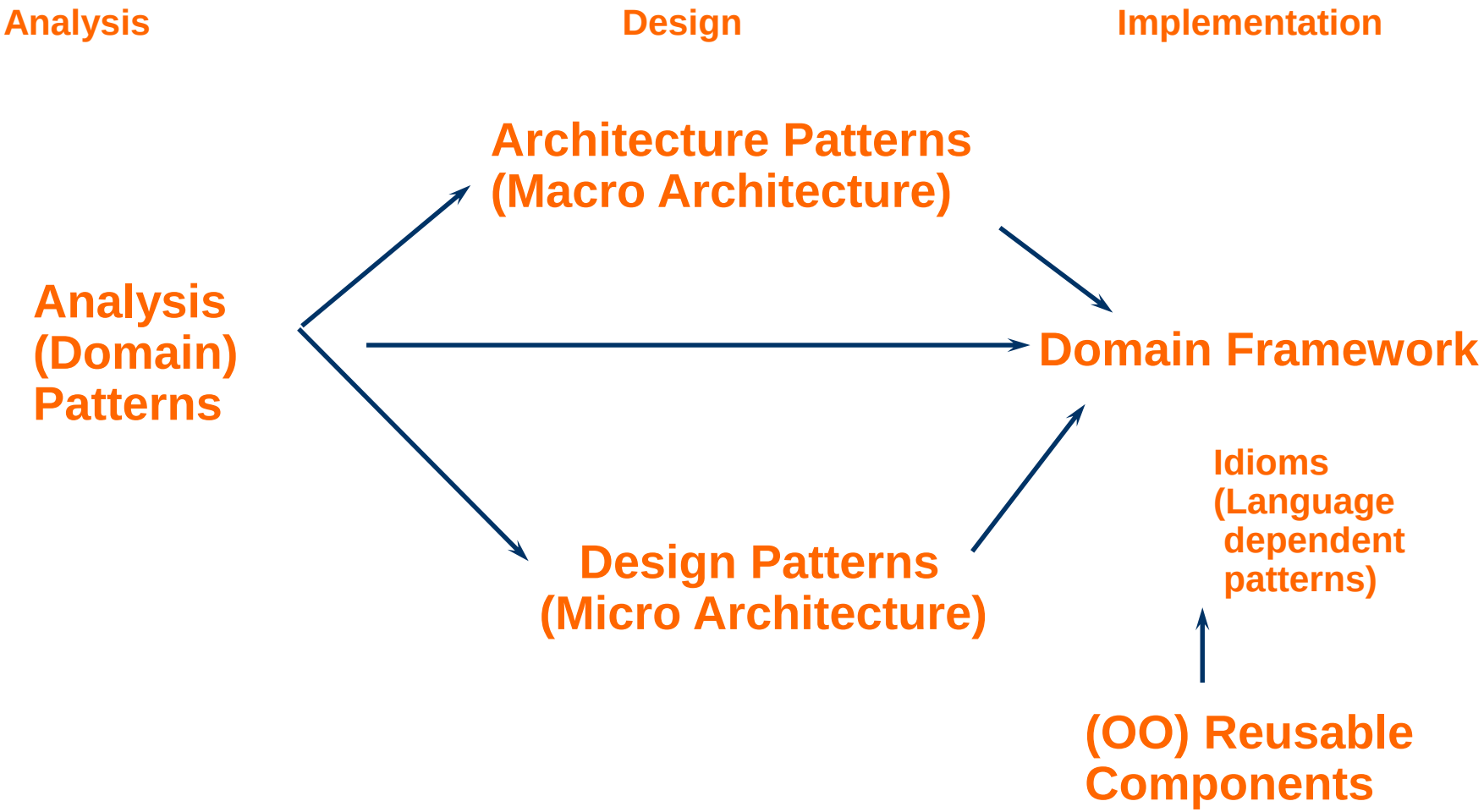
INF5120 – Modellbasert Systemutvikling

- **F07-2: Architectural Patterns, Design Patterns and Refactoring**

Lecture 27.02.2017

Arne-Jørgen Berre

Patterns: From Analysis to Implementation



Patterns on various design levels

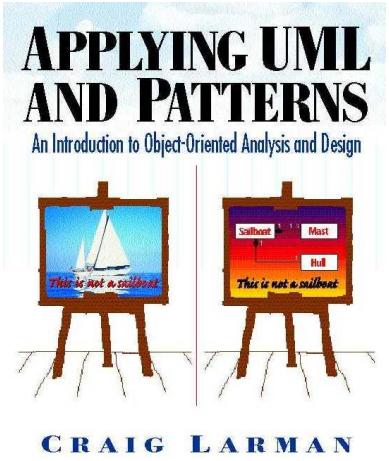
Module level patterns: Architecture Patterns

Collaboration level patterns: Design Patterns

Object level patterns: GRASP

Refactoring

GRASP



General Responsibility Assignment Software Patterns.

