# Theme 2
# Program Design
## A Pattern Catalog

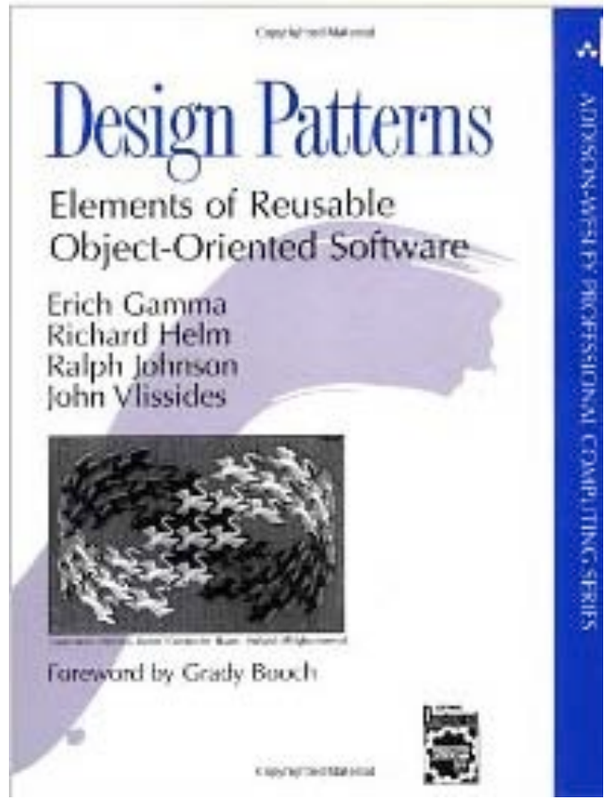
# Learning Objectives

- to present principles for flexible design and see how these principles map to the compositional process
- to present additional definitions of design patterns in order to convey an even deeper understanding of the concept
- to show how patterns are key concepts for understanding and keeping overview of complex, compositional, designs by presenting a set of patterns

# Table of Contents

# Design Pattern Book

- Original Design Pattern book is organised as a catalogue of 23 patterns AND
- introduction on various aspects of writing reusable and flexible software.

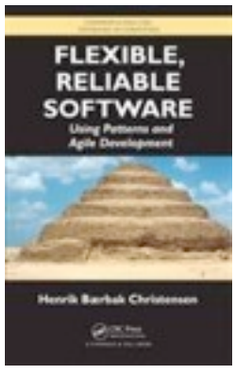


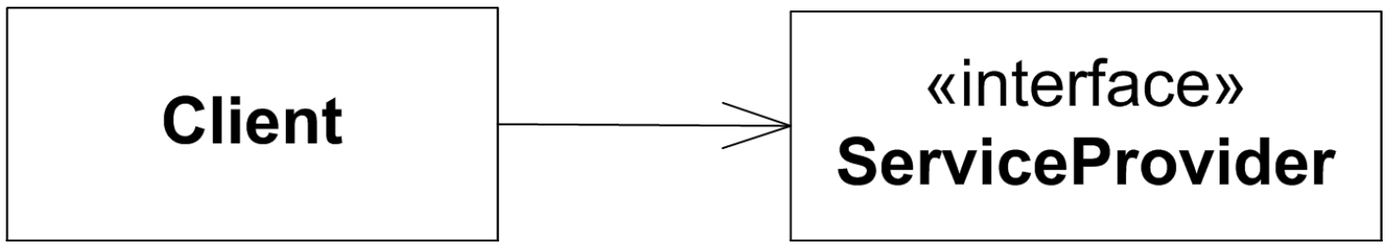
# Principles for Flexible Design


1. Program to an interface, not an implementation
2. Favour object composition over class inheritance
3. Consider what should be variable in your design.

Let's go over these principles:

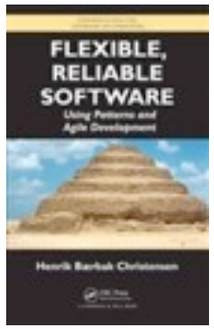
# First Principle



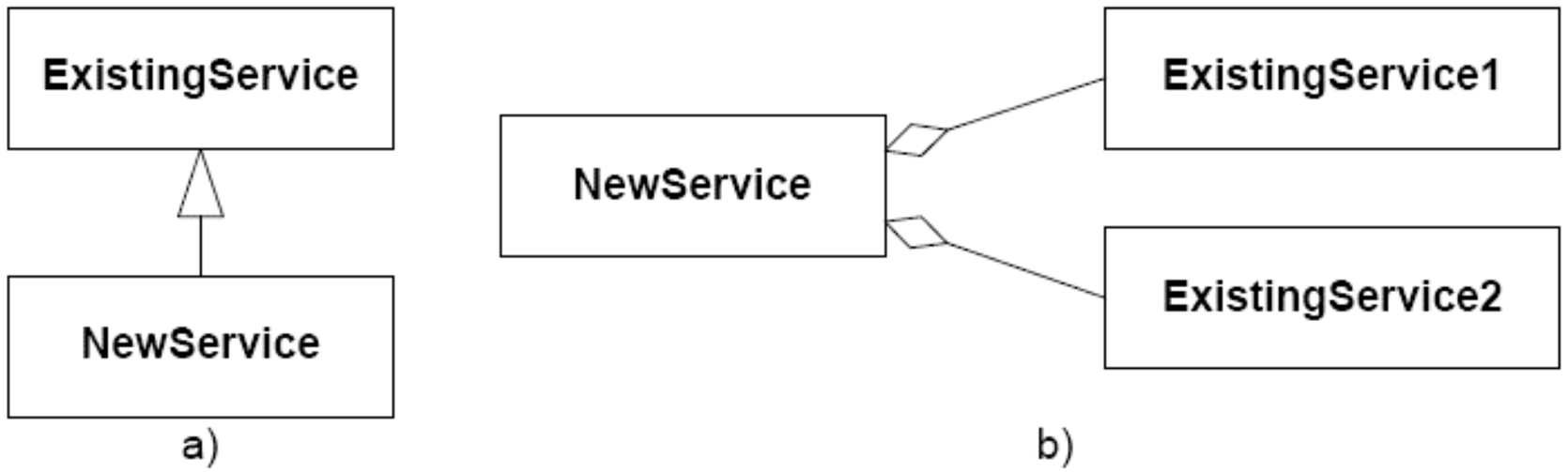




Assume only the contract, i.e.,
the responsibilities

# Second Principle



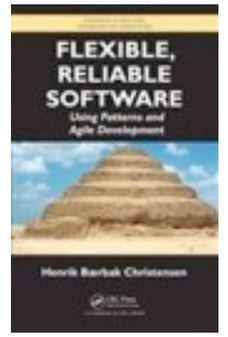

Two ways to reuse code in OO



# Analysis

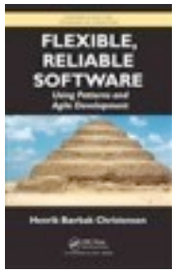

- Inheritance breaks encapsulation
- only add responsibilities, do not remove them
- compile-time binding
- recurring modifications
- separate testing
- increased possibility of reuse

# Third Principle




- Consider what should be variable in your design
  - closely linked to:
  - Change by addition, not by modification
    - Identify
      - design/code that should remain stable
      - design/code that may vary
      - use techniques that ensure that the stable part remains stable!!

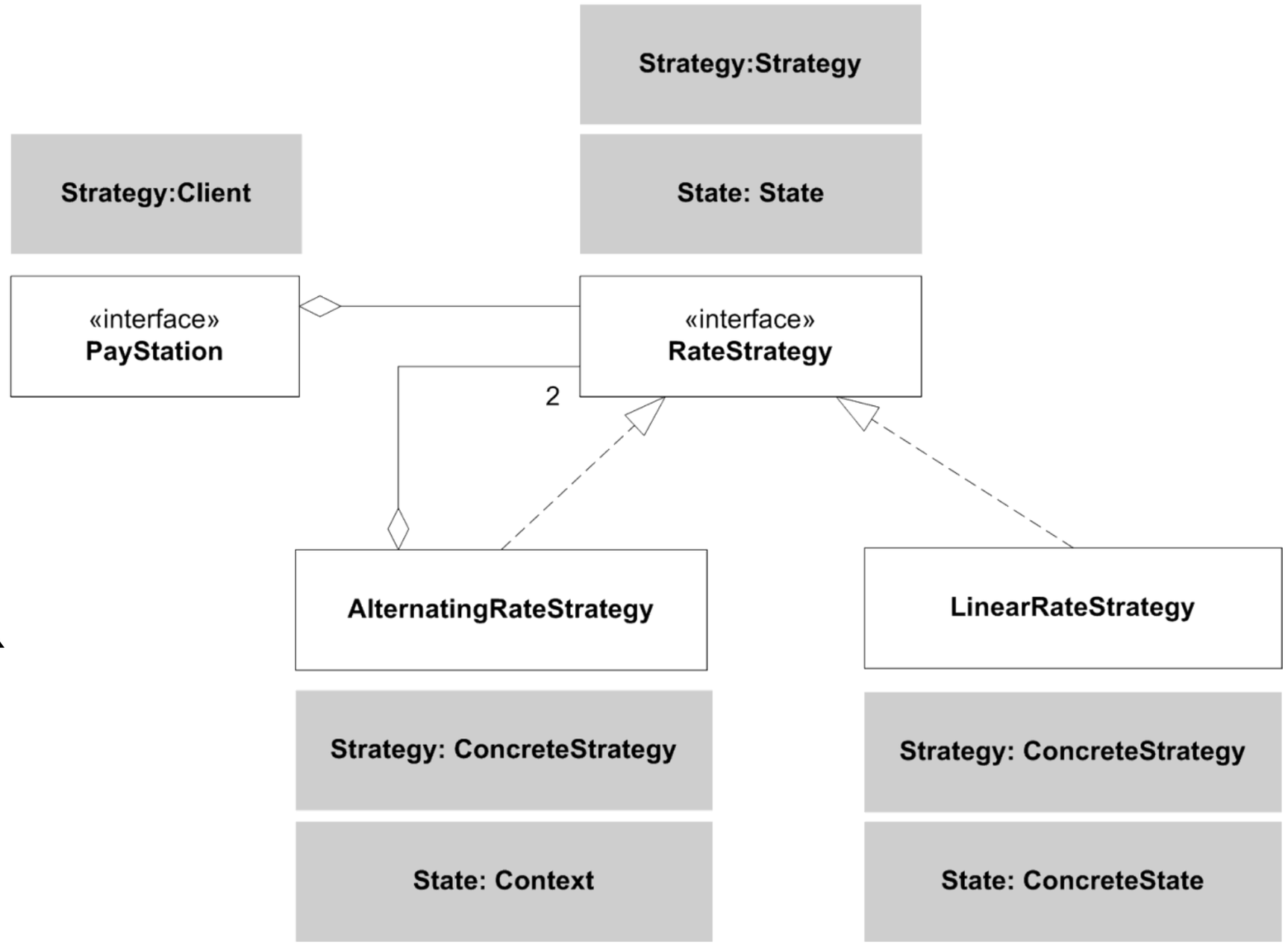
# Summary



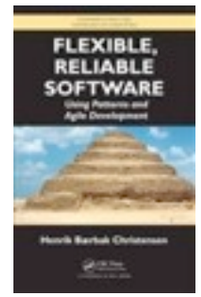

- We identified some behaviour that was likely to change…
- We stated a well defined responsibility that covers this behaviour and expressed it in an interface
- Instead of performing behaviour ourselves we delegated to an object implementing the interface
- Consider what should be variable in your design
- Program to an interface, not an implementation
- Favor object composition over class inheritance

# Design Patterns: another definition


REMEMBER



# Design Pattern - Role View




- The essence of design patterns is at a higher level of abstraction than what you may see in e.g. UML class diagrams.
- **Design Pattern (Role view)**
  A design pattern is defined by a set of roles, each role having a specific set of responsibilities, and by a well defined protocol between these roles.

# Sorts of Design Patterns


**Creational Patterns:**
are concerned with the process of object creation

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

purpose

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

**Class Patterns** deal with static relationships between classes and subclasses

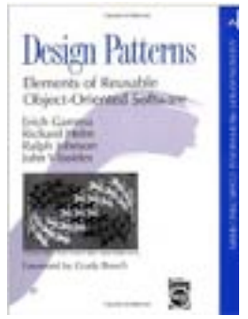
scope

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

# Creational Patterns


- Abstracts the instantiation process:
  - Encapsulates knowledge about which concrete classes to use
  - Hides how the instances of these classes are created and put together
  - Gives a lot of flexibility in what gets created, who creates it, how it gets created, and when it it gets created
  - A class creational pattern uses inheritance to vary the class that is instantiated
  - An object creational pattern delegates instantiation to another object

# Structural Design Patterns




- Structural design patterns are concerned with how classes and objects are composed to form larger structures
- A class structural pattern uses inheritance to compose interfaces or implementation; compositions are fixed at design time
- An object structural  pattern describes ways to compose objects to realise new functionality; the added flexibility of object composition comes from the ability to change the composition at run-time

# Behavioural Design Patterns


- Behavioural design patterns are concerned with algorithms and the assignment of responsibilities between objects
- Behavioural class patterns use inheritance to distribute behaviour between classes
- Behavioural object patterns use object composition rather than inheritance;
  - some describe how a group of peer objects cooperate to perform a tasks no single object can carry out by itself;
  - others are concerned with encapsulating behaviour in an object and delegating request to it.

# Overview


**Creational Patterns**

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

**Structural Patterns**

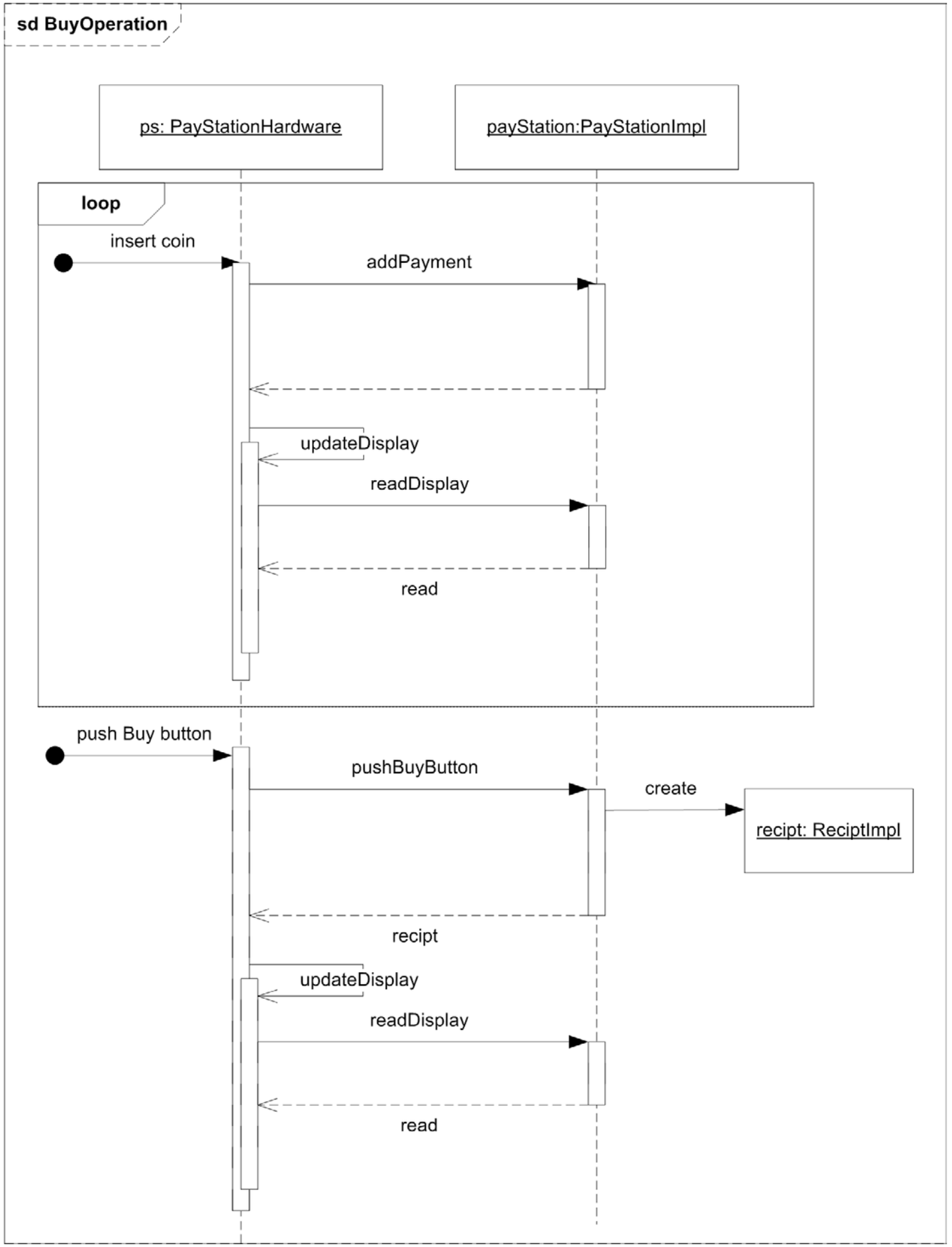
- ✓ Composite
- ✓ Façade
- ✓ Proxy
- ✓ Flyweight
- ✓ Adapter
- ✓ Bridge
- ✓ Decorator

**Behavioral Patterns**

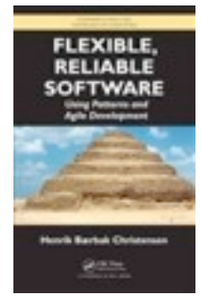
- ✓ Chain of Respons.
- ✓ Command
- ✓ Interpreter
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template Method
- ✓ Visitor

# Façade


FLEXIBLE,
RELIABLE
SOFTWARE

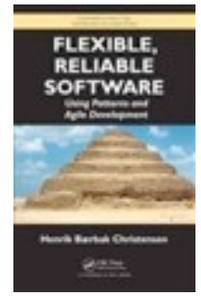

sd BuyOperation
ps: PayStationHardware
payStation:PayStationImpl
loop
insert coin
addPayment
updateDisplay
readDisplay
read
push Buy button
pushBuyButton
create
recipt: ReciptImpl
recipt
updateDisplay
readDisplay
read

# Design Considerations



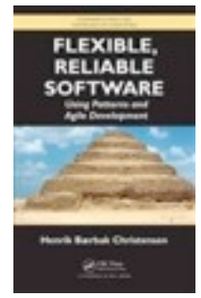

- Behaviour that may vary
  - the same hardware must operate varying  pay station implementations: AlphaTown, BetaTown, EpsilonTown...
- Variable behaviour behind  interface
  - *PayStation* interface...
- Compose behaviour by delegation
  - Gui/Hardware does not itself calculate rates, issue receipts, etc., but lets an instance of *PayStation* do the dirty job...

