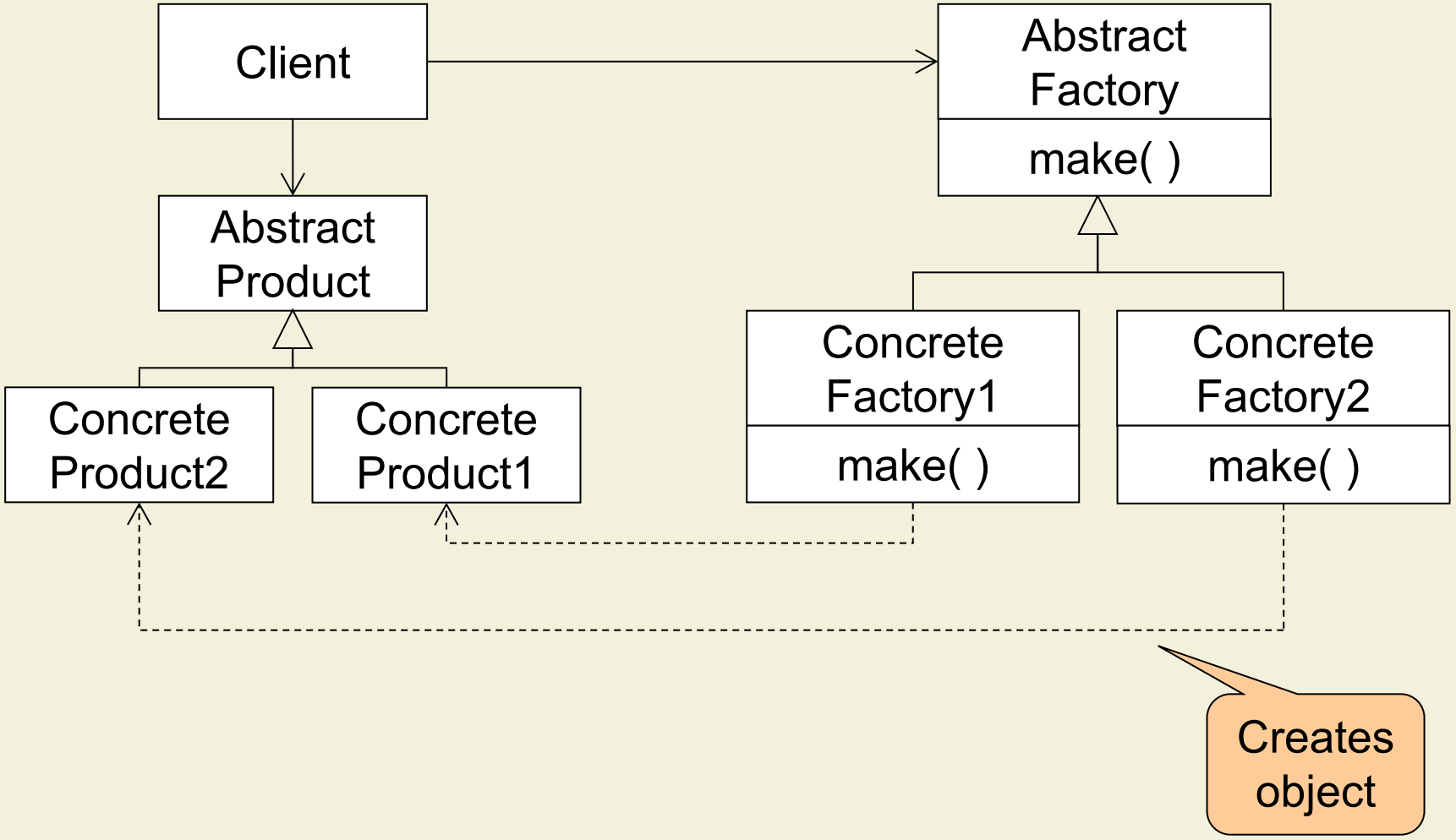
```
class SymbolTable {
  Map<Ident, Type> types;

  SymbolTable( Map<Ident, Type> t ) {
    types = t;
  }
}
```

Approach 3: Delegating Allocations

- Dependency injection
  - The dependencies between classes are defined in a separate configuration file (e.g., in XML)
  - Frameworks use this information to initialize fields (e.g., via reflection)