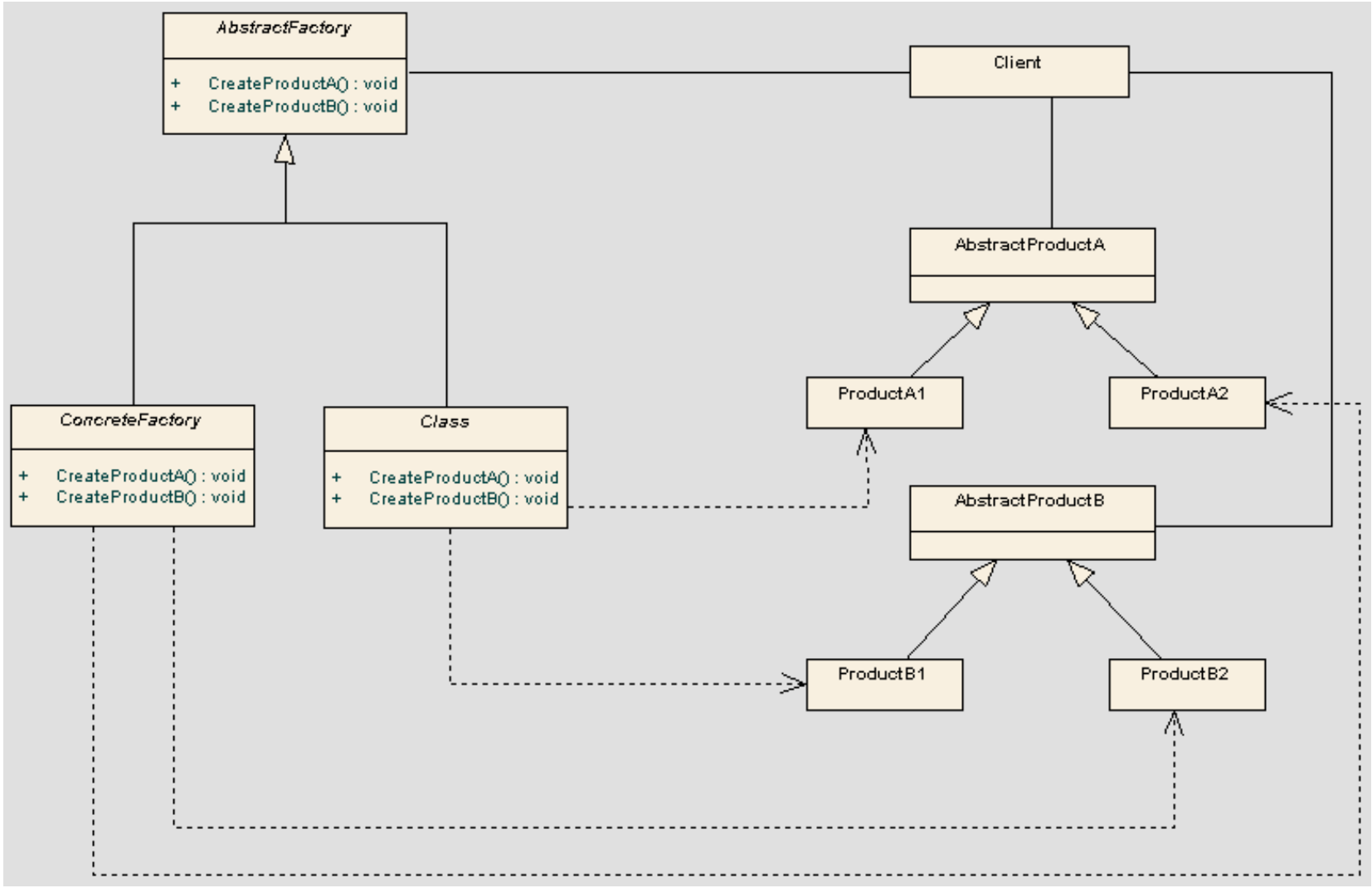
Responsibility assignment.

1. knowing (answering)
2. or, doing

Guidance and evaluation in mechanistic design.

1. **Expert**
2. **Creator**
3. **Controller**
4. **Low Coupling**
5. **High Cohesion**
6. **Polymorphism**
7. **Pure Fabrication**
8. **Indirection**
9. **Don't Talk to Strangers**

Patterns – Abstract Factory



Design patterns
(with UML & Java examples)

Based on:
Gamma/Helm/Johnson/Vlissides (GoF):
Design Patterns, 1995

R. Ryan:, D. Rosenstrauch:
Design Patterns in Java, 1997

What are patterns?

- "A solution to a problem in a context"?
  - Insufficient, says the “Gang of Four” (GOF)
  - What’s missing? 3 things:
    - Recurrence
    - Teaching (e.g., implementation consequences, trade-offs, and variations)
    - A name
- GOF:
  - Patterns contain 4 essential elements
    - pattern name
    - problem
    - solution
    - consequences
- Christopher Alexander (as quoted in the GOF book):
  - "Each pattern describes a problem which occurs over and over again ... and then describes the core of [a] solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Design Pattern

A design pattern describes a basic scheme for structuring subsystems and components of a software architecture as well as their relationships. It identifies, names, and abstracts a common structural or functional principle by describing its different parts, their collaboration and responsibilities.

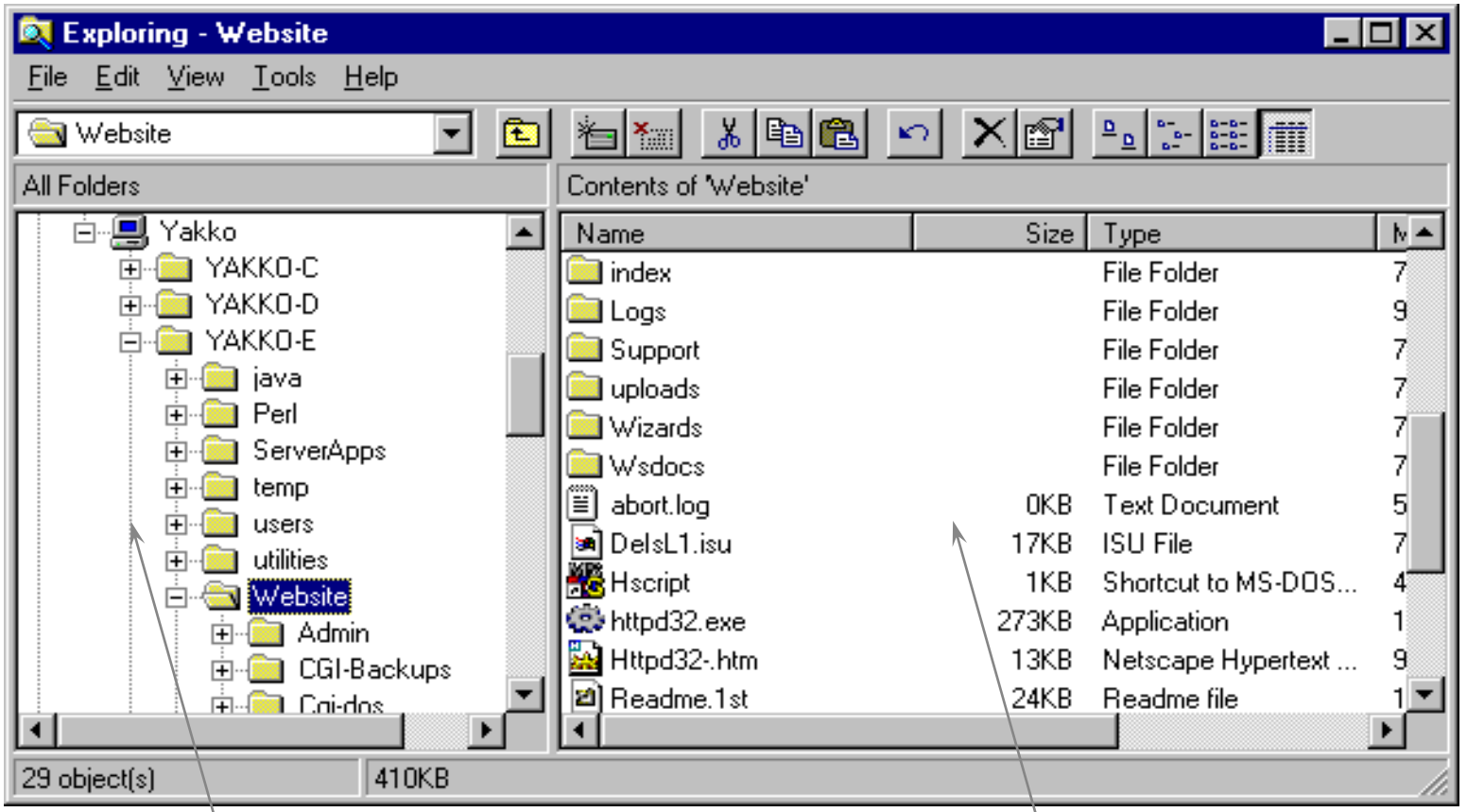
GOF (Gang of Four) 23 Patterns

- Creational Patterns (5)
- *Abstract Factory, Builder, Factory Method, Prototype, Singleton*
- Structural Patterns (7)
- *Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy*
- Behavioural Patterns (11)
- *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor*