# Result

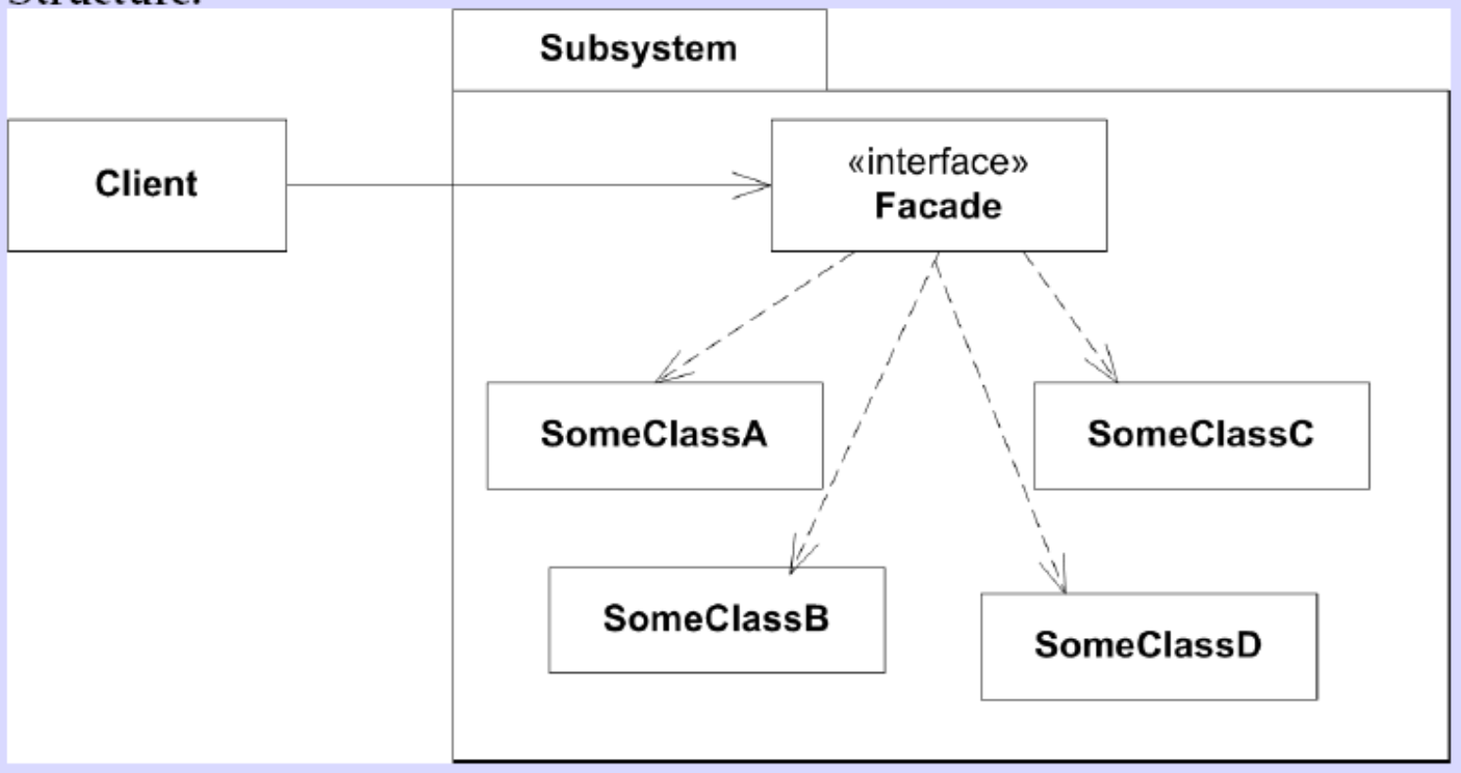



- The side effect of this decisions is that interface decouples both ways!!!
  - Hardware may operate different kinds of *PayStation* implementations
    - Alpha, Beta, Gamma, …
  - Different kinds of user interfaces may operate the same *PayStation* implementation

# PayStation is an example of the Facade pattern



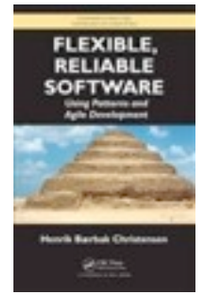

Structure:




**Intent**: Provide a unified higher level interface to a set of interfaces in a subsystem to make the subsystem easier to use.

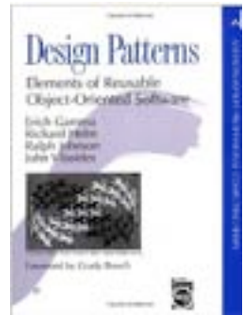
The **Facade** knows which **subsystem classes** are responsible for a request and delegates clients requests to appropriate subsystem objects. The **subsystems classes** implement the subsystem functionality and handle work assigned by the Facade object but they have no knowledge of the Facade object.

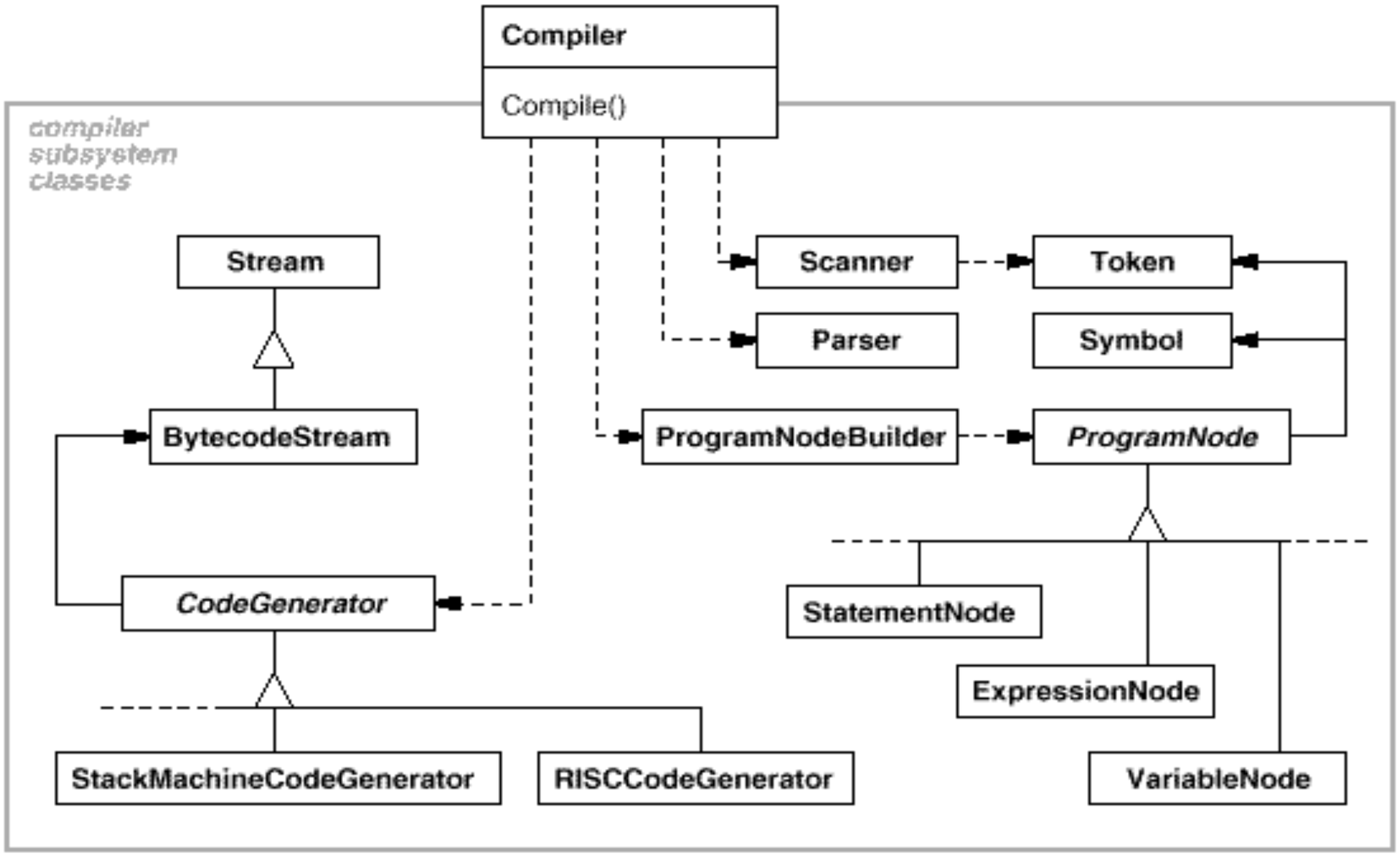
# Consequences




- Benefits
  - shields clients from subsystem objects
  - weak coupling
    - many to many relation between client and façade
- Liabilities
  - bloated interface with lots of methods
    - because façade must have the sum of responsibilities of the subsystem
  - how to avoid access to the inner objects?
    - read-only interfaces; no access (require dumb data objects to be passed and parsed over the façade).

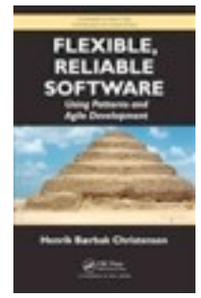
# Another example



# Decorator

# New Requirement (again)




- Alphatown wants to log all coin entries:
  - [time] [value]

- Example:
  - 14:05:12  5 cent
  - 14:05:14  25 cent
  - 14:55:10  25 cent

# The Compositional Approach


- Identify the responsibility whose concrete behaviour may vary.
- Express responsibility as an interface.
- Let someone else do the job.

- How does this apply?

# The Compositional Approach


- Identify the responsibility whose concrete behaviour may vary
  - It is the "Accept payment" responsibility

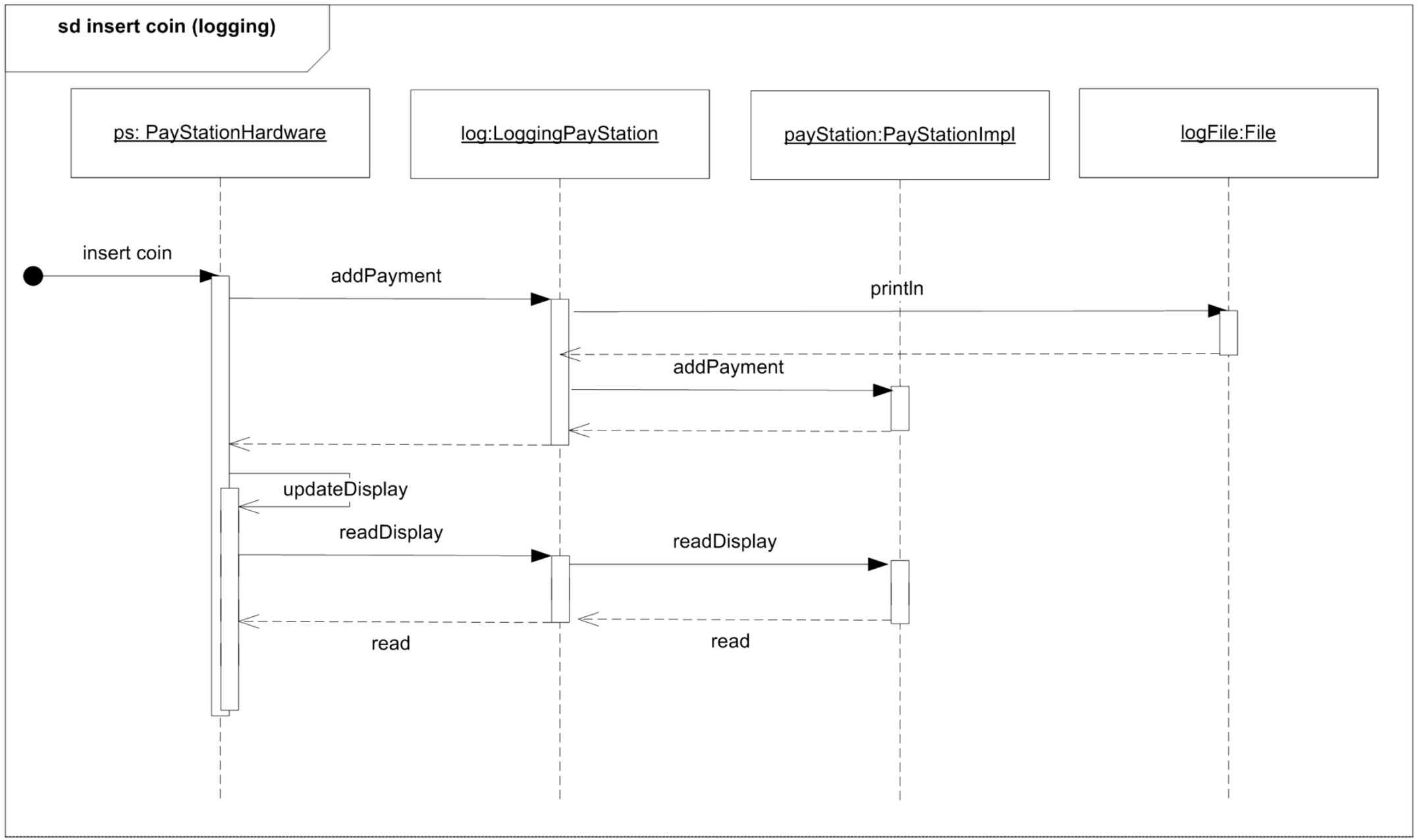
- Express responsibility as an interface

  A. PaymentAccepter role?

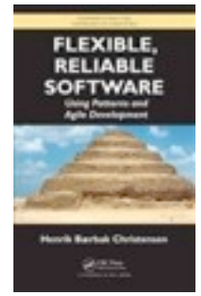
  B. PayStation role? Already in place!