- Factories
  - Delegate allocations to a dedicated class called an abstract factory
  - Different concrete factory classes make objects of different classes
  - The concrete factory to be used is chosen by the client

Abstract Factory Pattern



Abstract Factory Example

```
interface MapFactory<K,V> { Map<K,V> make( ); }
```

```
class TreeMapFactory implements MapFactory<K,V> {
  Map<K,V> make( ) { return new TreeMap<K,V>( ); }
}
```

```
class SymbolTable {
  MapFactory<Ident, Type> factory;
  Map<Ident, Type> types;

  SymbolTable( MapFactory<Ident, Type> f ) {
    factory = f;
    types = factory.make( );
  }
}
```

Symboltable is no longer coupled to TreeMap

Coupling: Summary

- Low coupling is a general design goal

- However, there are trade-offs
  - Cohesion: each module has a clear responsibility
  - Performance and convenience
    (e.g., List and Iterator access nodes)
  - Adaptability: some design patterns increase coupling to improve adaptability
  - Code duplication

- Coupling to stable classes is less critical
  - For example, using or inheriting from library classes

4. Modularity

4.1 Coupling
4.2 Adaptation
4.2.1 Parameterization
4.2.2 Specialization

Change

- Since software is (perceived as being) easy to change, software systems often deviate from their initial design

- Typical changes include
  - New features (requested by customers or management)
  - New interfaces (new hardware, new or changed interfaces to other software systems)
  - Bug fixing, performance tuning

- Changes often erode the structure of the system

Parameterization

- Modules can be prepared for change by allowing clients to influence their behavior

- Make modules parametric in:
  - The values they manipulate
  - The data structures they operate on
  - The types they operate on
  - The algorithms they apply

- *One man's constant is another man's variable.*

[Alan J. Perlis]

Parameterization: Example

Source of data is a fixed class

Number of sources is fixed

Type of data is fixed

Alternation between sources is fixed

Filter criterion is fixed

```
class Merger {
  StringStream f1, f2;
  boolean toggle;

  String getNext( ) {
    String res = null;
    do {
      res = (toggle ? f1.getNext( )
                    :  f2.getNext( ));
    } while( res == null );
    toggle = !toggle;
    return res;
  }
}
```

```
class StringStream {
  String getNext( ) { … }
}
```

Parameterizing Values

- Modules can be made parametric by using variable values instead of constant values

```
class Merger {
  StringStream[ ] streams;
  int next;

  String getNext( ) {
    String res = null;
    do {
      res = streams[ next ].getNext( );
    } while( res == null );
    next = (next + 1) % streams.length;
    return res;
  }
}
```

Parameterizing Data Structures

- Modules can be made parametric by using interfaces and factories instead of concrete classes

```
class StringStream
      implements Filter {
  String getNext( ) { … }
}
```

```
class Merger {
  Filter[ ] filters;
  int next;

  String getNext( ) {
    String res = null;
    do {
      res = filters[ next ].getNext( );
    } while( res == null );
    next = (next + 1) % filters.length;
    return res;
  }
}
```

Parameterizing Types

- Modules can be made parametric by using generic types

```
class StringStream
      implements Filter<String> {
  String getNext( ) { … }
}
```