Skylight Spelunker

- “Skylight Spelunker” is a Java framework for a file browser similar in appearance to the “Windows Explorer” included with Windows 98.
- Spelunker has two views:
  - Disks and folders in tree structure (FolderView - Left pane)
  - All contents of selected folder (ContentsView - Right pane)
- Spelunker provides support for :
  - Multiple ways of arranging ContentsView icons
  - Accessing network drives as well as local
  - Deleting, renaming and viewing disk contents

Windows Explorer Screen Shot



FolderView

ContentsView

Patterns in Spelunker example

- Composite
  - used to model the file tree data structure
- Strategy
  - used to layout the file and folder icons in ContentsView
- Observer
  - used to re-display FolderViews and ContentsViews after user requests
- Proxy and State
  - used to model password-protected network disk drives
- Command
  - used to carry out user requests

The “Composite” pattern

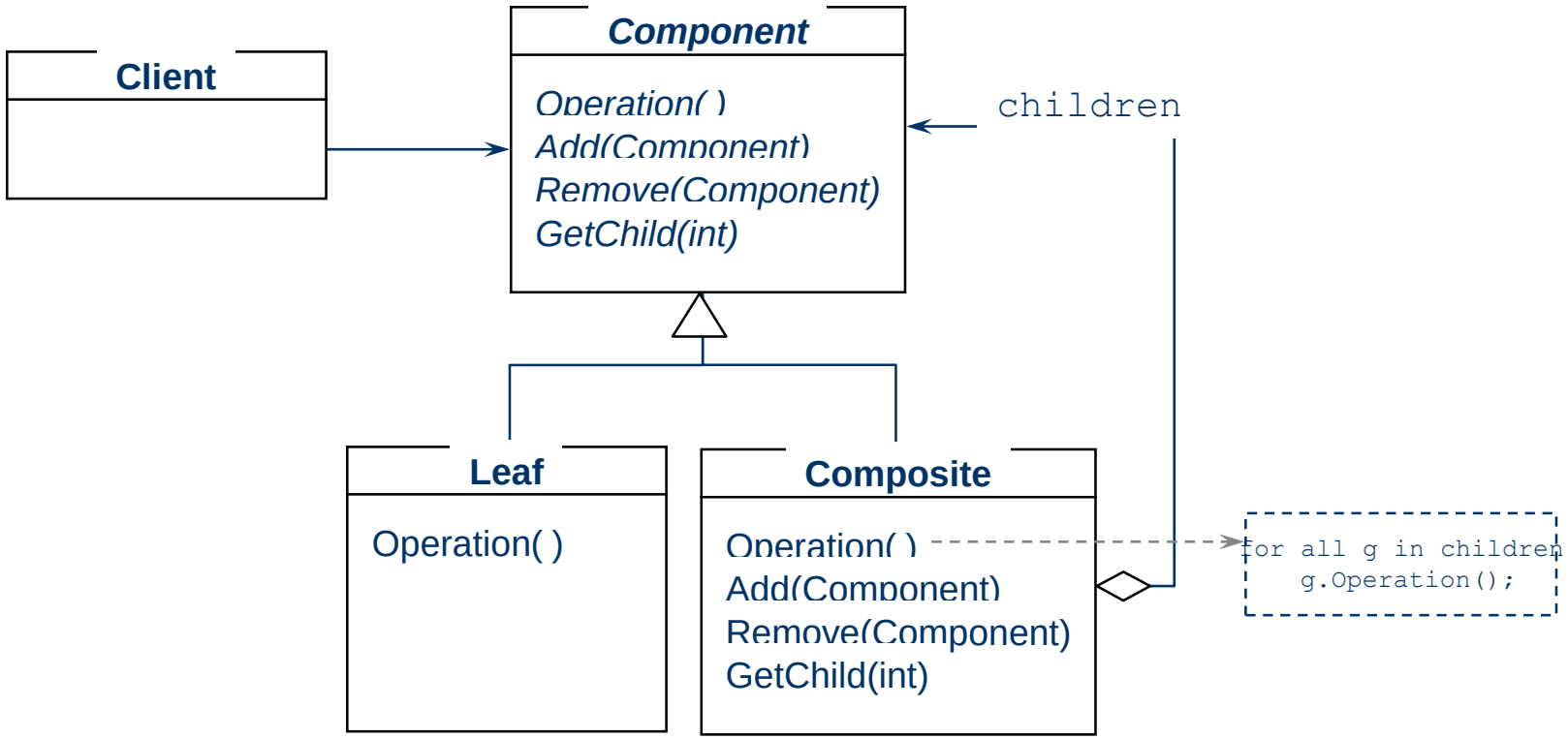
- Problem
  - What is the best way to model the Spelunker file tree?
    - The Spelunker file tree is a classic tree structure.
      Thus we need a leaf class (File) and a tree class (Folder)
      which contains pointers to the Files and Folders in it.
    - However, there are many operations that are relevant
      to both a File and a Folder (e.g., getSize()).
    - The user doesn’t treat Files and Folders differently,
      so why should calling modules have to?
    - The design would be less complex and more flexible
      if the calling module could initiate operations on a target object,
      without knowing whether the target was a File or a Folder.
    - File and Folder should share a common interface.

The “Composite” pattern

- How the pattern solves the problem
  - Intent
    - “Compose objects into tree structures to represent part-whole hierarchies. ***Composite*** lets clients treat individual objects and compositions of objects uniformly.” [GHJV94]
  - Explanation
    - The ***Composite*** pattern works by having leaf and tree objects share a common interface.
    - Create an abstract base class (or interface) that represents both File and Folder.
    - Files and Folders need to provide implementations for the same operations, but they can implement them differently.
      - E.g., leaves usually handle an operation directly, while trees usually forward the operation to its children (and/or perform additional work before or after forwarding)