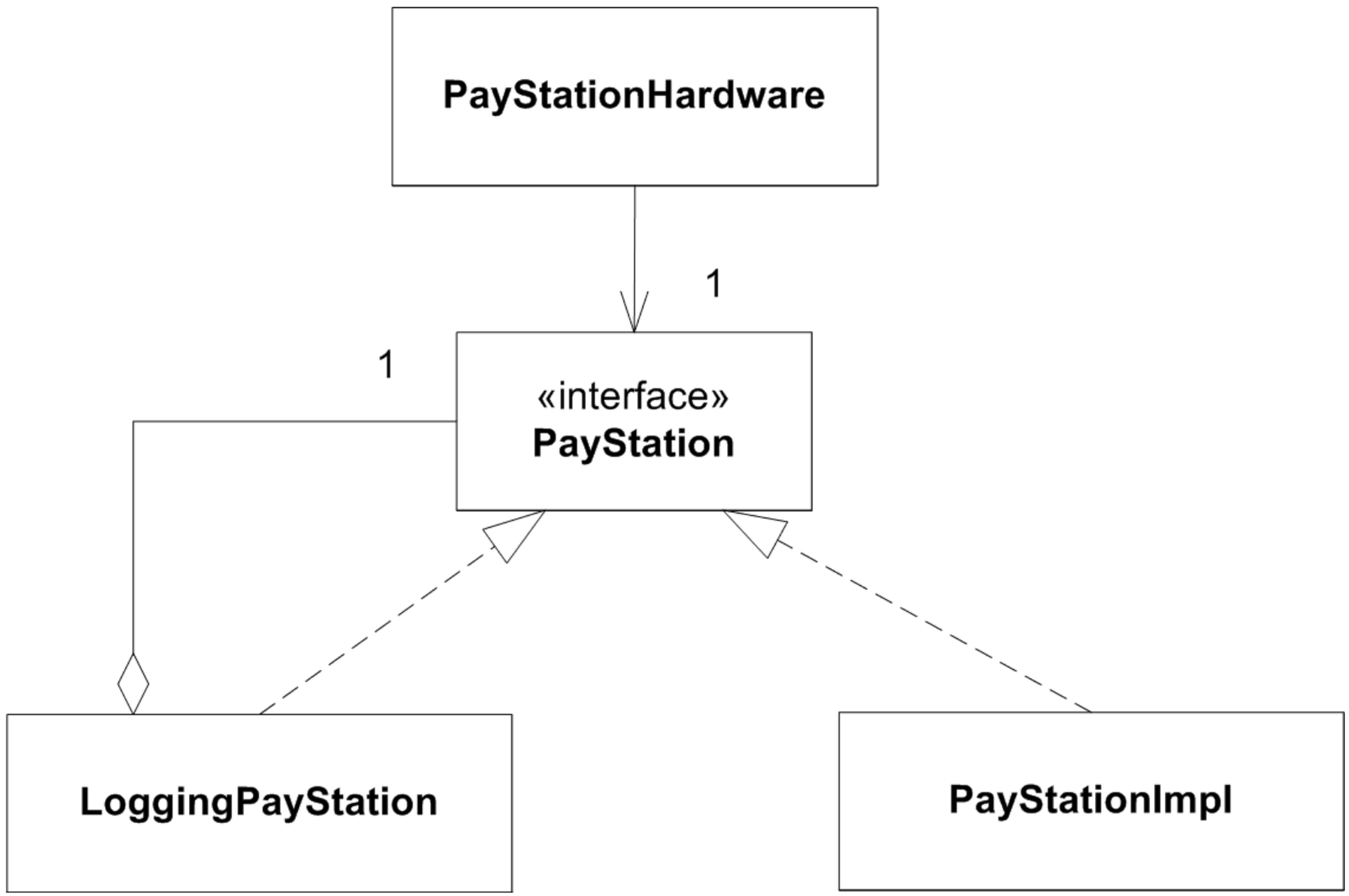
- Let someone else do the job
  - Maybe let someone handle the coins before the parking machine receives them?

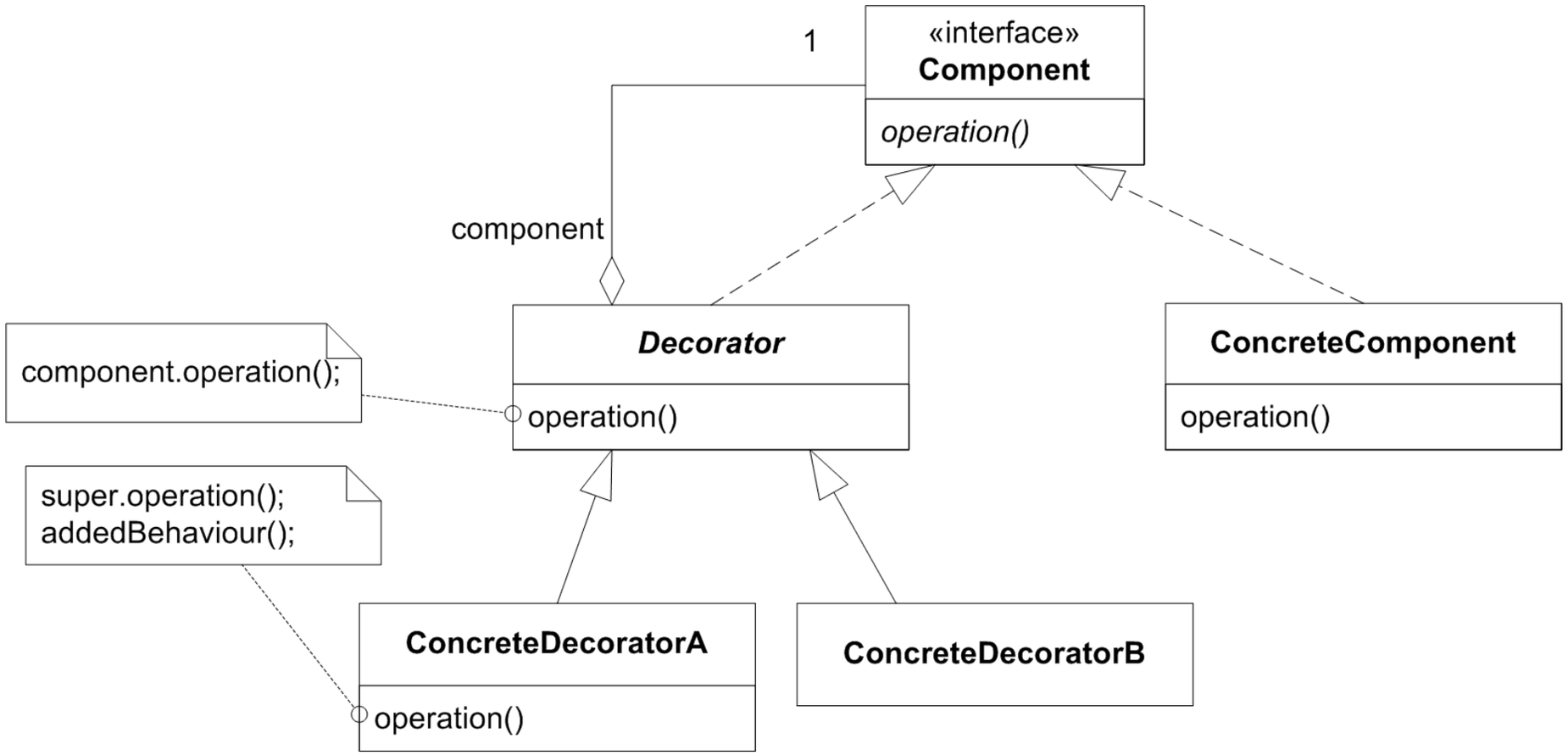
# The Dynamics

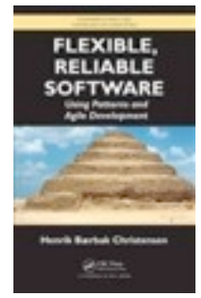



# The Structure






**Intent:** Attach additional responsibilities tot an object dynamically instead of relying on subclassing to extend functionality.
Also known as: Wrapper

**Component** defines the interface for objects that can have responsibilities added to them dynamically. A **ConcreteComponent** defines an objects to which additional responsibilities can be attached. The **Decorator** maintains a reference to a component object and defines an interface that conforms to the **Component's** interface. A **ConcreteDecorator** adds responsibilities to the component.

# Consequences


- Benefits
  - adding and removing behaviour at run-time
  - incrementally add responsibilities
  - complex behaviour by chaining decorators
- Liabilities
  - Analysability suffers as you end up with lots of little objects
    - Behaviour is constructed at run-time instead of being written in the static code
  - Delegation code tedious to write

# Adapter

# New Requirement


- New Town wants to use an existing rate calculation algorithm
  - You must use the implementation bought from a consultancy company (closed source!)

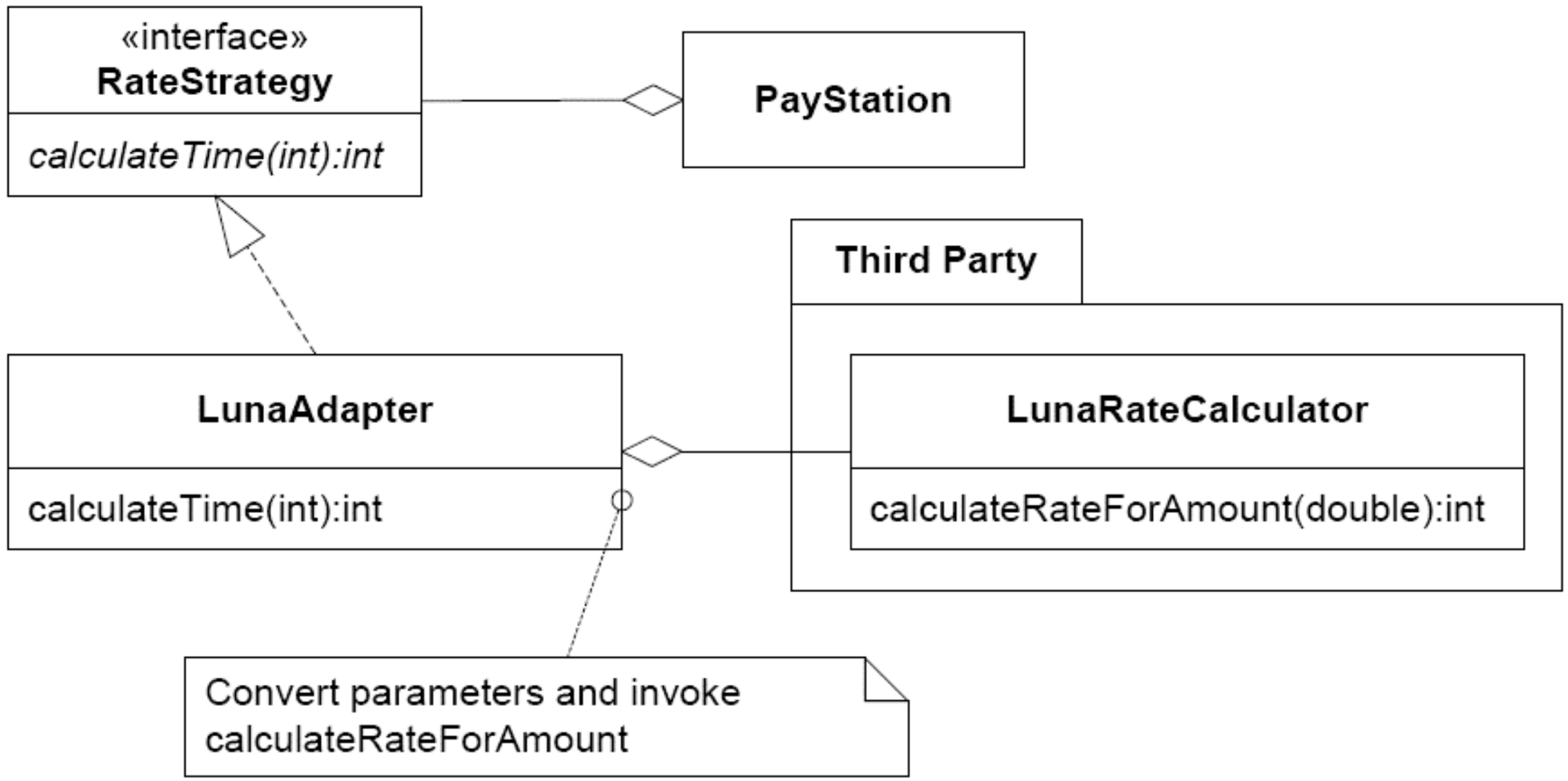
- Challenge:
  - The interface does not match ours

# Solution


- Compose behaviour even further then we already did!
- *Put an intermediate object between the two interfaces, one that does the translation from one interface to the other!*

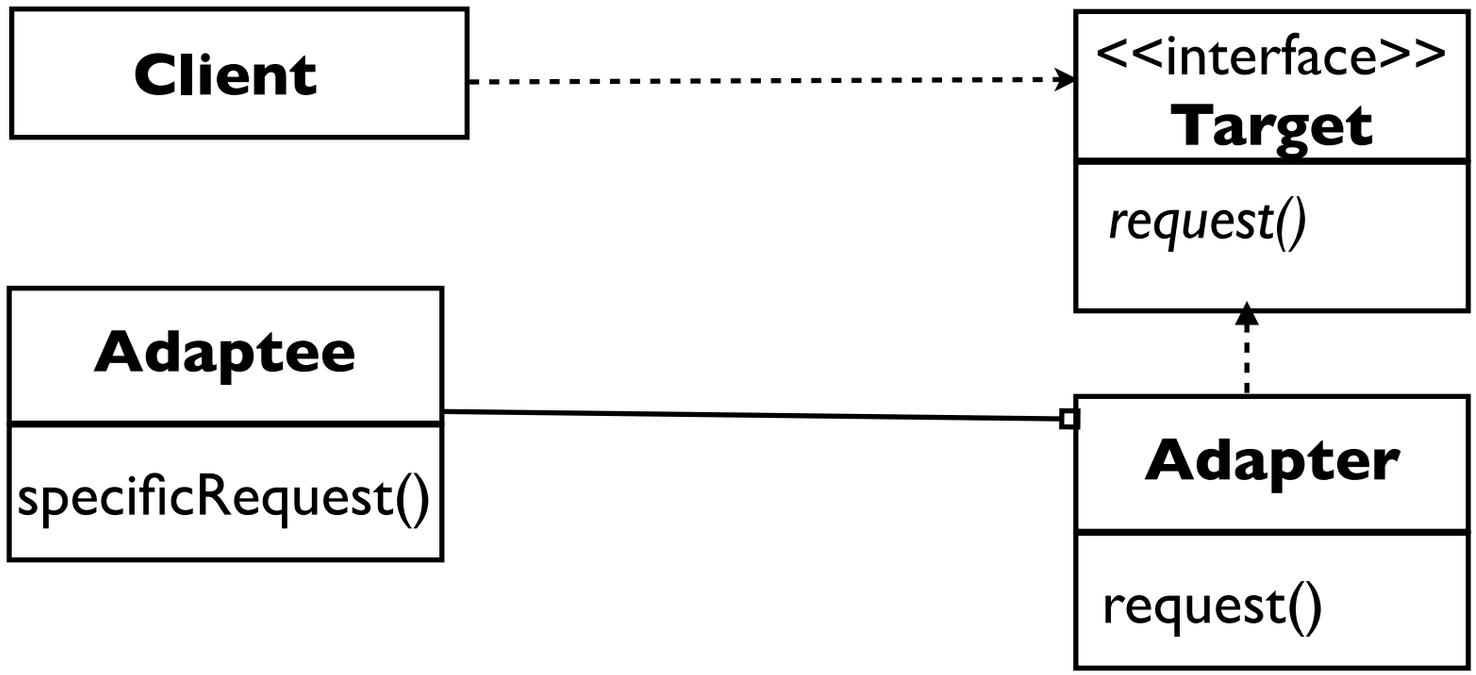
```
/** An adapter for adapting the Lunatown rate calculator
*/
public class LunaAdapter implements RateStrategy {
  private LunaRateCalculator calculator;
  public LunaAdapter() {
    calculator = new LunaRateCalculator();
  }

  public int calculateTime( int amount ) {
    double dollar = amount / 100.0;
    return calculator.calculateRateForAmount( dollar );
  }
}
```

# Structure of the Solution

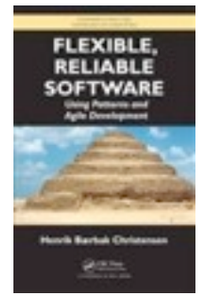
# Adapter Pattern


**Intent**: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.




**Targets** encapsulate behaviour used by the **client**. The **Adapter** implements the **Target** role and delegate actual processing to the **Adaptee** performing parameter and protocol translations in the process.