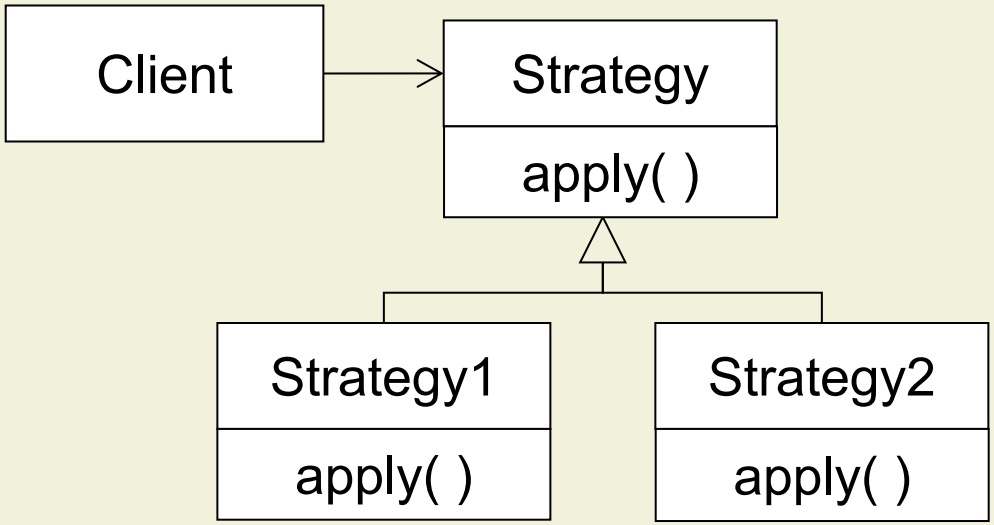
```
class Merger<D> {
  Filter<D>[ ] filters;
  int next;

  D getNext( ) {
    D res = null;
    do {
      res = filters[ next ].getNext( );
    } while( res == null );
    next = (next + 1) % filters.length;
    return res;
  }
}
```

Parameterizing Algorithms

- Modules can be made parametric by using function objects
  - Closures (Scala)
  - Delegates (C#)
  - Function pointers (C++)
  - Agents (Eiffel)
  - Lambda expressions (Java)

```
class Merger<D> {
  Filter<D>[ ] filters;
  int next;
  Selector<D> s;

  D getNext( ) {
    D res = null;
    do {
      res = filters[ next ].getNext( );
    } while( !s.select( res ) );
    next = (next + 1) % filters.length;
    return res;
  }
}
```

Strategy Pattern




```
interface Selector<D> {
  boolean select( D val );
}
```

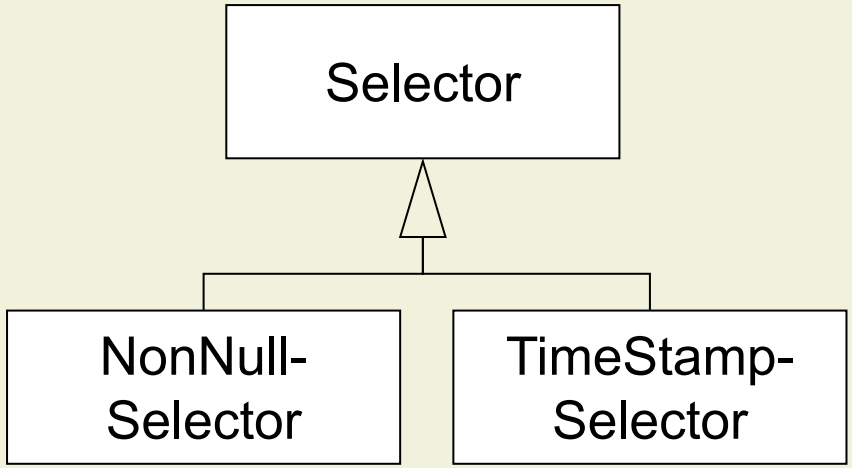
```
class NonNullSelector<D>
      implements Selector<D> {
  boolean select( D val ) {
    return val != null;
  }
}
```