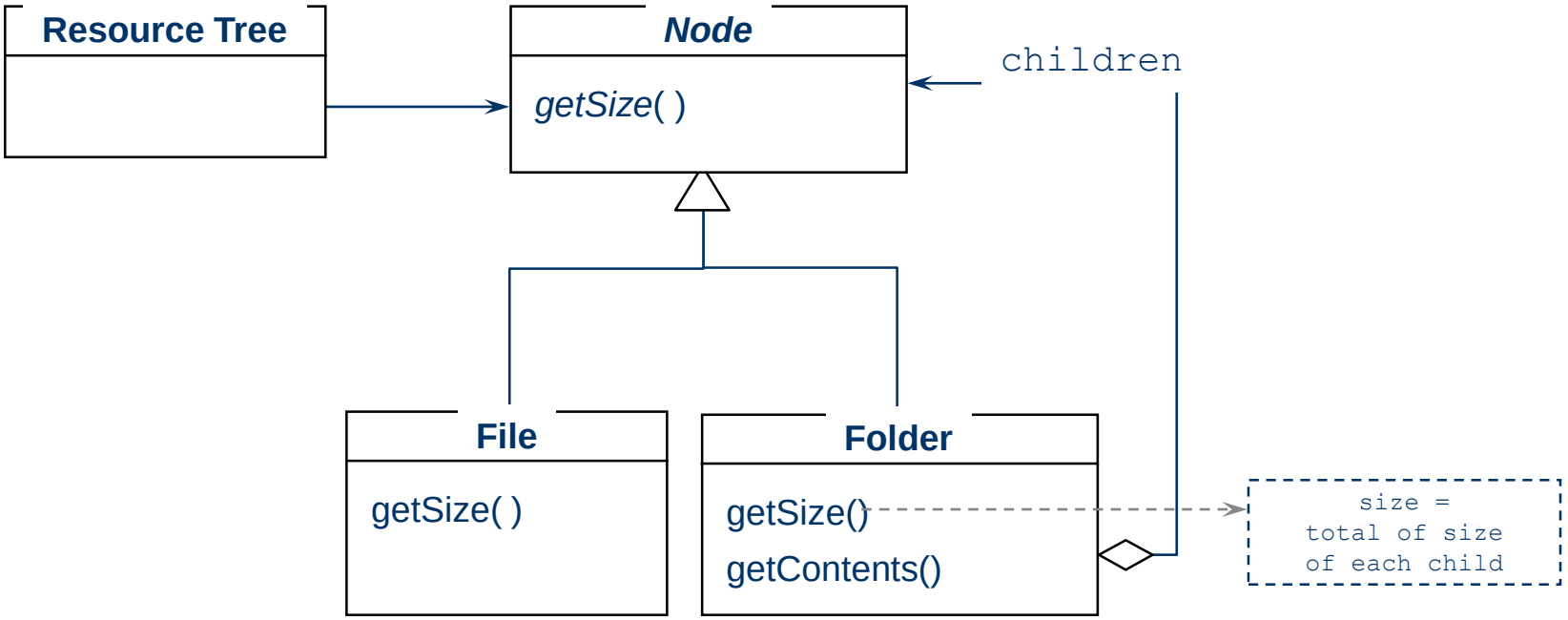
The “Composite” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “Composite” pattern

- Use of the pattern in Spelunker
  - Both File and Folder share a common interface: Node.
  - Spelunker UML



The “Composite” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public class File extends Node
{
    private long size = 0;

    public long getSize()
    {
        return size;
    }
}
```

```
public class Folder extends Node
{
    private Vector contents;

    public long getSize()
    {
        long size = 0;