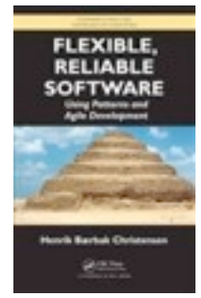
# Consequences




- Benefits
  - Makes a client work with an otherwise incompatible object
    - One adapter can adapt many type of adaptee's namely all subclasses
- Liabilities
  - Adaptation spectrum: from simple method name conversions to radically different interfaces

# Builder

# Problem




- Consider your favourite text editor, word processor, spreadsheet, drawing tool, ...
  - allow editing a *complex data structure* representing a document, spreadsheet, etc.
  - but also need to *save* it to a persistent store, typically a hard disk.
    - Converting internal data structure to external format
    - Ex: Binary encoding, XML, HTML, RTF, PDF, …

# Example


A document in XML

The section header can be output in a number of different storage formats, like:

HTML

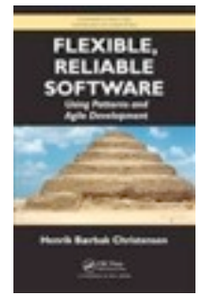
or ASCII

```
<document name=''The Pattern Book''>
  <section name=''Builder''>
    <paragraph>
      This is my section on builder.
    </paragraph>
    <subsection name= ``Analysis''>
      <paragraph>
        Here is my analysis of builder.
      </paragraph>
    </subsection>
  </section>
</document>
```

```
<H1>Builder</H1>
```

```
Builder
=======
```

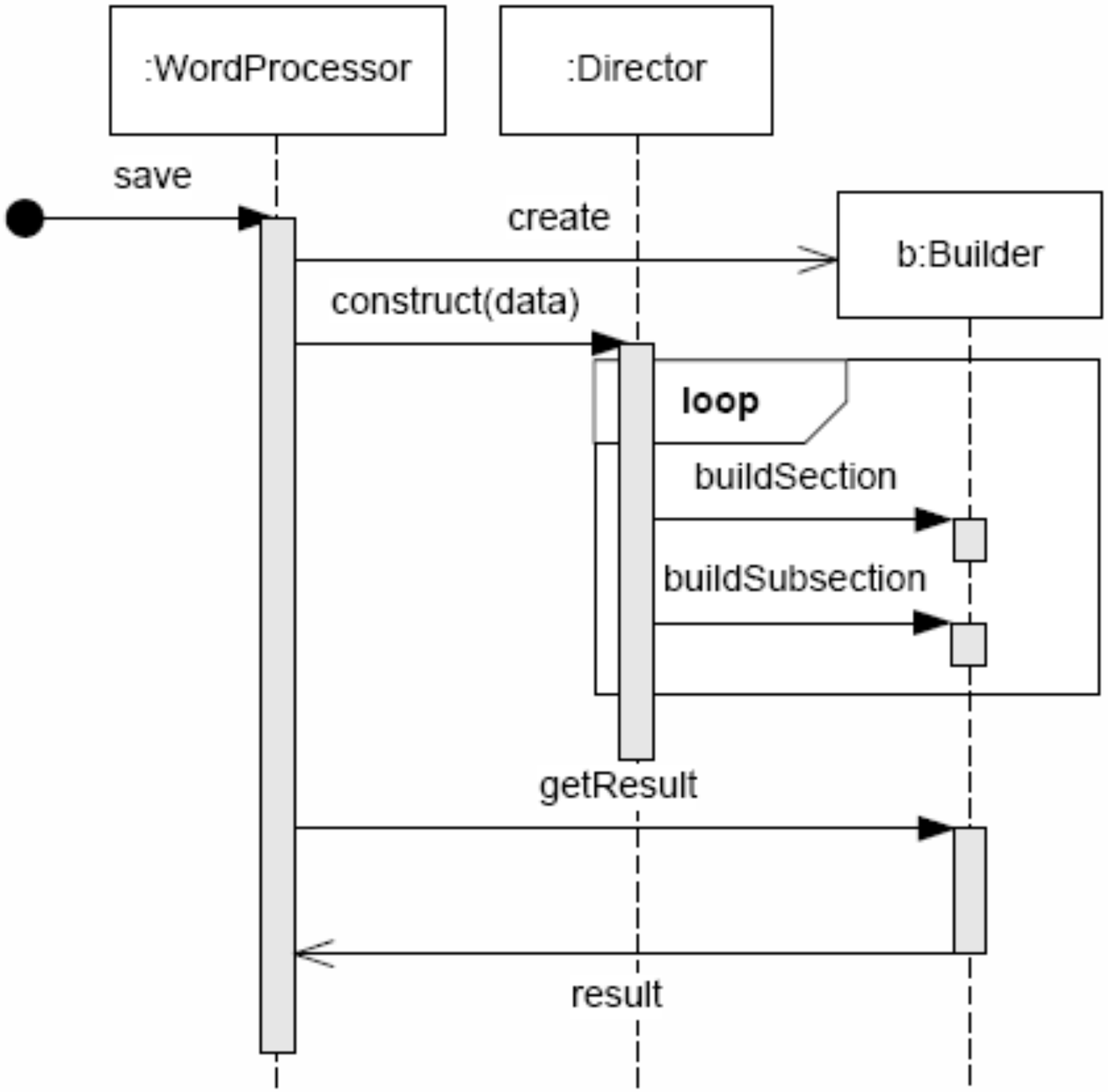
# Compositional Process




- All output consists of the same set of "parts" (section, subsection, paragraphs, etc.) but how the parts are built, varies.
  - That is, concrete construction of the individual node is variable
- Encapsulate the "construction of the parts" in a **builder** interface.
  - A builder interface must have methods to build each unique part.
- write the data structure iterator algorithm once, the **director**, and let it request a delegate builder to make the concrete parts.

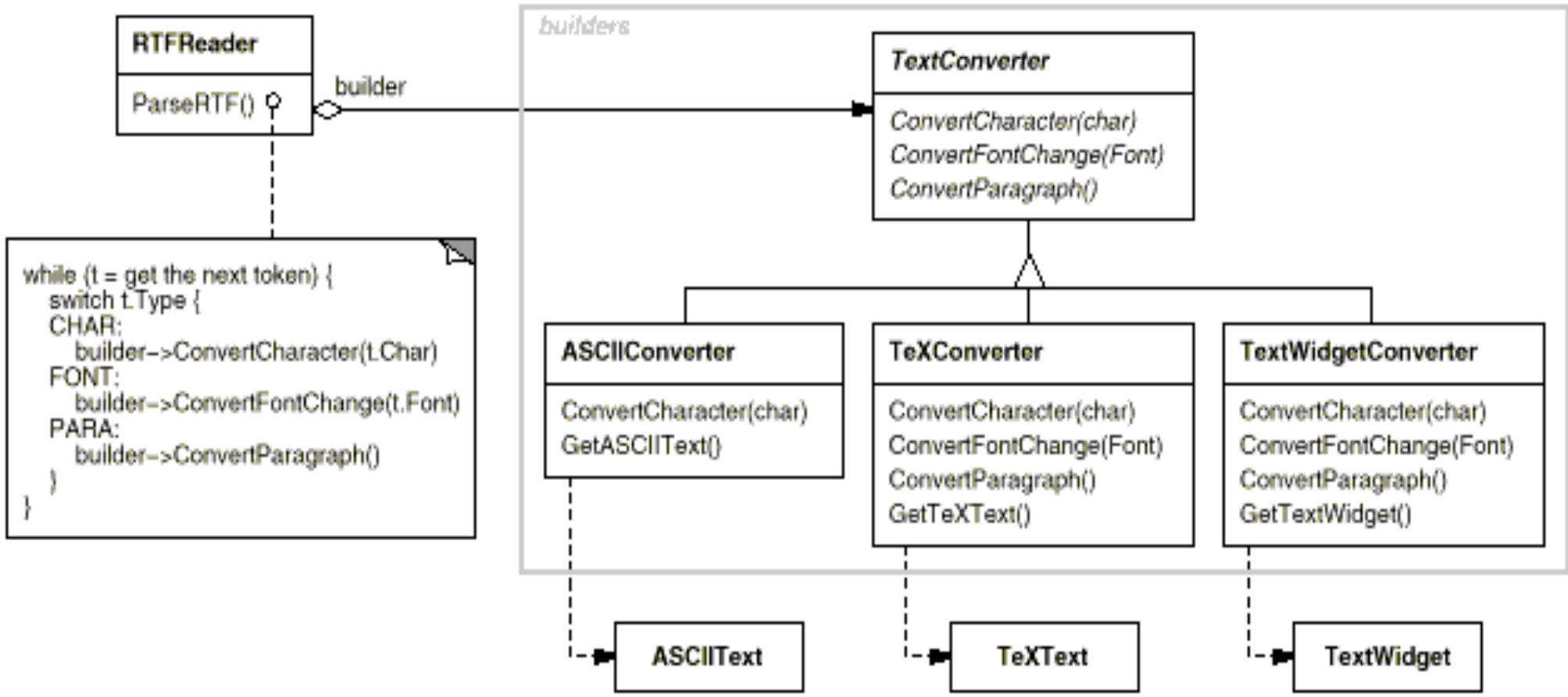
# Dynamics


FLEXIBLE,
RELIABLE
SOFTWARE

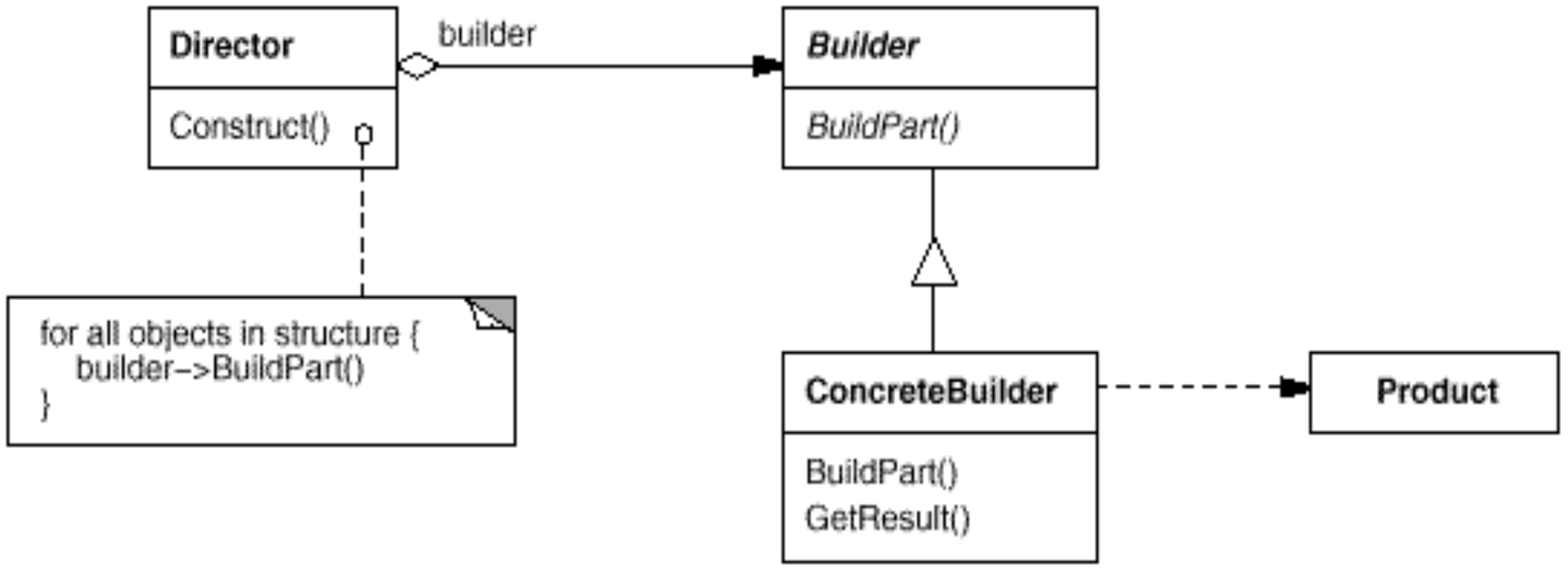

:WordProcessor
:Director
save
create
b:Builder
construct(data)
loop
buildSection
buildSubsection
getResult
result

# Builder


**Intent**: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
(You have a single defined construction process but the output format varies.)

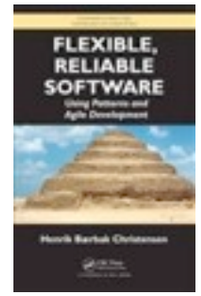


# Builder


**Builder**:specifies an abstract interface for creating parts of a **Product. A ConcreteBuilder** constructs and assembles parts of the **Product** by implementing the **Builder** interface, defines and keeps track of the representation it creates, and provides an interface for retrieving the product. The **Director** constructs an object using the Builder interface. The **Product** represents the complex object under construction and includes classes that define the constituent parts including the interfaces for assembling the parts into the final result.

# Consequences



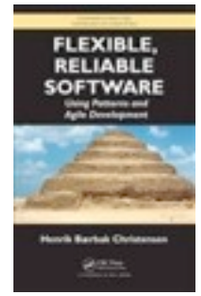

- Benefits are
  - Fine grained control over the building process
    - Compare to Abstract Factory
  - Construction process and part construction decoupled
    - Change by addition to support new formats
    - Many-to-many relation between directors and builders
      - Reuse the builders in other directors...
- Liabilities
  - Client must know the product of the builder as well as the concrete builder types.

# Implementation

- Assembly and construction interfaces:
  - The Builder interface must be general enough to allow the construction of products for all kinds of ConcreteBuilders
  - The model for construction and assembly is a key design issue
- Why no abstract class for products?:
  - In the common case, the products can differ so greatly in their representation that little is to gain from giving different products a common parent class.
  - Because the client configures the Director with the appropriate ConcreteBuilder, the client knows the resulting products.

# Command

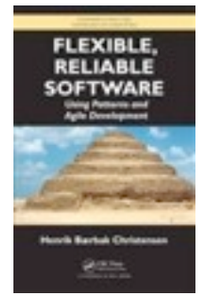
# Problem




- In my word processor system I would like the user to configure what the F1 button does freely
  - Like ‘save’ or ‘open new file’ or ?
  - Or perhaps record a macro of key strokes in F1
    - F1 => insert text ‘hallo’ at the cursor position
- But how to code this?

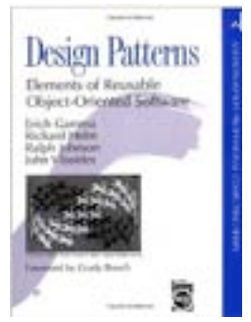
```
public void F1Press() {
  editor.showFileDialogAndOpen();
or
  editor.save();
or
  some other behavior?
}
```

# Compositional Process

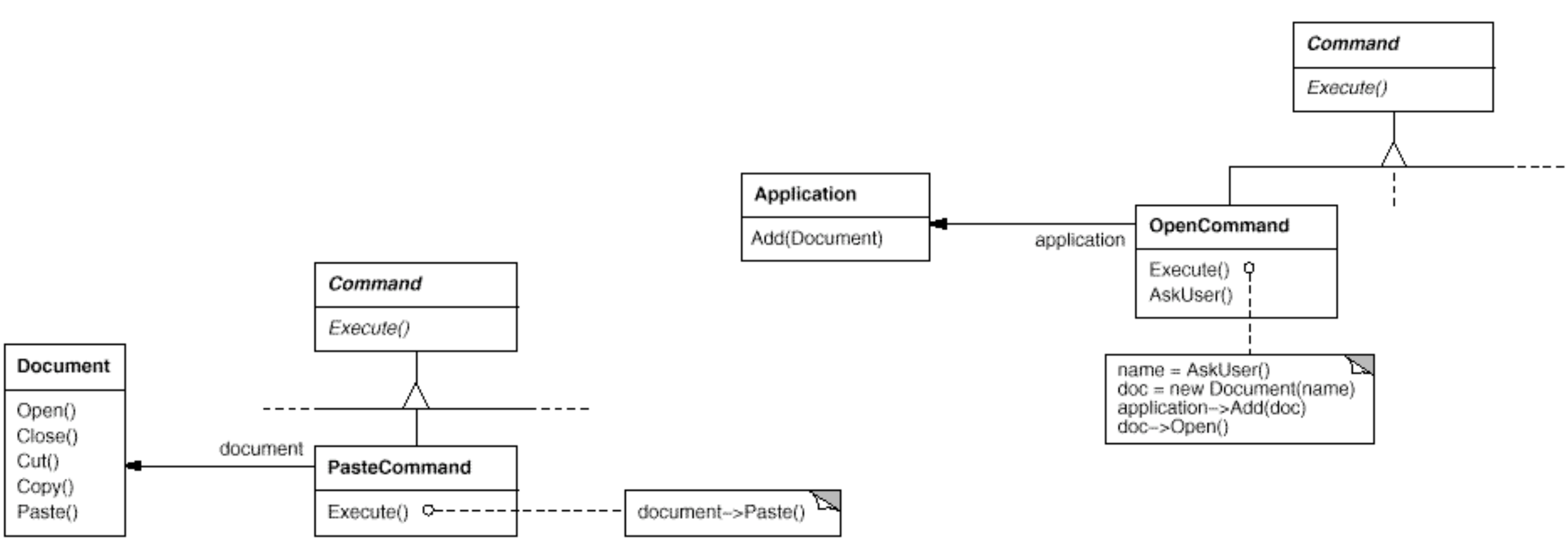



- **Encapsulate what varies**:
  need to handle behaviour as objects that can be assigned to keys or to buttons, etc.
  - the obvious responsibility of such a request object is to be executable. The next logical step is to require that it can "un-execute" itself in order to support undo.
- **Program to an interface**.
  The request objects must have a common interface to allow them to be exchanged across the user interface elements that must enact them.
- **Object composition.**
  Instead of buttons, menu items, key strokes hard coding behaviour, they delegate to their assigned **command** objects.

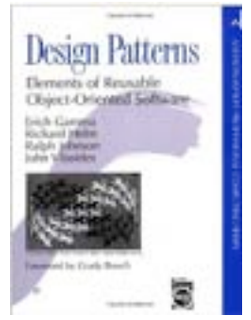
# Command

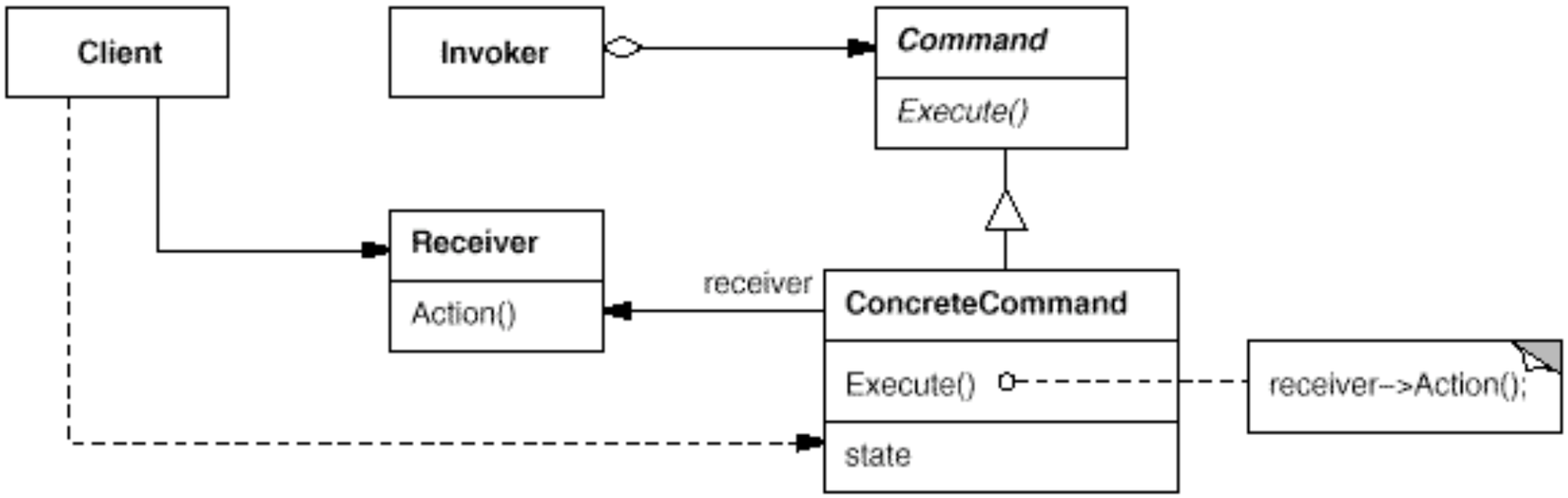



**Intent**: Encapsulate a request as an object so that
Clients can be parameterised with different requests
Requests can be queued and logged.
Undo and redo operations become possible.