4. Modularity

4.1 Coupling
4.2 Adaptation
4.2.1 Parameterization
4.2.2 Specialization

Dynamic Method Binding

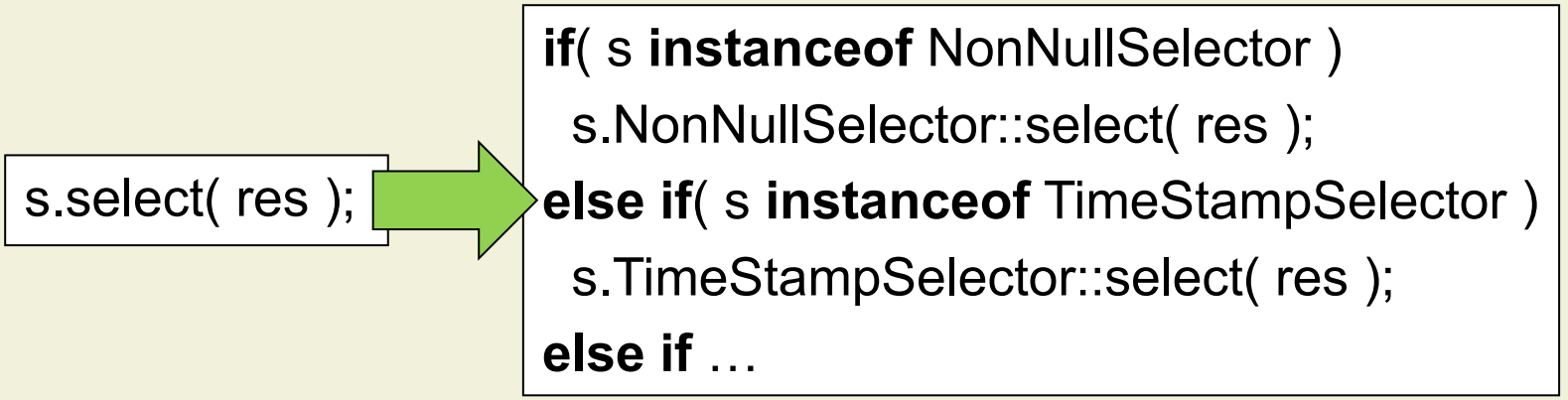
- In object-oriented programs, behaviors can be specialized via overriding and dynamic method binding




```
class Merger<D> {
  Filter<D>[ ] filters;
  int next;
  Selector<D> s;

  D getNext( ) {
    D res = null;
    do {
      res = filters[ next ].getNext( );
    } while( !s.select( res ) );
    next = (next + 1) % filters.length;
    return res;
  }
}
```

Dynamic Method Binding as Case Distinction

- Dynamic method binding is a case distinction on the dynamic type of the receiver object



- Adding or removing cases (method overrides) does not require changes in the caller
  - Client code is adaptable

Static vs. Dynamic Method Binding

- Dynamic method binding has drawbacks
  - Reasoning: Subclasses share responsibility for maintaining invariants
  - Testing: Dynamic binding increases the number of possible behaviors that need to be tested
  - Versioning: Dynamic binding makes it harder to evolve code without breaking subclasses
  - Performance: Overhead of method look-up at run-time
- Choose binding carefully for each method
  - Java: Consider making methods final
  - C++, C#: Consider making methods virtual

Replacing Case Distinctions by Dyn. Binding

```
class Movie {
  static final int REGULAR = 0;
  static final int CHILDREN = 1;

  int _priceCode;

  int getCharge( int days )  {
    if( _priceCode == REGULAR )
      return days * 3;
    else
      return days * 2;
  }
}
```

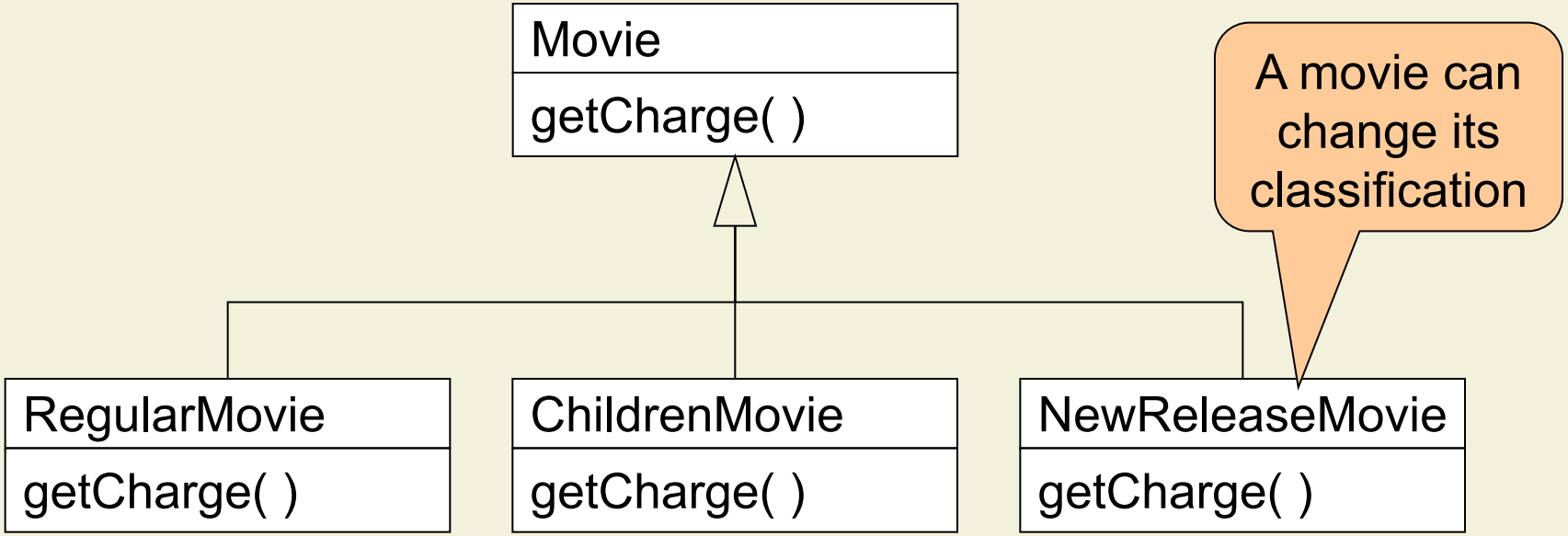
Introducing new price codes requires changes

```
abstract class Movie {
  abstract int getCharge( int days );
}
```