        if (contents != null) {
            Enumeration e = contents.elements();
            while (e.hasMoreElements()) {
                size += ((Node)e.nextElement()).getSize();
            }
        }
        return size;
    }
}
```

The “Strategy” pattern

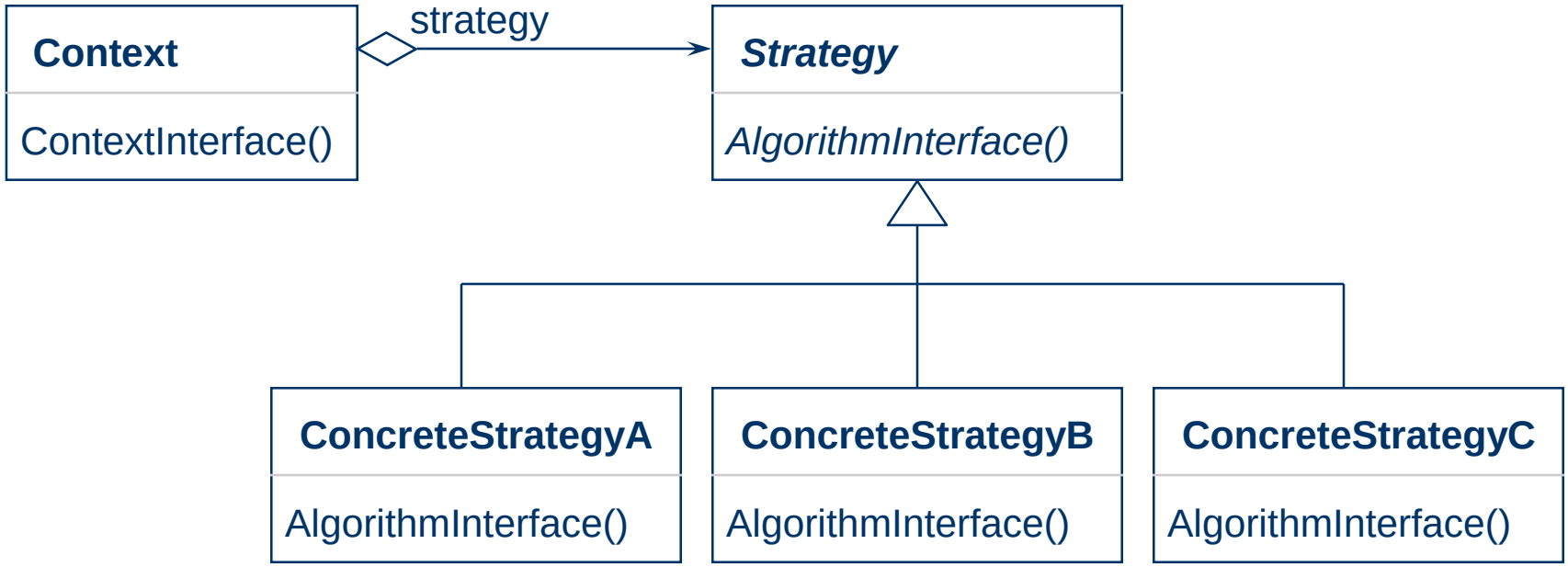
- Problem
  - The way in which the icons are arranged varies according to user preference - the user may choose an iconic view only, or a short/long detail view.
  - Including the algorithms to arrange the icons as methods in ContentsView would make it cumbersome to add new icon arrangement algorithms to ContentsView; ContentsView would have to be subclassed and some implementation details might have to be unnecessarily exposed.
  - A switch statement would most likely be used to choose the correct arrangement algorithm.

The “Strategy” pattern

- How the pattern solves the problem
  - Intent
    - “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” [GHJV94]
  - Explanation
    - The algorithms for arranging the icons are encapsulated into a separate interface.
    - The correct arrangement algorithm is chosen polymorphically.
    - ContentsView neither knows nor cares which arrangement is presently in use.

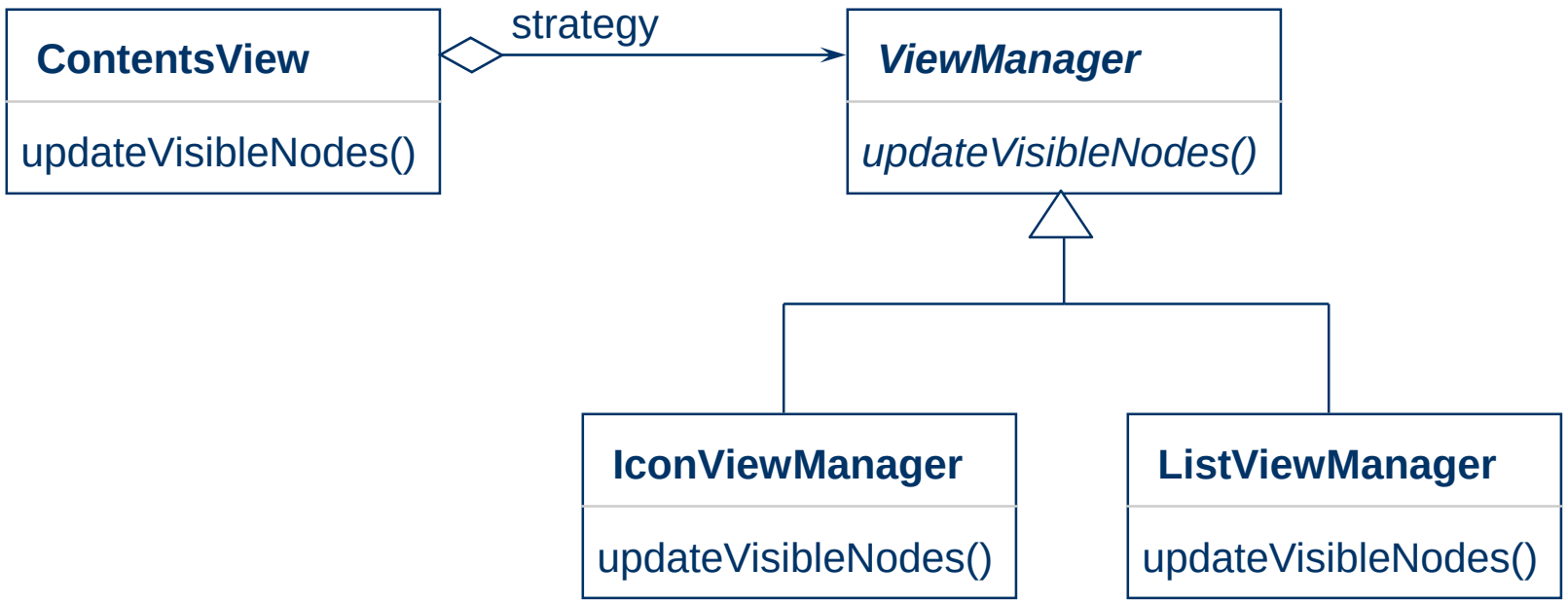
The “Strategy” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “Strategy” pattern

- Use of the pattern in Spelunker
  - ContentsView delegates the task of arranging the icons to ViewManager.
- Spelunker UML



The “Strategy” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public interface ViewManager
{
    public void updateVisibleNodes(Folder activeFolder);
}
```

```
public class ContentsView extends ResourceTreeView
{
    private ViewManager viewManager;

    public void showIconView()
    {
        viewManager = new IconViewManager(this);
    }

    public void showListView(boolean showDetail)
    {
        viewManager = new ListViewManager(this, showDetail);
    }

    public void updateVisibleNodes(Folder activeFolder)
    {
        viewManager.updateVisibleNodes(activeFolder);
    }
}
```

The “Observer” pattern

Ref. related to MVC, Model-View-Controller Pattern

https://en.wikipedia.org/wiki/Modelviewcontroller

Trygve Reenskaug, UiO/SINTEF, Norway

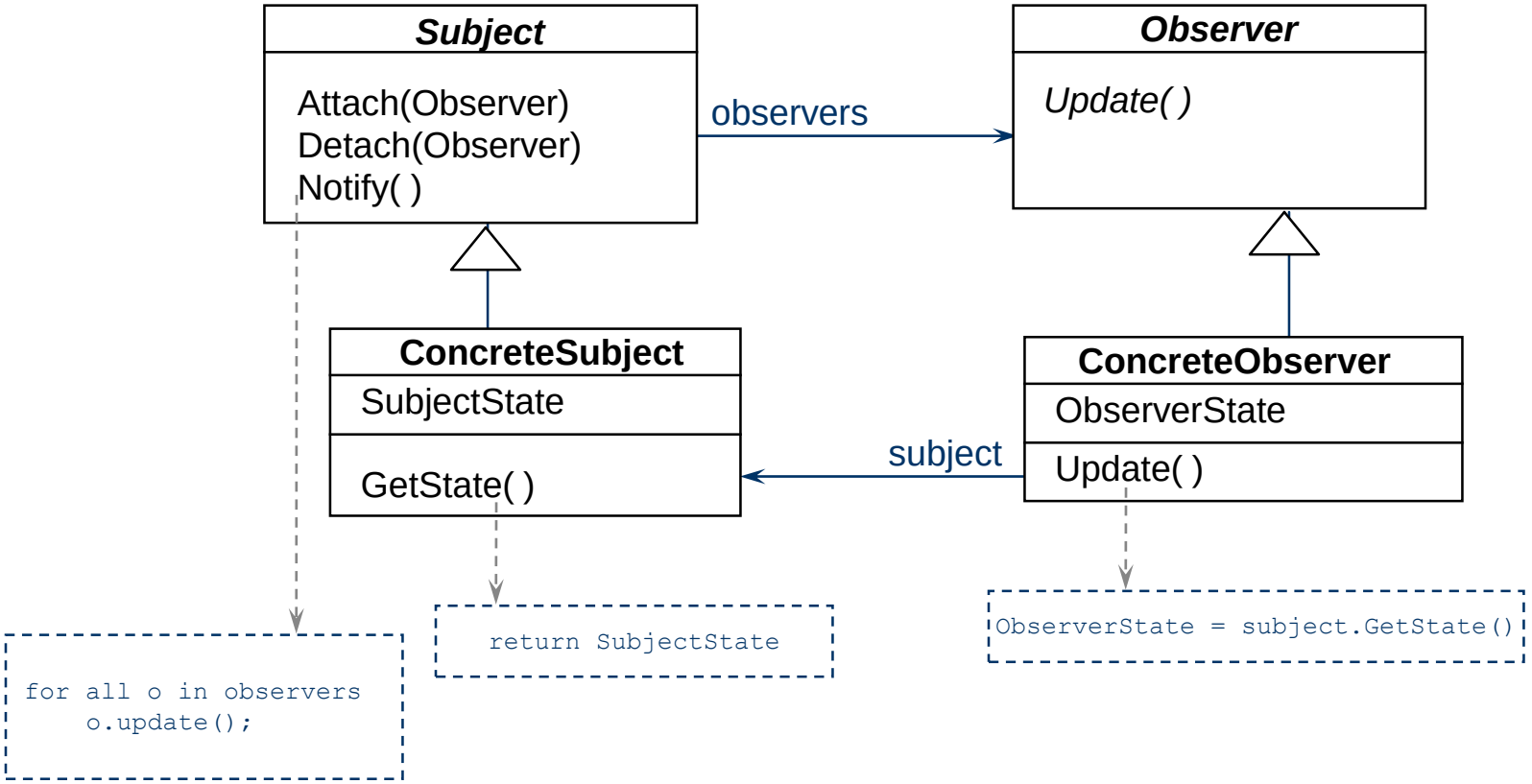
- Problem
  - What is the best way to keep all views of the file tree in sync?
    - We need to be able to re-draw the display window after the user modifies a file/folder (e.g., when user clicks on a folder to select it)
    - However, there may be several windows and panes that display the same file/folder. We need to re-draw all of them.
    - To do this, the tree needs to keep a list of all of its views, and notify each one after a modification is done.
    - However, the tree and view objects might:
      - have little other relationship besides this notification
      - need to have their code modified independently
      - need to be reused separately
    - So it would be preferable not to make them too tightly coupled to each other.