# Command

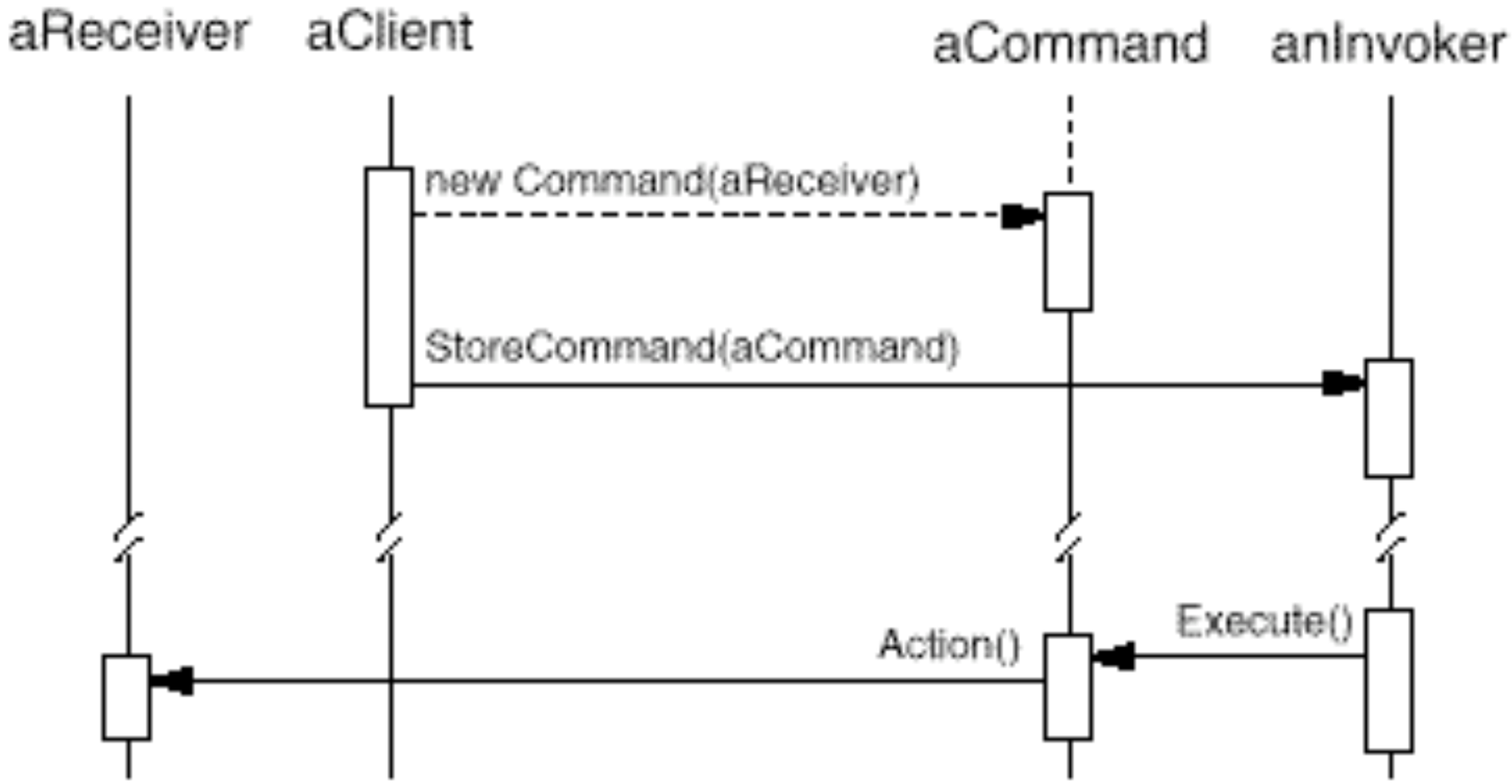


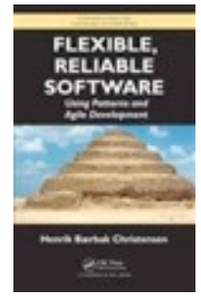

**Command** declares an interface for executing an operation. The **ConcreteCommand** defines a binding between a Receiver object and an action, it implements *Execute* by invoking the corresponding operation(s) on **Receiver**. The **Client** creates a **ConcreteCommand** object and sets its receiver. **Invoker** asks the command to carry out the request. The **Receiver** knows how to perform the operations associated with carrying out a request.

# Dynamics


Design Patterns

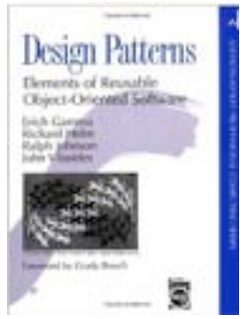

aReceiver
aClient
aCommand
anInvoker
new Command(aReceiver)
StoreCommand(aCommand)
Action()
Execute()

# Consequences




- Benefits
  - Decouples clients from set of commands
  - Command set can be extended at run-time
  - Easy to support multiple ways to execute command (menu, pop up, shortcut key, tool bar, …)
  - Commands are first-class objects
    - Log them, store them
  - Assembling macros is easy (composite of commands)
  - Undo can be supported
    - Add an 'unexecute()' method, and stack the set of executed commands.
- Liability: Cumbersome code for calling a method

# Implementation



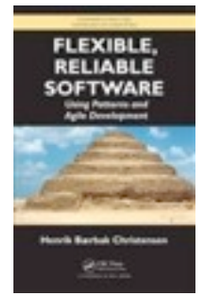

- How intelligent should a command be?
  - Merely define a binding between a receiver and the actions that carry out the request
  - Implement everything by itself without delegating to a receiver
  - Find the receiver dynamically
- Supporting undo and redo
  - Command must provide a way to revert the execution (Unexecute of Undo operation)
  - ConcreteCommand  class may need to store additional state to be able to do so
    - The Receiver object
    - The arguments to the operation(s) performed on the receiver
    - Any original state of the receiver that changes as a result of handling the request

# Implementation


- Supporting multiple level undo and redo
  - The application  needs to store a history list of commands
  - Commands must be copied before they are put on the history list when different invocations of the same command must be distinguished from each other

# Proxy

# Suppose that...



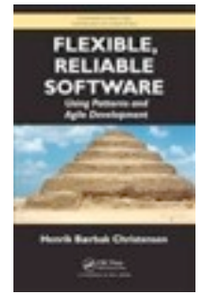

- A web page contains a lot of images, many of which will never be displayed as they are 'at the end of the scroll' where the average user does not look at all. Can we avoid downloading them?

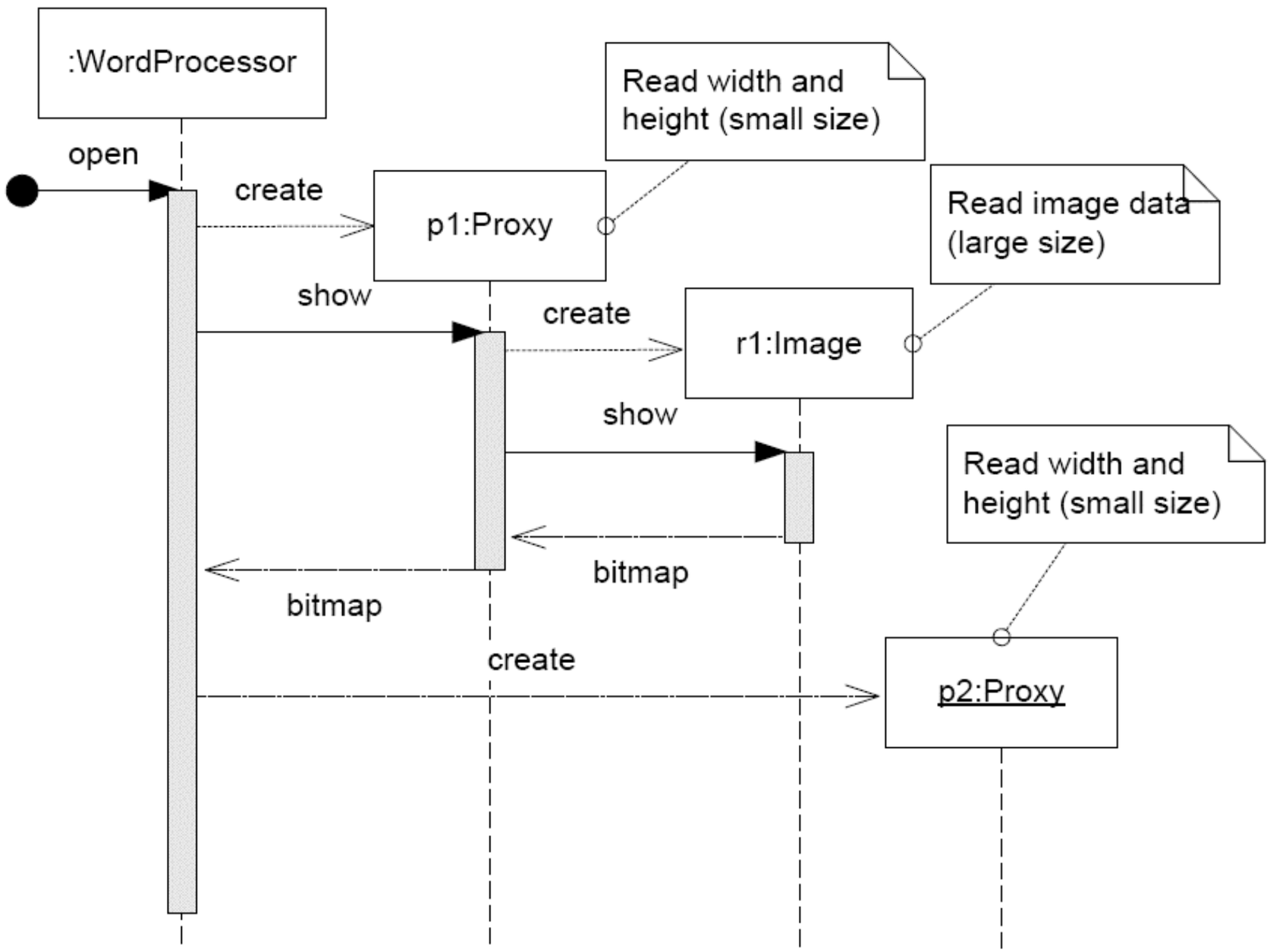
- *Download on demand* – i.e., only when they become visible.

# Compositional Approach

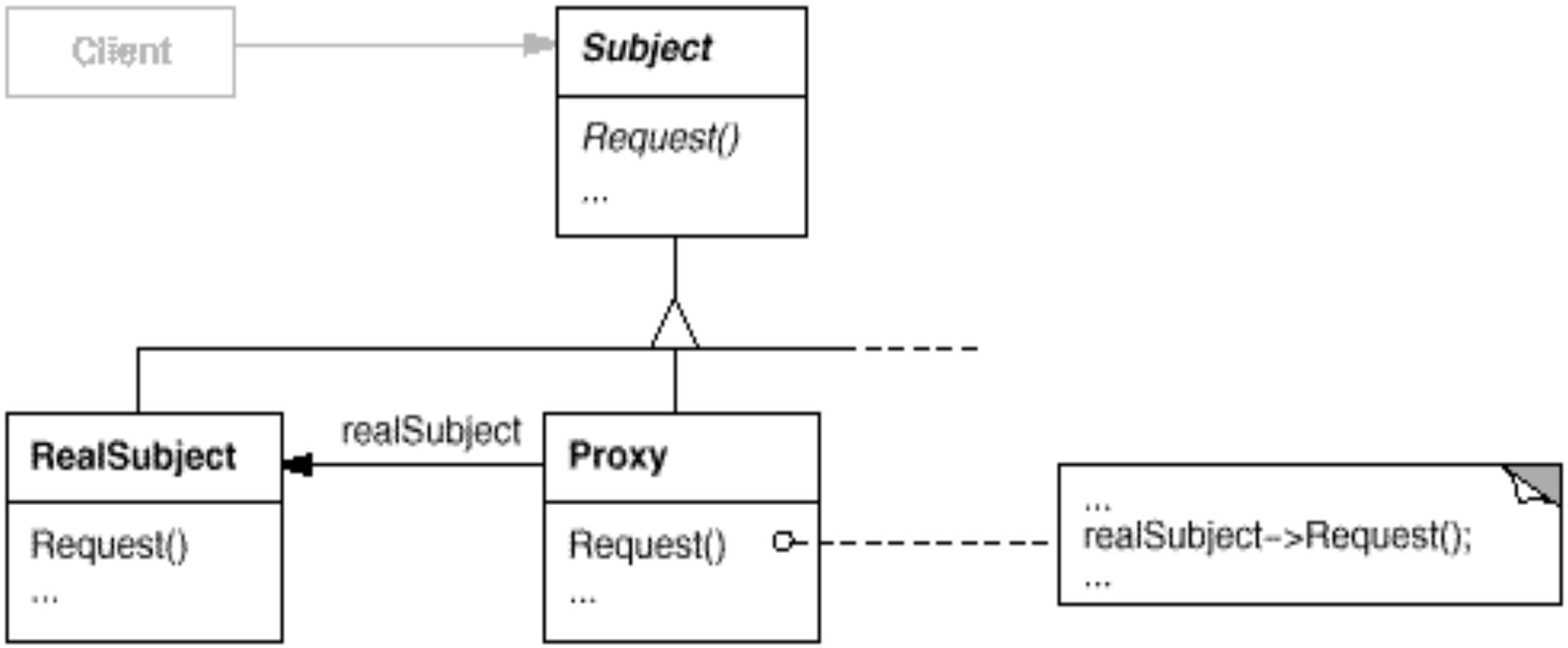

- **Encapsulate what varies:**
  - Seen from the client, the image objects should have variable behaviour. The ones that become visible will fetch image data, whereas those that have not yet been visible simply do not spend time loading the image data.
- **Program to an interface:**
  - Provide the client with an intermediate object that will defer loading until the "*show()*" method is called but in all other aspects act like a real image object.
- **Object composition:**
  - compose the real image behaviour by putting an "object in front". (that will only fetch the real image data once it needs to be shown)

# Dynamics


FLEXIBLE,
RELIABLE
SOFTWARE
Using Patterns and
Agile Development
Henrik Bærbak Christensen

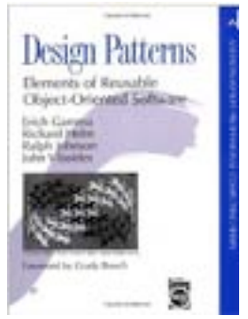

:WordProcessor
open
create
p1:Proxy
Read width and
height (small size)
Read image data
(large size)
show
create
r1:Image
show
Read width and
height (small size)
bitmap
bitmap
create
p2:Proxy

# **Intent**: Provide a surrogate or placeholder for another object to control access to it


The **Proxy** maintains a reference that lets the proxy access the real subject, it provides an interface identical to the **Subject's** so that a proxy can be substituted for the real subject and it controls access to the real subject and may be responsible for creating and deleting it.
The **Subject** defines a common interface for **Realsubject** and **Proxy** so that a **Proxy** can be used anywhere a **Realsubject** is expected.
**Realsubject** defines the real object that the proxy represents.

# Different kinds of Proxy




- a **remote proxy** provides a local representative for an object in a different address space
- a **virtual proxy** creates expensive objects on demand
- a **protection proxy** controls access to the original object and are useful when objects have different access rights
- a **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed: e.g., counting references, loading a persistent object when it is first referenced, locking the real object, ...