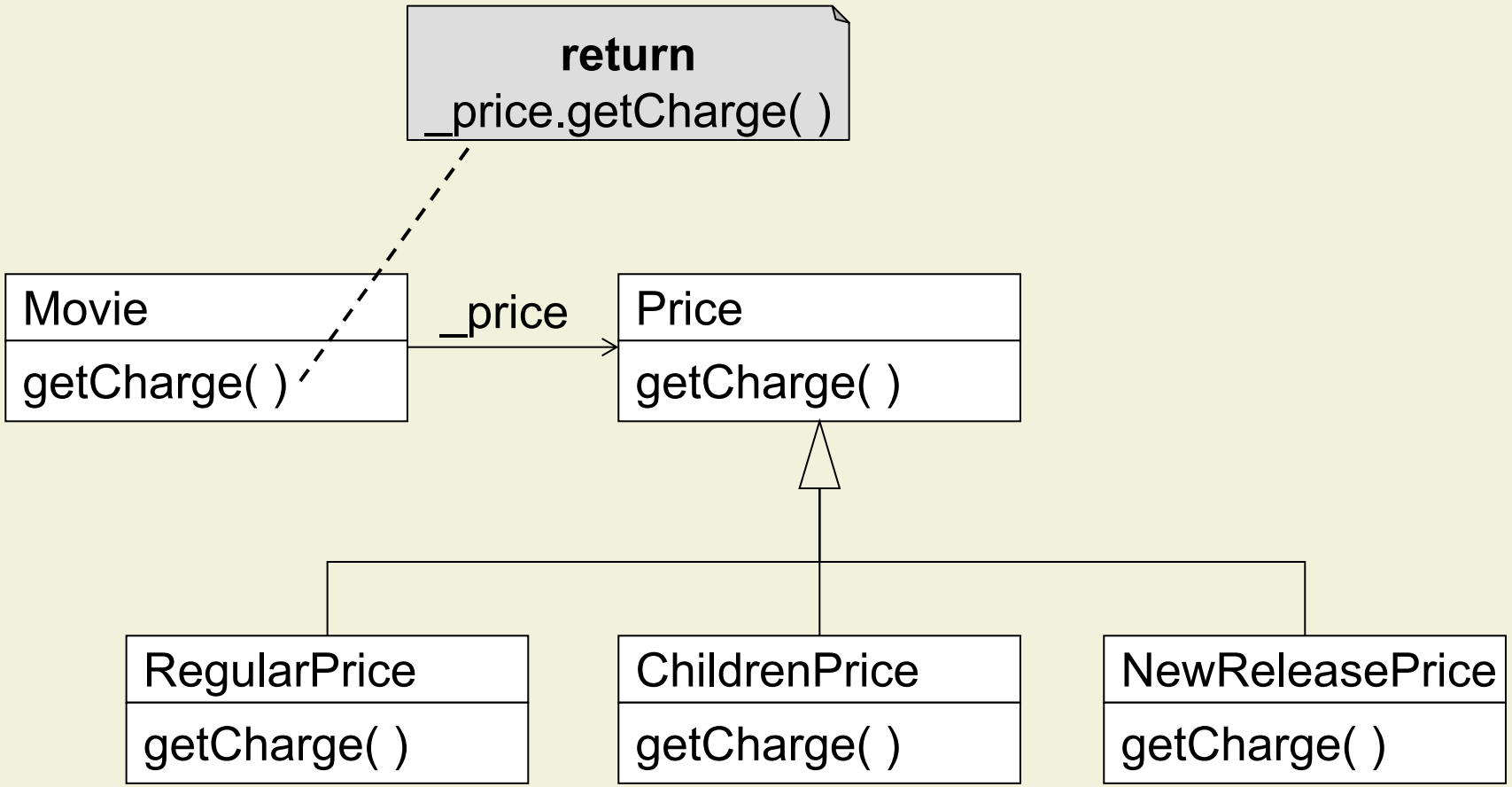
```
class RegularMovie extends Movie {
  int getCharge( int days )  {
    return days * 3;
  }
}
```

```
class ChildrenMovie extends Movie {
  int getCharge( int days )  {
    return days * 2;
  }
}
```