The “Observer” pattern

- How the pattern solves the problem
  - Intent
    - “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” [GHJV94]
  - Explanation
    - The ***Observer*** pattern works by defining an abstract class (or interface) with a single method signature. The method will be used as a mechanism for “observer” objects to be notified of changes in their “subject”.
    - Concrete observer sub-classes will each provide their own implementation of what to do when the notification occurs.
    - The subject can notify each observer the same way, without caring which specific sub-class of observer the object actually is.

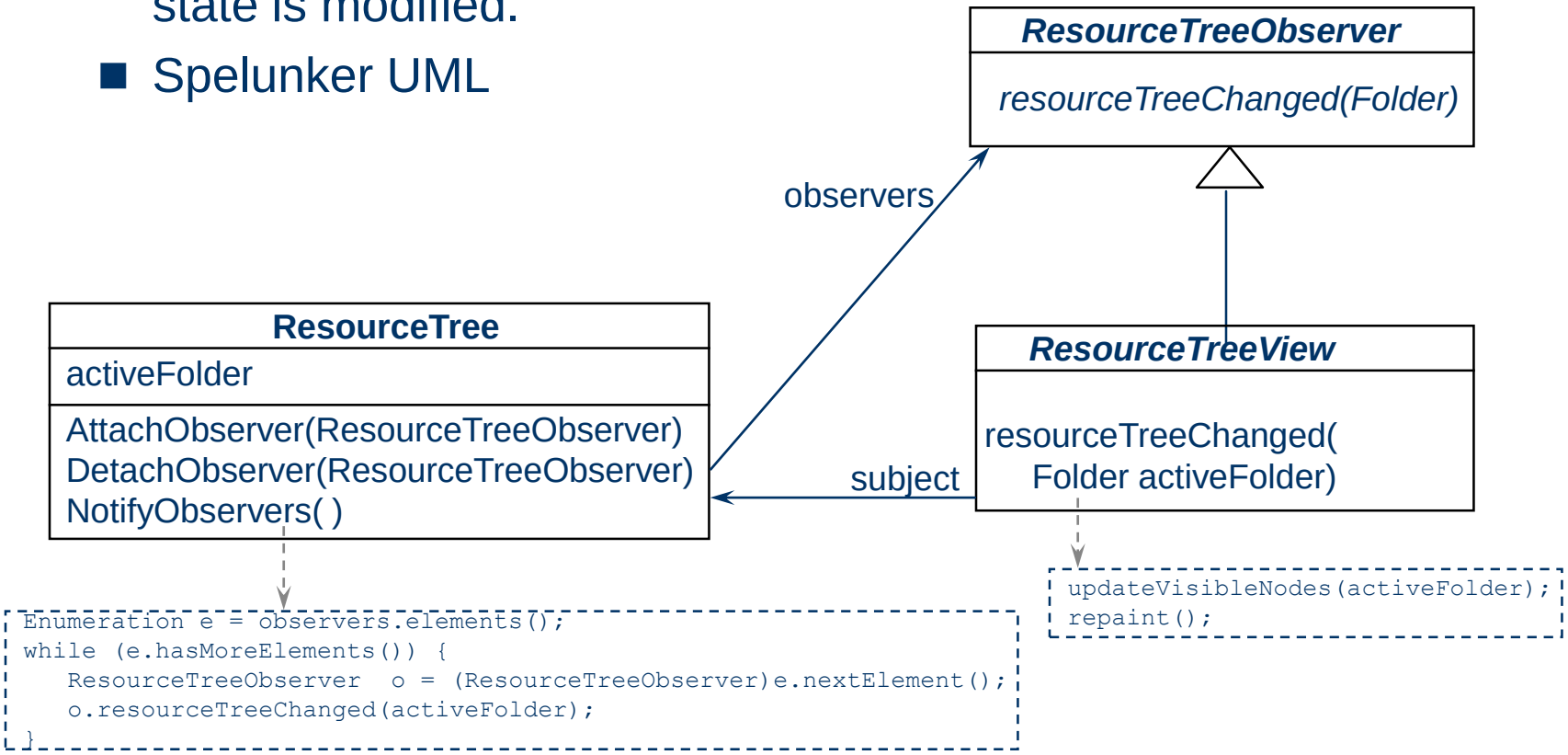
The “Observer” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “Observer” pattern

- Use of the pattern in Spelunker
  - ResourceTree notifies all ResourceTreeViews whenever its state is modified.
  - Spelunker UML



The “Observer” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public class ResourceTree {
    private Vector observers;

    public void setActiveFolder(Folder folder)
    {
        if (activeFolder != folder) {
            activeFolder = folder;
            notifyObservers();
        }
    }

    public void notifyObservers()
    {
        Enumeration e = observers.elements();
        while (e.hasMoreElements()) {
            ((ResourceTreeObserver)e.nextElement()).resourceTreeChanged(activeFolder);
        }
    }
}
```