# Consequences

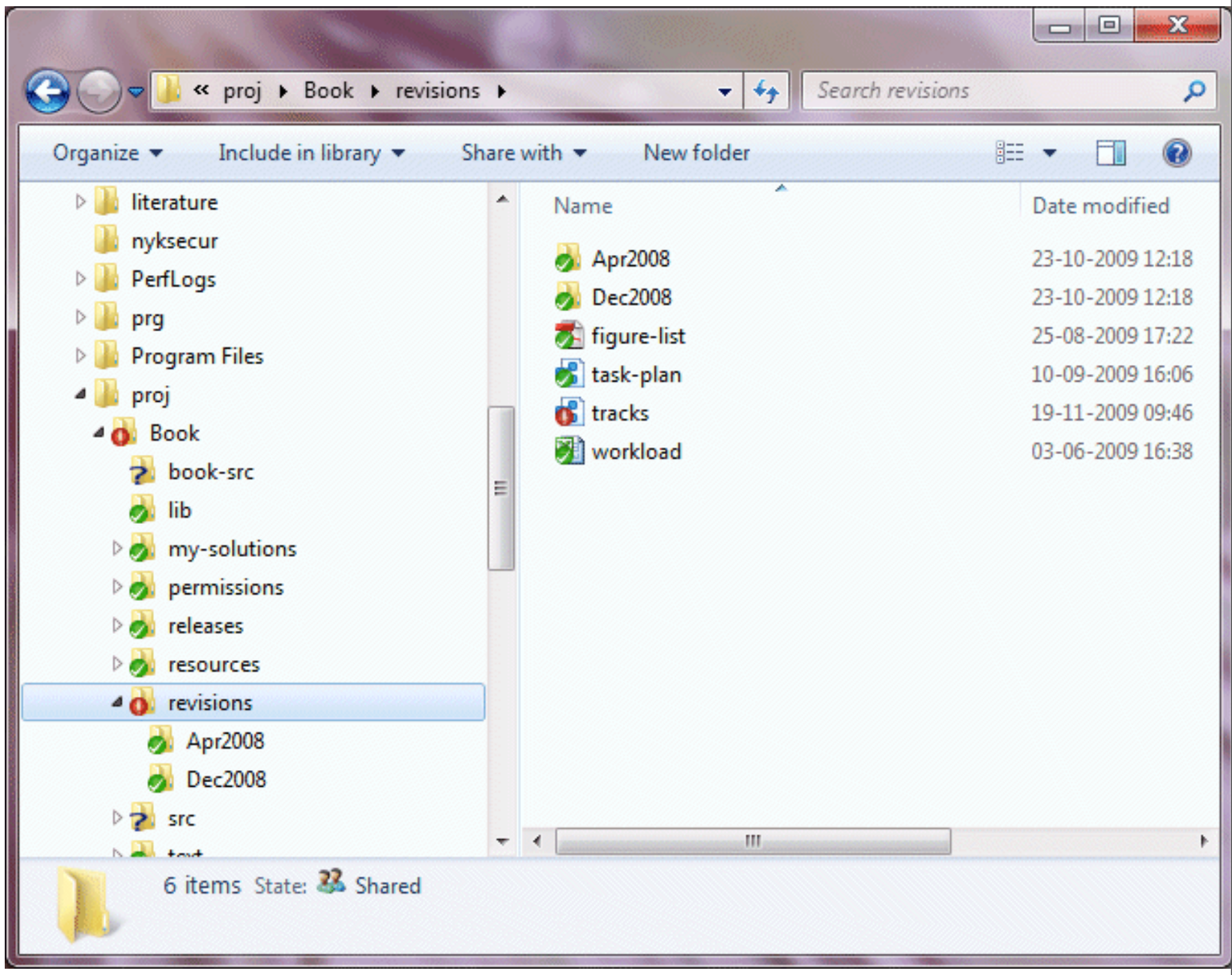

- + The Proxy pattern introduces a level of indirection when accessing an object.
  This indirection has many uses:
  - A remote proxy can hide the fact that the object resides in a different address space
  - A virtual proxy can perform optimisations
  - Both protection proxies and smart pointers allow additional housekeeping
- + The proxy patterns can be used to implement "copy-on-write"
  to avoid unnecessary copying of large objects the real subject is referenced counted; each copy request increments this counter but only when a client requests an operation that modifies the subject the proxy actually copies it.

# Composite

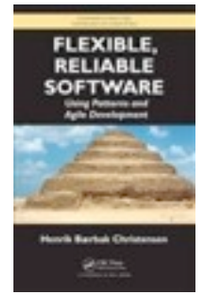
# Part-whole Structures


Hierarchical data structures pervade IT systems

Folders (whole) and files (part) is a classic example

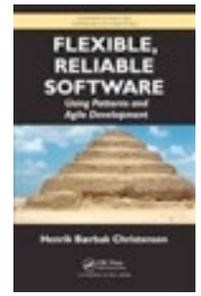


# How to design?




- Using the **model perspective** (who/what) we focus on concepts in the domain:
  - who: Folder and File
  - what: different things
    - Folder: addFile, addFolder, removeFile, etc.
    - File: open, close, getType, getSize, setReadOnly
- Using a **responsibility perspective** (what/who) we instead focus on behaviour:
  - what: calculate size, move in structure, delete, set to read only
  - who: actually both folders and files…

# Design 1

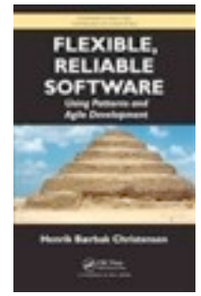



- Make disjoint classes as they are disjoint concepts
  - class Folder {…} and class File {…}

- But – will require a lot of casting…

```
private static void displaySize(Object item) {
  if (item instanceof File) {
    File file = (File) item;
    System.out.println( "File size is "+file.size() );
  } else if (item instanceof Folder) {
    Folder folder = (Folder) item;
    System.out.println( "Folder size is "+folder.size() );
  }
}
```

# Design 2




## Program to an interface

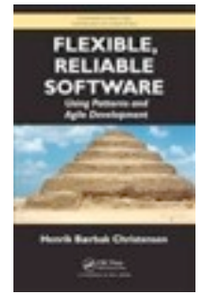
Fragment: chapter/composite/CompositeDemo.java

```
/** Define the Component interface
 * (partial for a folder hierarchy) */
interface Component {
  public void addComponent(Component sibling);
  public int size();
}
```

Fragment: chapter/composite/CompositeDemo.java

```
/** Define a (partial) folder abstraction */
class Folder implements Component {
  private List<Component> components = new ArrayList<Component>();
  public void addComponent(Component sibling) {
    components.add(sibling);
  }
  public int size() {
    int size = 0;
    for ( Component c: components ) {
      size += c.size();
    }
    return size;
  }
}
```

# Design 2 (c'td)

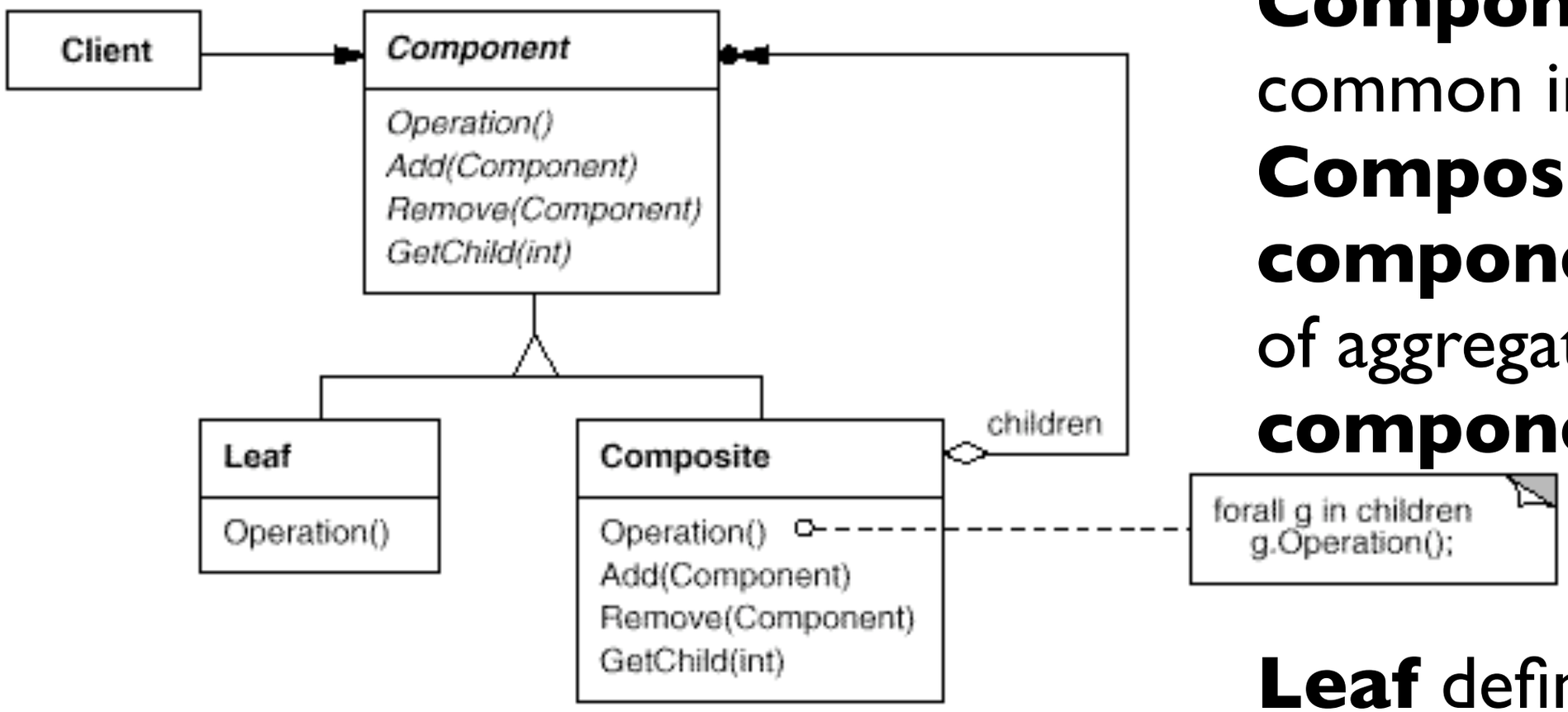



Notice that this is a recursive depth-first descent into the tree…

Fragment: chapter/composite/CompositeDemo.java

```
/** Define a (partial) folder abstraction */
class Folder implements Component {
  private List<Component> components = new ArrayList<Component>();
  public void addComponent(Component sibling) {
    components.add(sibling);
  }
  public int size() {
    int size = 0;
    for ( Component c: components ) {
      size += c.size();
    }
    return size;
  }
}
```

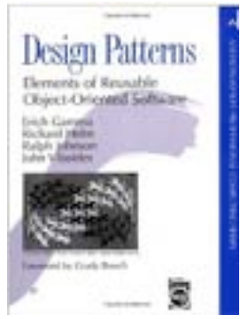
**Intent**: Compose objects into tree-like structures to represent part-whole hierarchies and let clients treat individual objects and compositions of objects uniformly




**Component** defines a common interface.
**Composite** defines a **component** by means of aggregating other **components**.

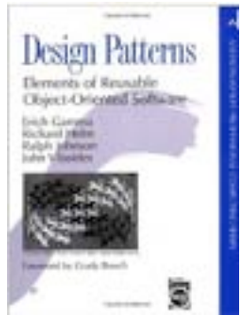

**Leaf** defines a primitive, atomic, component, i.e., one that has no substructure.

# Consequences



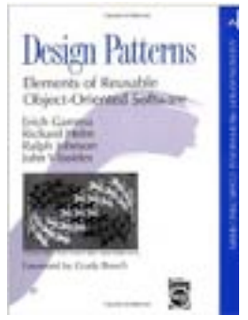

- **+ Makes the Client simple:**
  clients can treat composite structures and individual objects uniformly,
  clients normally don't know and should not care whether they are dealing with a leaf or a composite
- **+ Makes it easier to add new types of components:**
  client code works automatically with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

# Implementation I




- **Explicit parent references**:
  simplify traversal and management of composite structures; are best defined in the Component class;
  it is essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children
- **Sharing components**:
  can be useful for example to reduce storage requirements but can lead to ambiguities when requests propagate up the structure
- **Maximising the Component interface**:
  to achieve that clients are unaware of the specific Leaf or Composite class they are using, the Component class should define as  many common operations on Leaf and Composite classes as possible;
  this conflicts with class hierarchy design that says that a class should only implement operations that make sense on all of its subclasses; some creativity is needed to find good default implementations for operations (ex. children (leaf) = {})

# Implementation 2




- **The child access and management operations**: defining the child management interface at the Component level gives transparency but costs safety, clients can do meaningless things like add to or delete from a leaf; this must be captured in a default implementation: do nothing or throw error ???
- **The instance variable(s) holding the children**: commonly this instance variable belongs where the access and management operations are but when put with the Component class a space penalty is involved since Leafs never have children.
- **Deleting Components:**
  in languages without garbage collection it is best to make Composite responsible for deleting its children; an exception is when Leaf objects are immutable and thus can be shared.

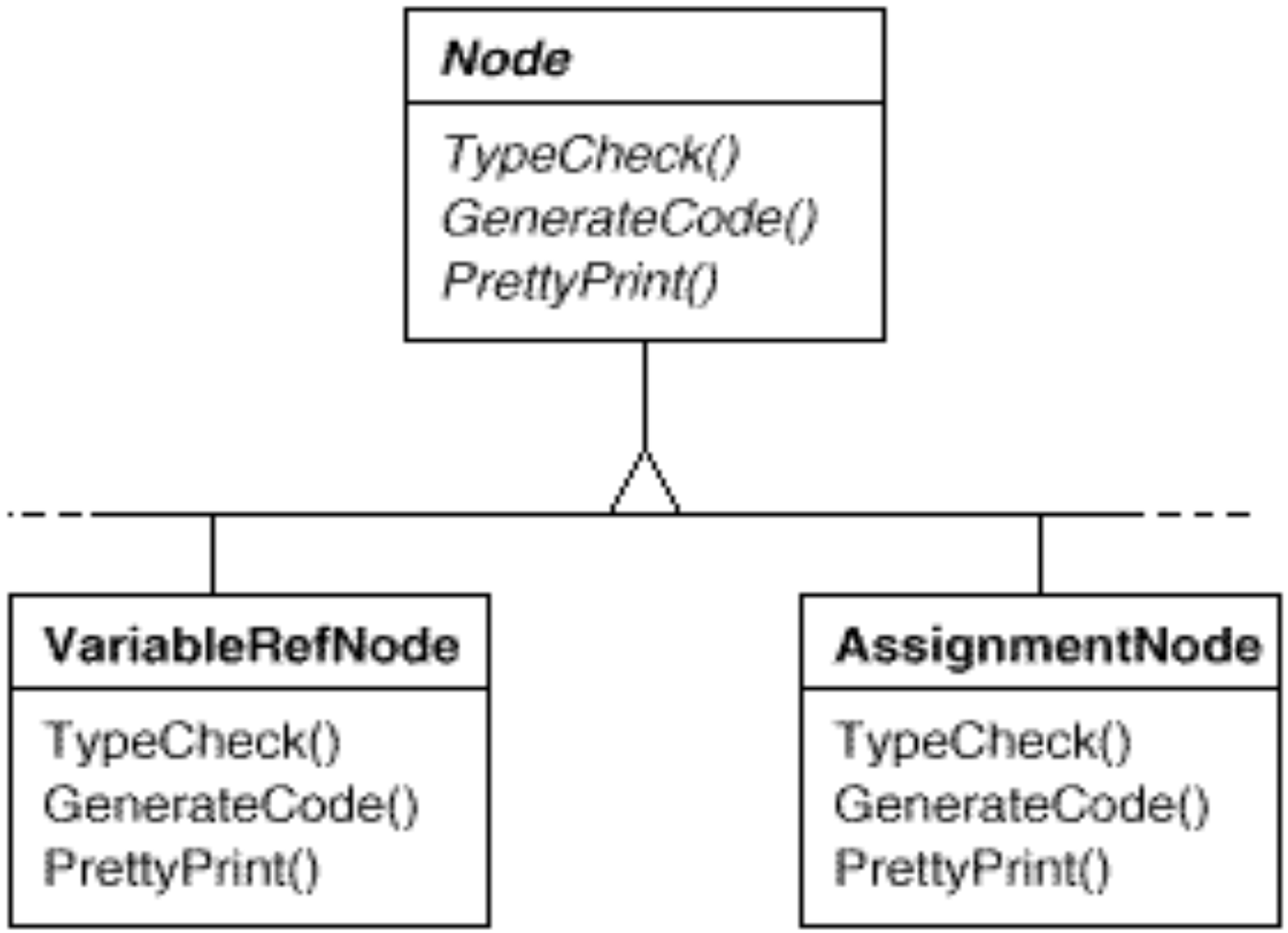
# Implementation 3


- **The datastructure for storing children**:
  a variety of datastructures is available: lists, trees, arrays, hash tables; the choice depends on efficiency; alternatively each child can be kept in a separate instance variable; all access and management operations must then be implemented on each Composite subclass
- **Child ordering**:
  when child order is an issue the child access and management interface must be designed carefully to manage this sequence.
- **Caching:**
  when compositions need to be traversed or searched frequently the Composite class can cache traversal or search information on its children; changes to a component will require invalidating the caches of its parent.