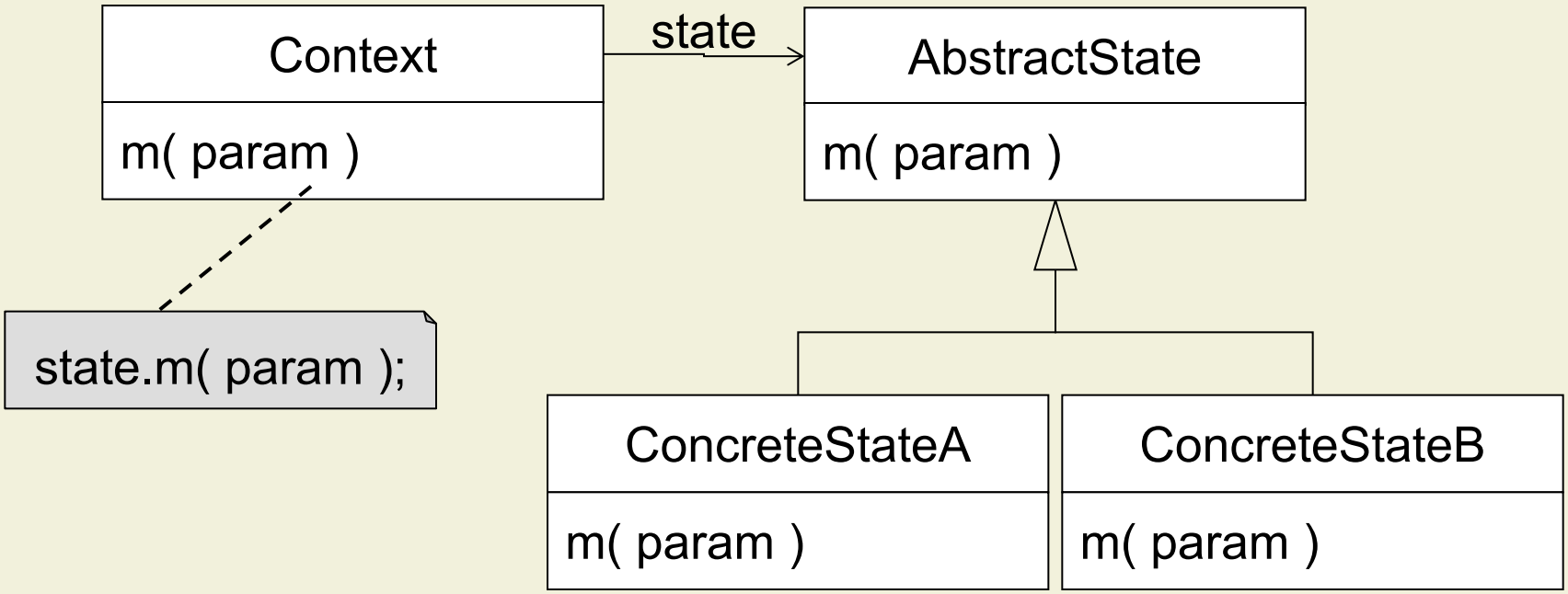
Adaptation to New Cases



Using Aggregation Plus Dynamic Binding

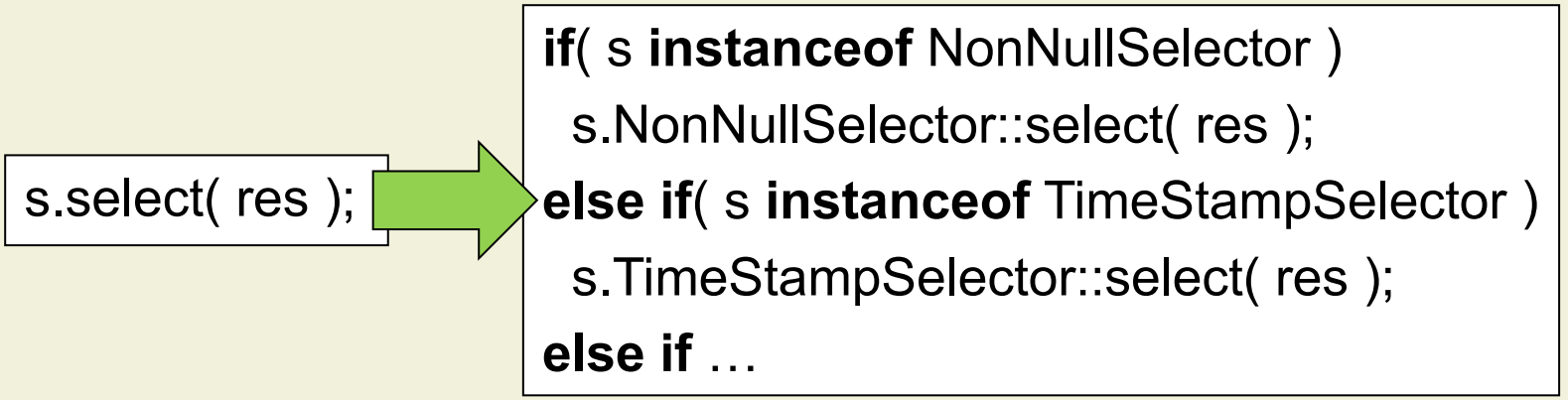


State Pattern



Case Distinction on Several Arguments

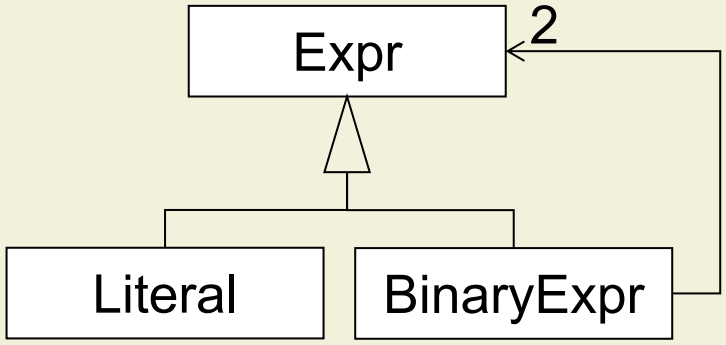
- Dynamic method binding is a case distinction on the dynamic type of the receiver object



- In some cases, it is useful to select an operation based on the dynamic type of the receiver object and of the argument(s)

Example: Operations on a Syntax Tree

- Consider a data structures with nodes of different types
- The behavior of operations depends on the type of node it is applied to
- The set of operations is not fixed



- Operations
  - Type checking
  - Evaluation
  - Code generation
  - Pretty printing

Double Invocation

```
abstract class Expr {
  abstract void accept( Visitor v );
}
```

```
class Literal extends Expr {
  int val;

  void accept( Visitor v ) {
    v.visitLiteral( this );
  }
}
```