```
public abstract class ResourceTreeView extends Panel implements ResourceTreeObserver {

    public void resourceTreeChanged(Folder activeFolder)
    {
        updateVisibleNodes(activeFolder);
        repaint();
    }
}
```

The “Proxy” pattern

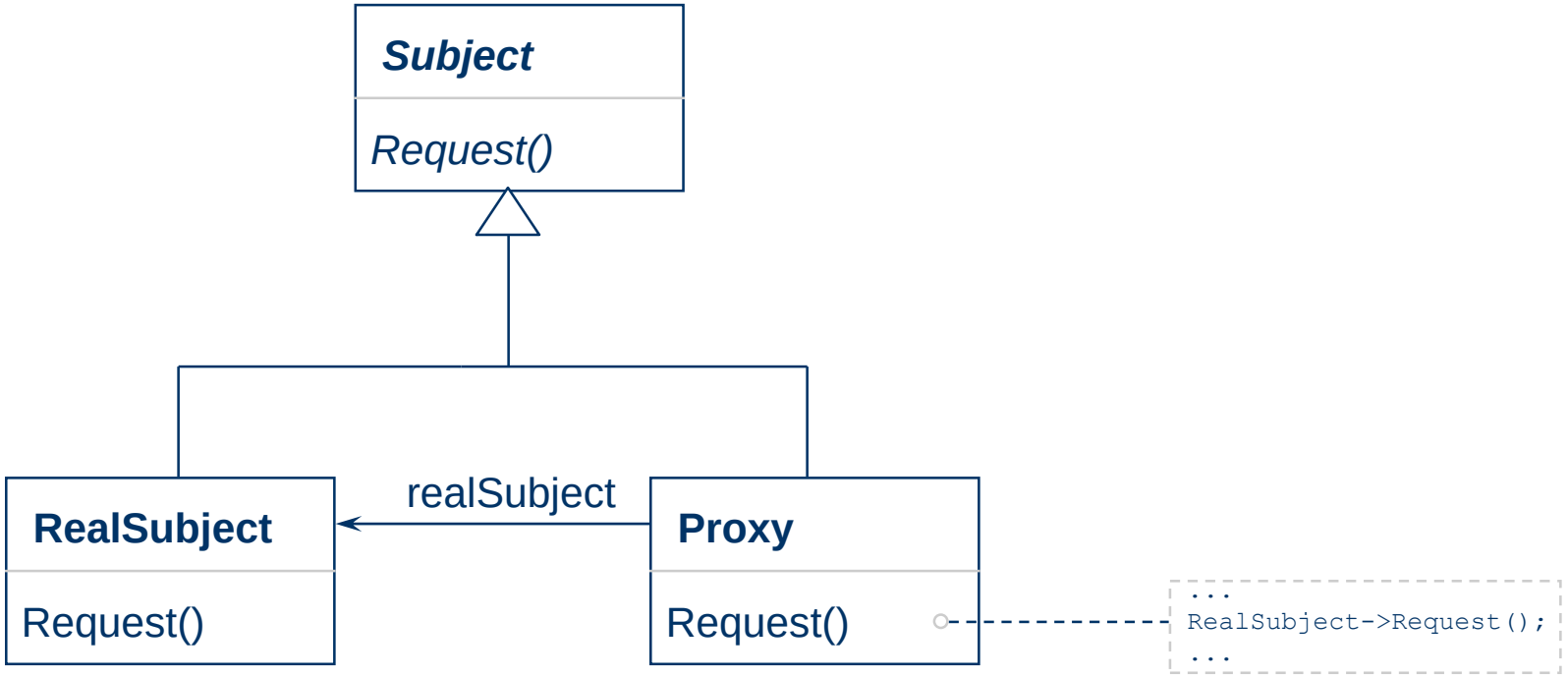
- Problem
  - Network drives might require the user to login before the drive can be accessed - however, the protocol for accessing a network drive when logged in might not differ from accessing a local drive.
  - LocalDrive should not contain network code - this code should be moved to a separate class, i.e. NetworkDrive.
  - Creating NetworkDrive as a subclass of LocalDrive would be complicated and unwieldy - we would have to check access everytime a drive operation was requested.
  - Creating NetworkDrive as a subclass of Folder would force us to duplicate all drive access operations already in LocalDrive.

The “Proxy” pattern

- How the pattern solves the problem
  - Intent
    - “Provide a surrogate or placeholder for another object to control access to it.” [GHJV94]
    - “A ***Protection Proxy*** controls access to the original object. ***Protection Proxies*** are useful when objects should have different access rights.” [GHJV94]
  - Explanation
    - The network protocols necessary for logging in and out are moved into a subclass of Folder called NetworkDrive.
    - NetworkDrive contains the code necessary for logging in and out of a network drive.
    - After logging in, NetworkDrive delegates drive access requests to LocalDrive (indirectly through ConnectionState).

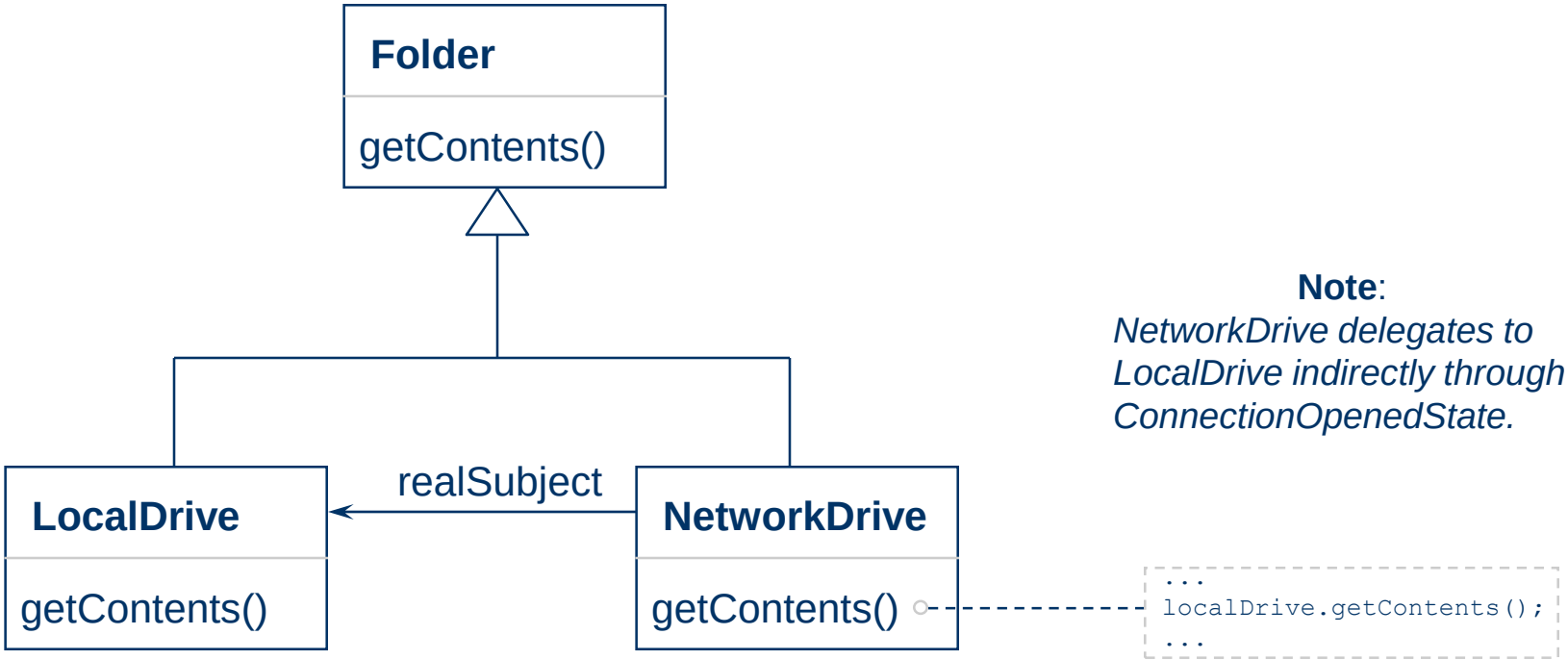
The “Proxy” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “Proxy” pattern

- Use of the pattern in Spelunker
  - NetworkDrive acts as a Proxy for a remote LocalDrive.
  - Spelunker UML



Note:
NetworkDrive delegates to LocalDrive indirectly through ConnectionOpenedState.

The “Proxy” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public class NetworkDrive extends Folder
{
    private ConnectionState connectionState;

    public Vector getContents(Folder folder)
    {
        return connectionState.getContents(folder);
    }
}
```

```
public class ConnectionOpenedState extends Object implements ConnectionState
{
    private LocalDrive localDrive;

    public Vector getContents(Folder folder)
    {
        return localDrive.getContents(folder);
    }
}
```