# Visitor

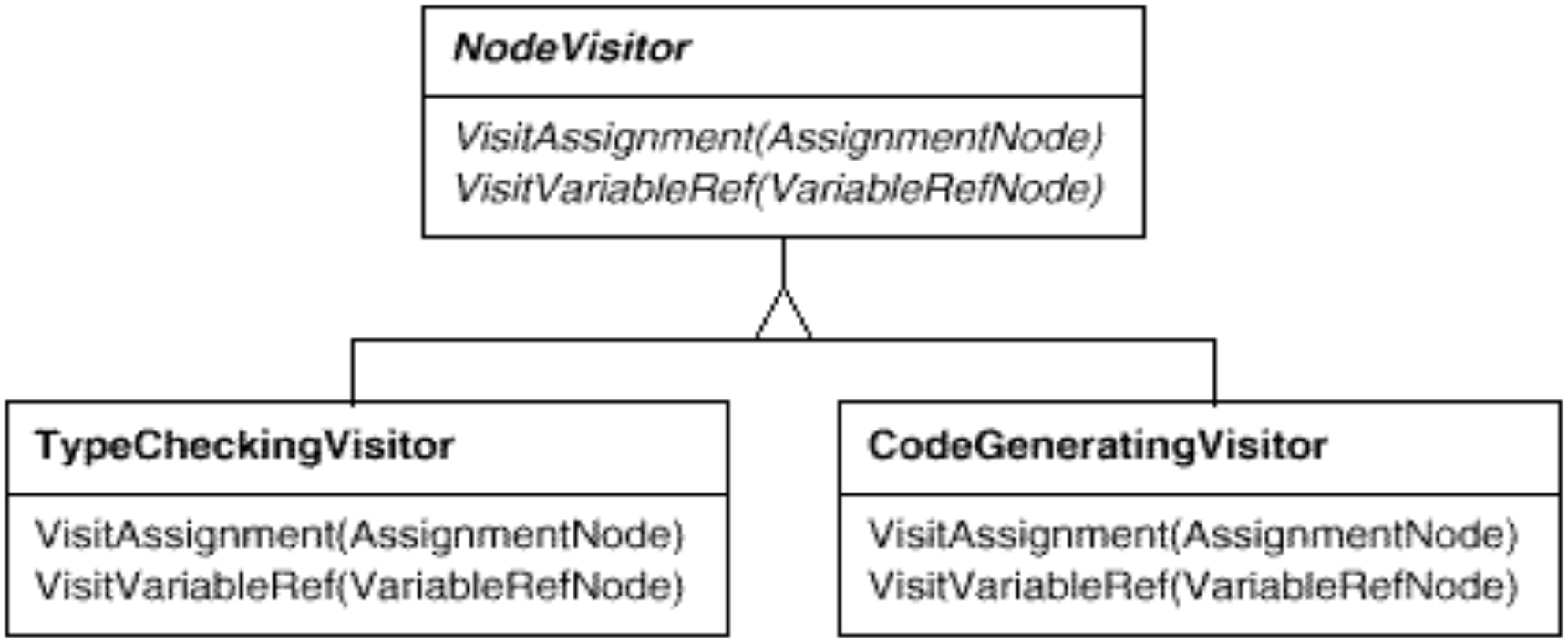
# Example

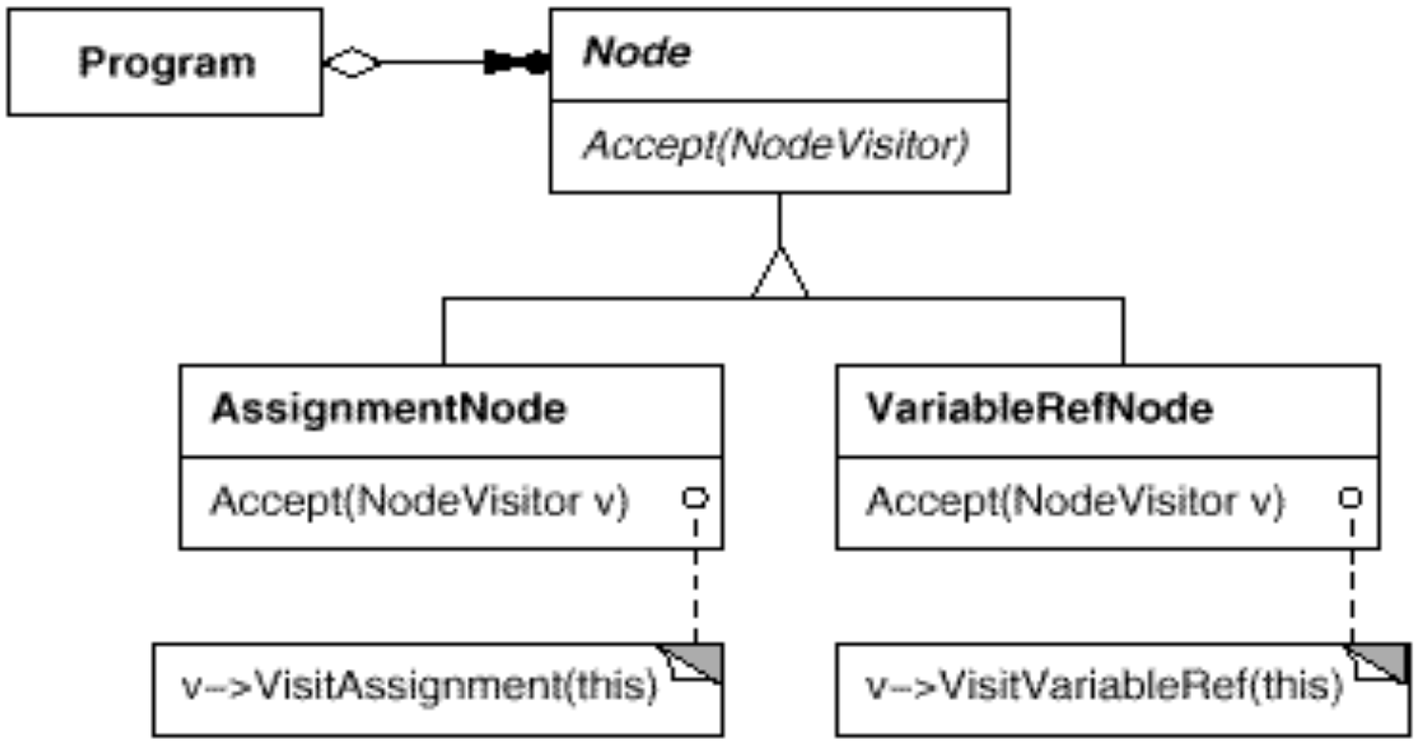

Consider a compiler representing programs as abstract syntax trees. Operations like type checking, generating code, flow analysis, etc. need to be performed.



# Solution

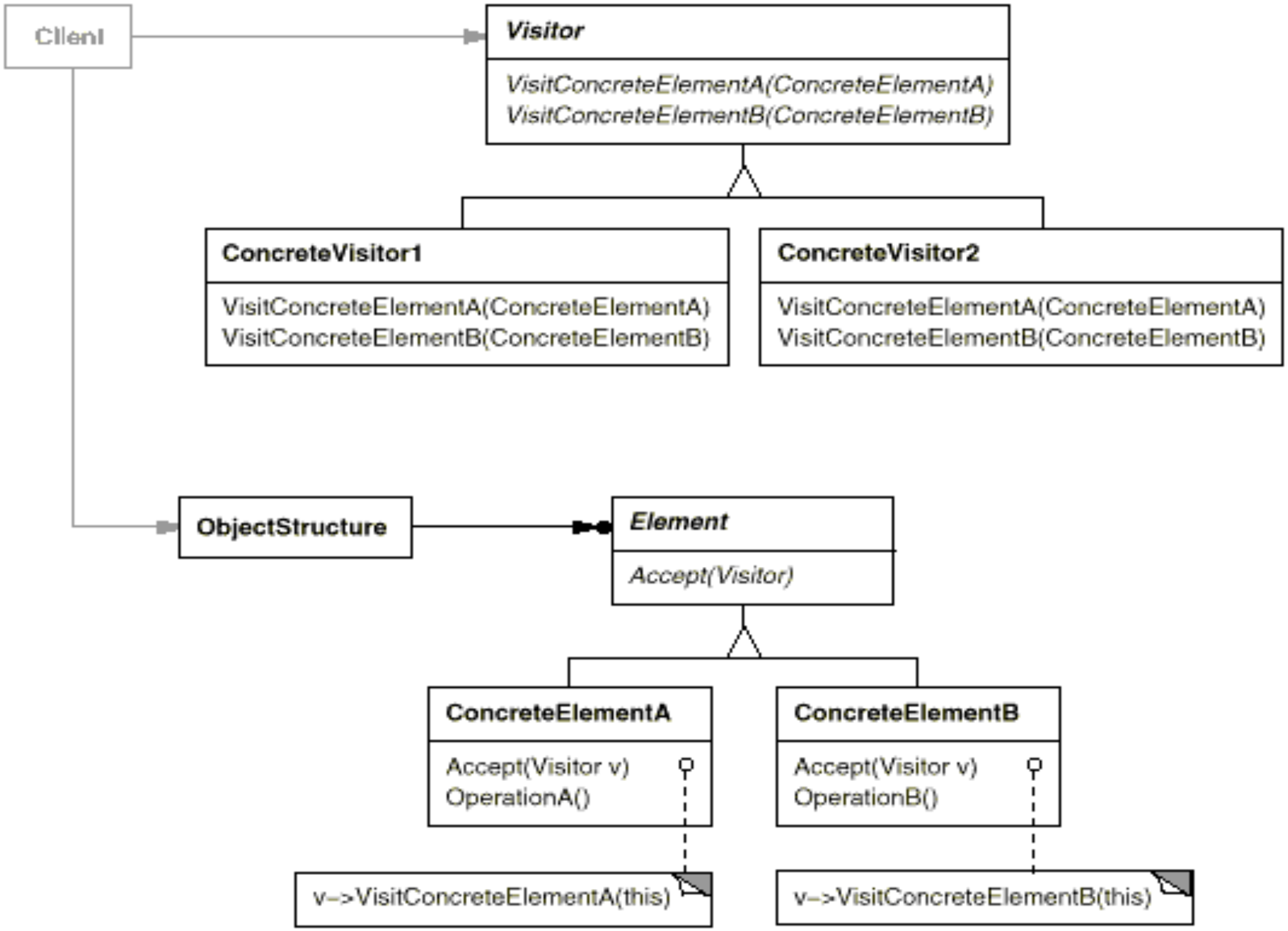

Design Patterns


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)


Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

# Structure


Design Patterns


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

**Intent**: represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates


**Context**:
declares a Visit operation for each class of ConcreteElement in the object structure.
The operations name and signature identified the class that sends the Visit request.

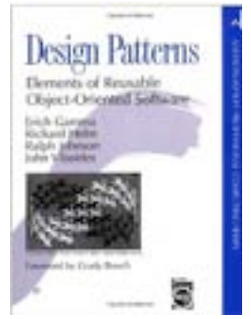
**ConcreteVisitor:**
implements each operation declared by Visitor. Each operation implements a fragment of the algorithm for the corresponding class of object in the object structure. It provides the context for the algorithm and stores its state (often accumulating results during traversal).

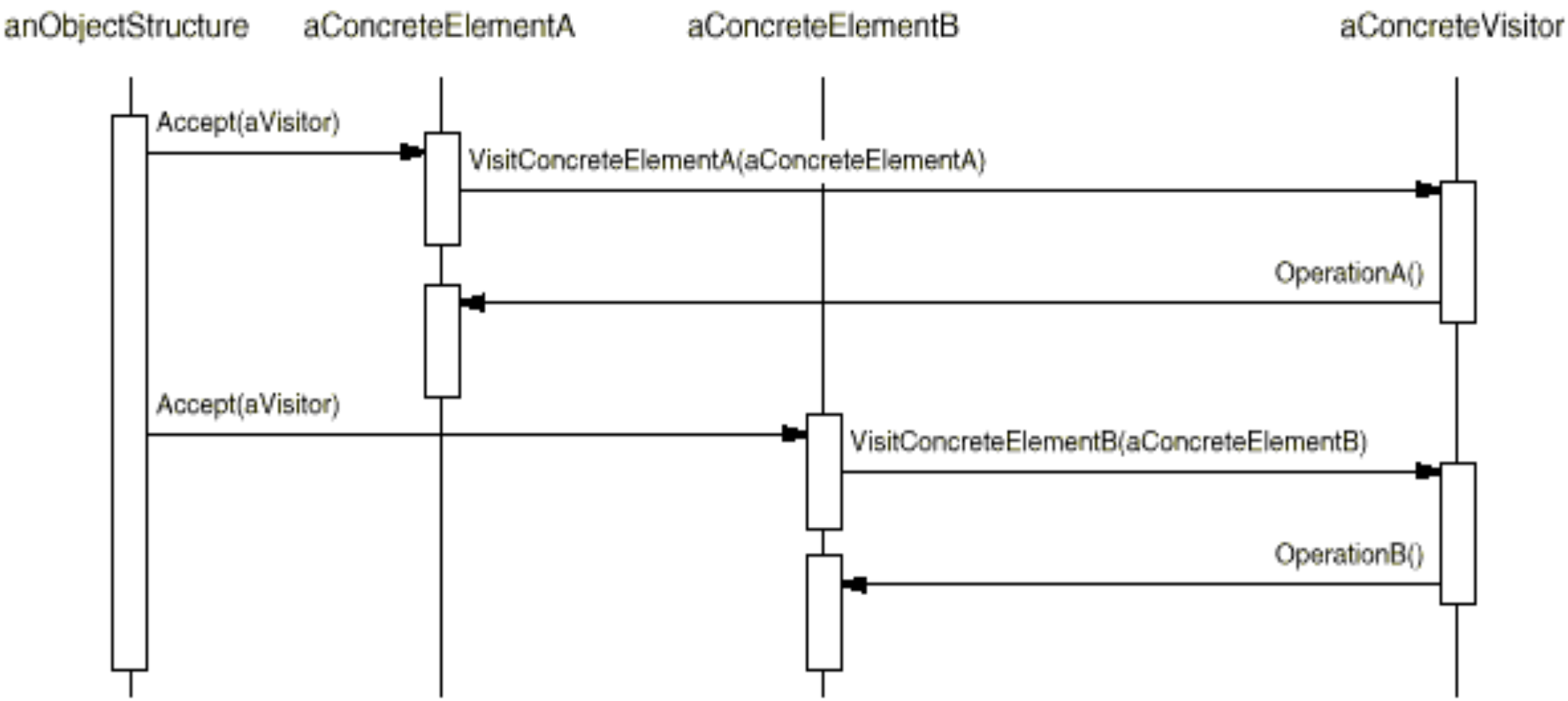
**Element**:
defines an accept operation that takes a visitor as an argument.

**ConcreteElement**:
implements an accept operation that takes a visitor as an argument

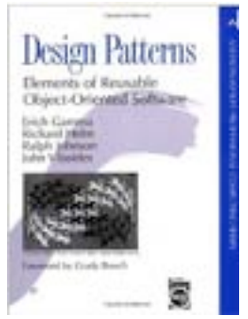
**ObjectStructure**:
can enumerate its elements
may provide a high-level interface to allow the visitor to visit its elements
may either be a Composite or a collection such as a list or a set

# Dynamics


Design Patterns


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

# Consequences 1

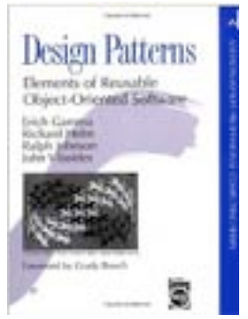



- **+ Makes adding new operations easy:**
  a new operation is defined by adding a new visitor (in contrast, when you spread functionality over many classes each class must be changed to define the new operation)
- **+ Gathers related operations and separates unrelated ones:**
  related behaviour is localised in the visitor and not spread over the classes defining the object structure
- **- Adding new ConcreteElement classes is hard:**
  each new ConcreteElement gives rise to a new abstract operation in Visitor and a corresponding implementation in each ConcreteVisitor

# Consequences 2


- **+ Allows visiting across class hierarchies:**
  an iterator can also visit the elements of an object structure as it traverses them and calls operations on them  but all elements of the object structure then need to have a common parent. Visitor does not have this restriction.
- **+ Accumulating state:**
  visitor can accumulate state as it proceeds with the traversal. Without a visitor this state must be passed as an extra parameter of handled in global variables
- **- Breaking encapsulation:**
  Visitor's approach assumes that the ConcreteElement interface is powerful enough to allow the visitors to do their job. As a result the pattern ofthen forces to provide public operations that access an element's internal state which may compromise its encapsulation

# Implementation




- **Who is responsible for traversing the object structure?**
  responsibility for traversal can be with:
  - The object structure
  - The visitor:
    is advisable when a particular complex traversal is needed (for example one that depends on the outcome of the operation), otherwise it I not advisable because a lot of traversal code will be duplicated in each ConcreteVisitor for each aggregate ConcreteElement
  - A separate iterator object
- **Double Dispatch.**
  The key to visitor is a double dispatch: the meaning of the accept operation depends on the visitor and on the element. Languages that support double dispatch (CLOS) can do without this pattern.