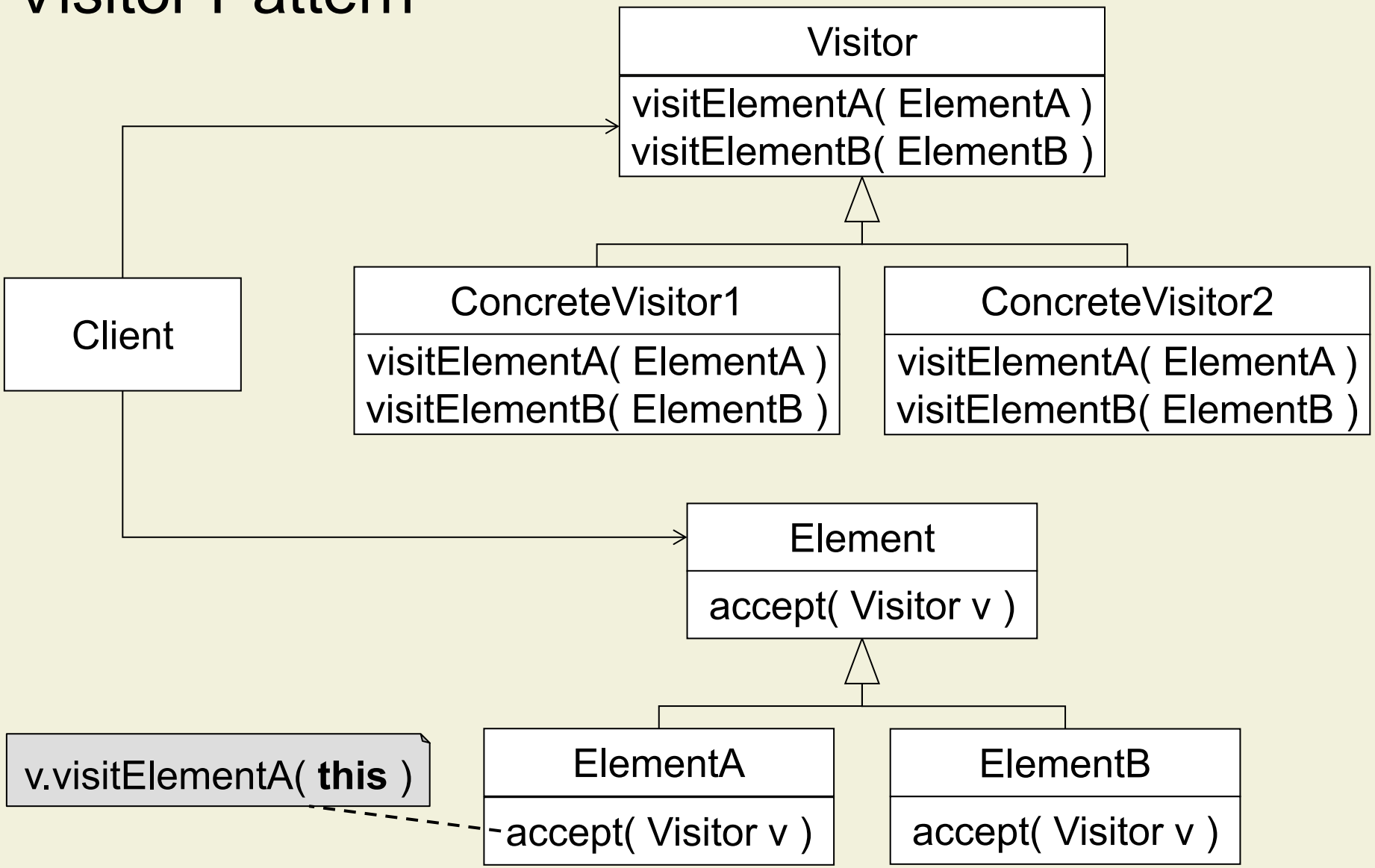
```
class Binary extends Expr {
  void accept( Visitor v ) {
    v.visitBinary( this );
  }
}
```

```
abstract class Visitor {
  abstract void visitLiteral( Literal e );
  abstract void visitBinary( Binary e );
}
```

```
class Evaluator extends Visitor {
  int value;
  void visitLiteral( Literal e ) {
    value = e.val;
  }

  void visitBinary( Binary e)  { … }
}
```

```
class PrettyPrinter extends Visitor {
  …
}
```

Visitor Pattern



Adaptation: Summary

- Designing adaptable modules
  - Makes inevitable changes easier
  - Facilitates reuse

- Parameterization allows clients to customize the behavior by supplying different parameters

- Specialization allows clients to customize behavior by adding subclasses and overriding methods