# Example double dispatch

```
; UI that draws on a window using the Canvas.rkt library
(define (make-canvas-ui)
  (let ((window-w 800)
        (window-h 600)
        (black (make-color 0 0 0))
        (green (make-color 0 15 0)))

    ;clears the window by painting it black
    (define (clear)
      (fill-rectangle! 0 0 window-w window-h black))

    ;draws the given ball in green
    (define (draw-ball ball)
      (let ((position (send-message ball 'position))
            (radius (send-message ball 'radius)))
        (fill-ellipse! (coordinates-x position)
                       (coordinates-y position)
                       radius
                       radius
                       green))))

(define (dispatch message)
  (case message
    ((clear) clear)
    ((draw-ball) draw-ball)
    (else (error 'canvas-ui "unknown message ~a" message))))
                                    dispatch))
```

```
; Instantiates a ball with the given radius and x-coordinate
(define (make-ball radius x)
  (let ((dy 10) ;change in vertical position
        (position (make-coordinates x 100))) ;initial position

    (define (get-position) position)
    (define (get-radius) radius)

    ;Adjusts the ball's position upwards
    (define (up!)
      (coordinates-y! position
                      (+ (coordinates-y position) dy)))

    ;Adjusts the ball's position downwards
    (define (down!)
      (coordinates-y! position
                      (- (coordinates-y position) dy)))

;Draws the ball on the given game UI
(define (draw ui)
  ;does not draw directly, but asks the UI to draw the ball instead
  ;this way, the game can be configured with a different UI
  (send-message ui 'draw-ball dispatch))

(define (dispatch message)
  (case message
    ((position) get-position)
    ((radius) get-radius)
    ((draw) draw)
    ((up!) up!)
    ((down!) down!)
    (else (error 'ball "unknown message ~a" message))))
dispatch))
```

# Observer

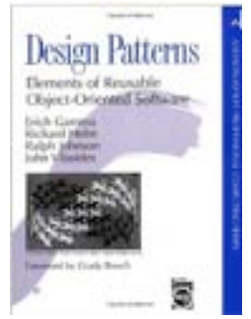
# Challenges

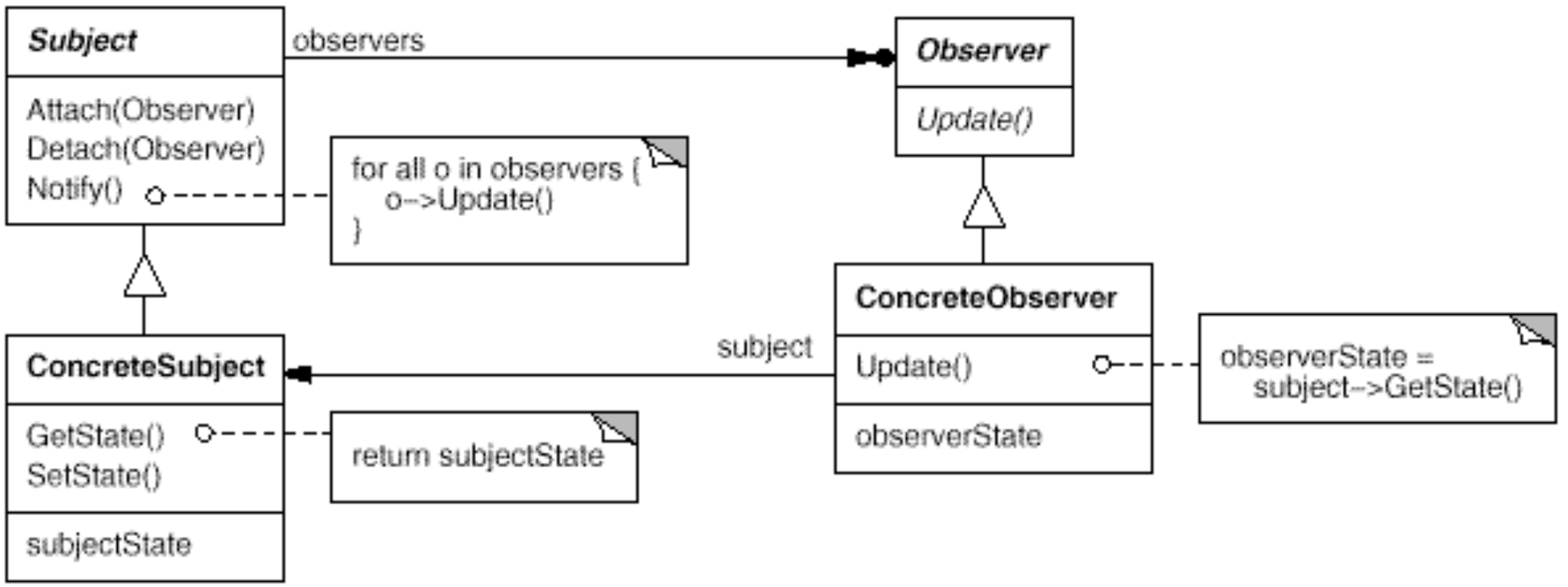

- Objects dependent on some underlying core information:
  if this information changes then the objects must react accordingly.
- Examples:
  - spreadsheet
    the data in the sheet may be simultaneously displayed as bar charts and pie charts in their own windows. => want these windows to be synchronized
  - calendar application
    shows current time both as text in a status field and as a highlight of entries in the day view.
    As time passes both need to be updated.
  - UML diagram editor
    the object representing an association line between two classes must monitor any movement of either of the connected classes in order to reposition and redraw.

# Compositional Approach


- Take as an example, the spreadsheet:
  - consider what should be variable in your design:
    Data is shared but visualisation is variable!
    variable behaviour is the processing that must take place each time the cell changes value.
  - program to an interface:
    define an interface that encapsulates the processing responsibility.
  - favour object composition

# Observer

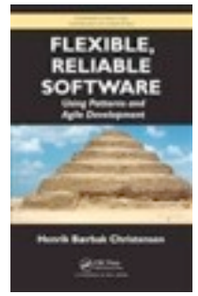


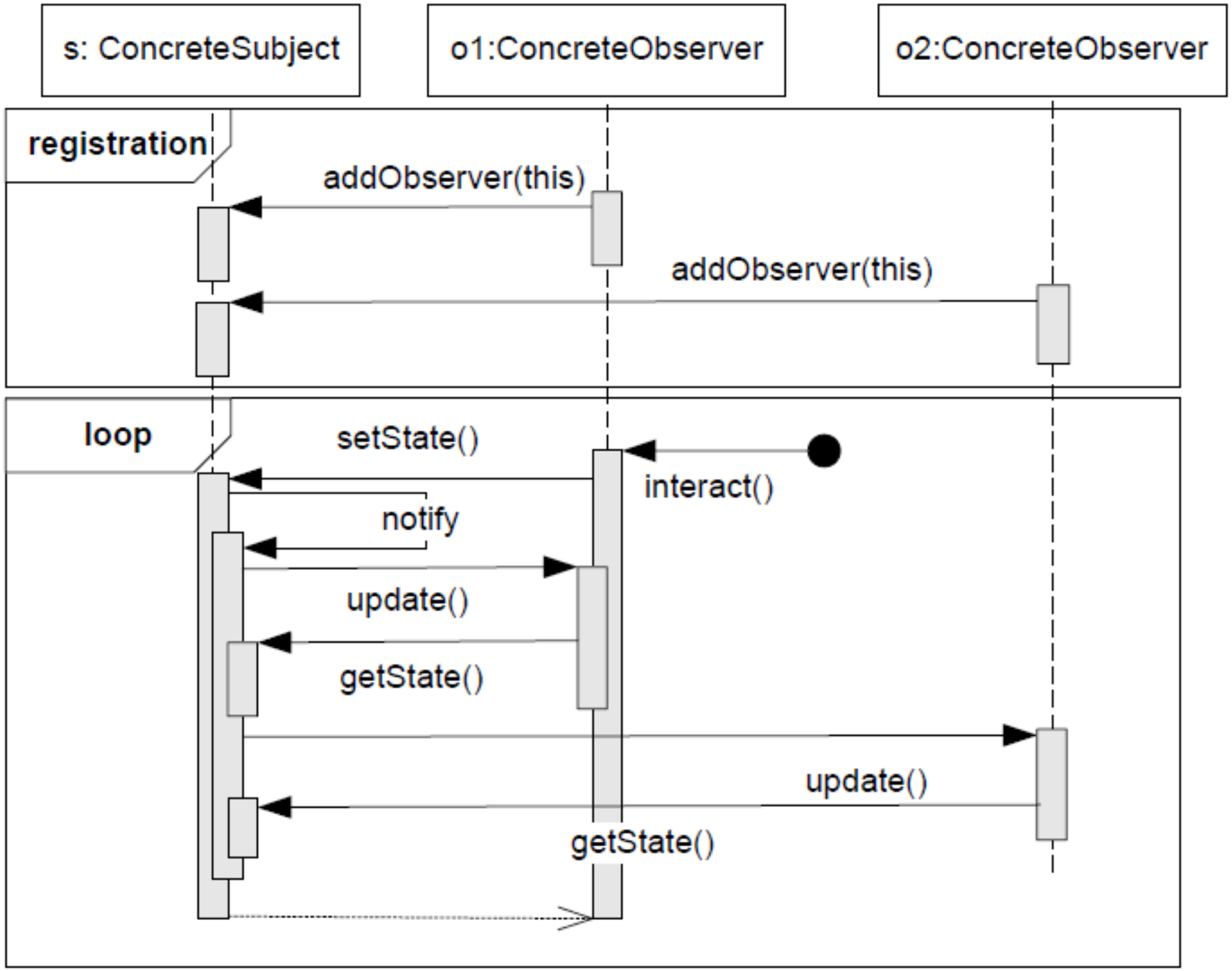




**Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

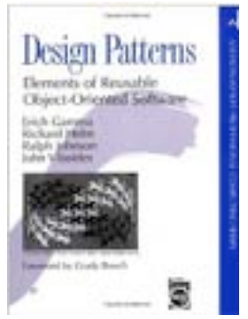
**Observer** specifies the responsibility and interface for being able to be notified.
**Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the update method on all observers in its list.
**ConcreteObserver** defines concrete behaviour for how to react when the **subject** experiences a state change.

# Protocol


FLEXIBLE,
RELIABLE
SOFTWARE

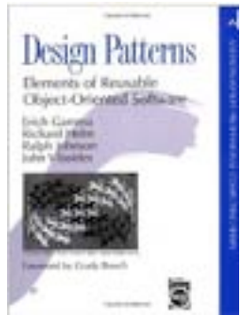

s: ConcreteSubject
o1:ConcreteObserver
o2:ConcreteObserver
registration
addObserver(this)
addObserver(this)
loop
setState()
interact()
notify
update()
getState()
update()
getState()

# Consequences




- **+ Abstract and minimal coupling between Subject and Observer:**
  the subject does not know the concrete class of any observer, concrete subject and concrete observer classes can be reused independently, subject and observer can even belong to different abstraction layers in the system
- **+ Support for broadcast communication:**
  the notification a subject sends does not need to specify a receiver, it will broadcast to all interested (subscribed) parties
- **- Unexpected updates:**
  observers don't have knowledge about each others presence, a small operation might cause a cascade of spurious updates

# Implementation I



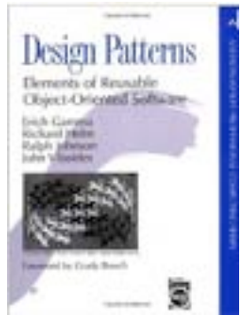

- **Mapping subjects to their observer:**
  the Subject keeps explicit references to the Observers it should notify or some associative lookup (e.g. a hash table) is installed; memory/time trade off must be made
- **Observing more than one Subject:**
  can make sense in some situations; the update interface must be extended to keep track of the Subject that is sending the update allowing the Observer to know which Subject to examine
- **Who triggers the updates (i.e. who calls notify):**
  - have all state changing operations on Subject call notify after the subject's state is changed; consecutive operations cause several consecutive updates which may not be necessary and is inefficient
  - make clients responsible for calling notify at the right time; clients get the added responsibility to call notify which makes errors likely

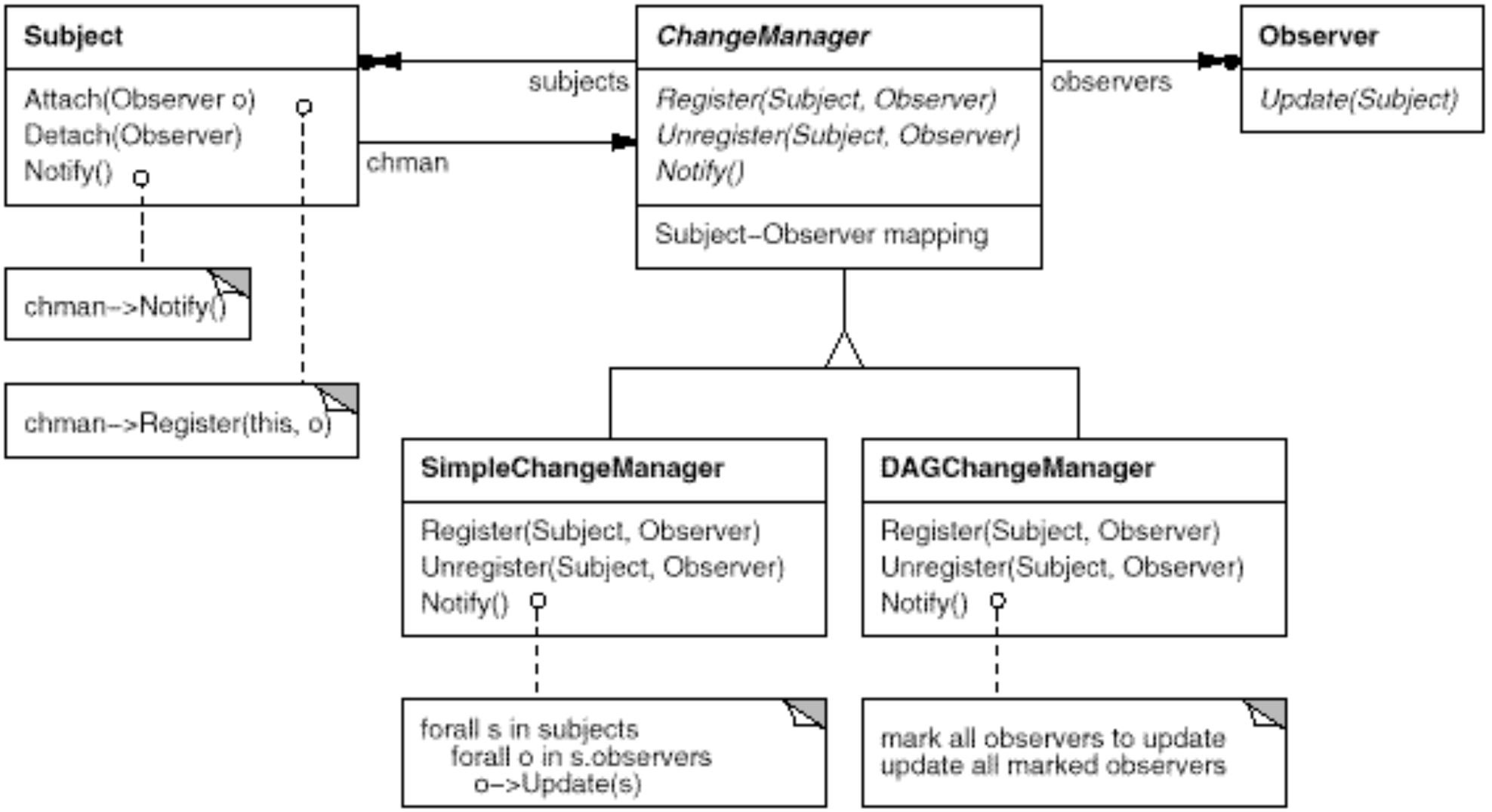
# Implementation 2


- **Dangling references to deleted Subjects:**
  deleting a Subject should not produce dangling references in its Observers; simply deleting the Observers is often not the best idea (they can be referenced by others or they can be observing another Subject); when deleting a Subject the Observers should be notified so that they can reset their Subject reference
- **Make sure that Subject is self consistent before calling notify:**
  Observers query the subject's state to do their update ; this rule is easy to violate unintentionally when Subject subclass operations call inherited operations
- **Specifying modification of interest explicitly:** update efficiency can be improved when the observers register for specific events of interest only

# Implementation 3


- **Avoiding observer-specific update protocols, the pull and the push model:**
  - In the push model the Subject sends its Observers detailed information on what is changed; the Subject class makes assumptions about the Observers' needs
  - In the pull model the Subject sends nothing but the most minimal notification and the Observers ask for details explicitly; emphasizes the Subject's ignorance of its Observers; can be inefficient because Observers must assess what is changed without help
- **Encapsulate complex update semantics:**
  install an explicit change manager that manages the Subject-Observer relationships and defines a particular update strategy


Design Patterns


Subject
Attach(Observer o)
Detach(Observer)
Notify()
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject–Observer mapping
observers
Observer
Update(Subject)
chman->Notify()
chman->Register(this, o)
SimpleChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
forall s in subjects
forall o in s.observers
o->Update(s)
DAGChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
mark all observers to update
update all marked observers