The “State” pattern

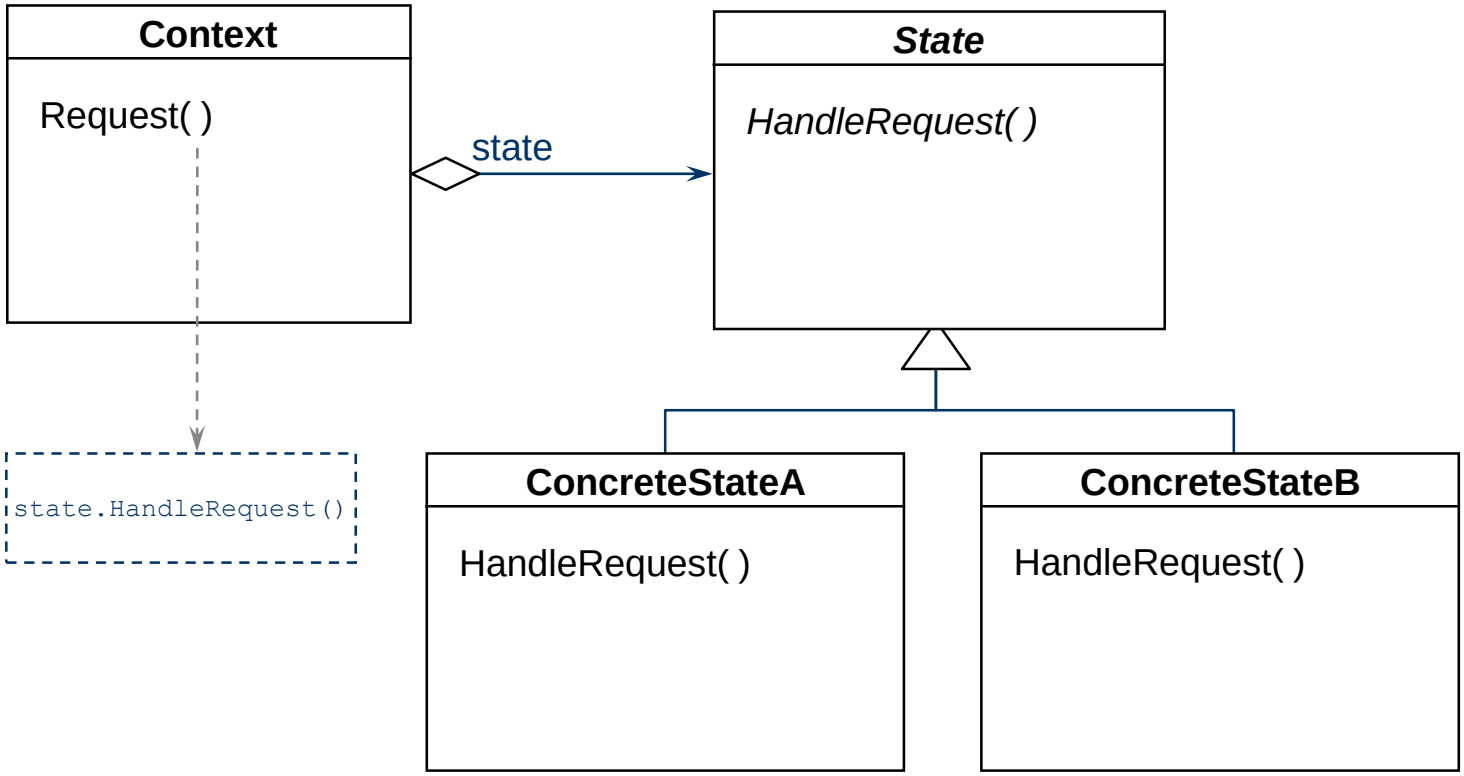
- Problem
  - What is the best way to perform password-protection processing on network drives?
    - Network drives need to act differently depending on whether the user has logged in or not; e.g., the user cannot examine or modify a network drive until they log in.
    - This can be accomplished by checking a condition before executing each operation; e.g., “if (loggedIn())”. But this is ugly code, as well as being inefficient and repetitive.
    - This is also difficult to extend: what if we need to implement another set of checks for another condition; e.g., “if (!disconnected())”?
    - The design would be less complex and more flexible if we could isolate in one location all behavior related to a particular state of the object.

The “State” pattern

- How the pattern solves the problem
  - Intent
    - “Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” [GHJV94]
  - Explanation
    - The ***State*** pattern works by creating an abstract class (or interface) with method signatures for every state-dependent operation in the main object, and concrete sub-classes that provide implementations for these methods. The main object then delegates each of these operations to the state object it is currently using.
    - Each state class can implement each operation in its own way (e.g., perform unique processing, disallow the operation, throw an exception, etc.).
    - The main object can change its behavior by changing the state object it is using.
    - This is a very clean design - and also extendible: we can simply add new state classes to add additional behavior, without modifying the original object.

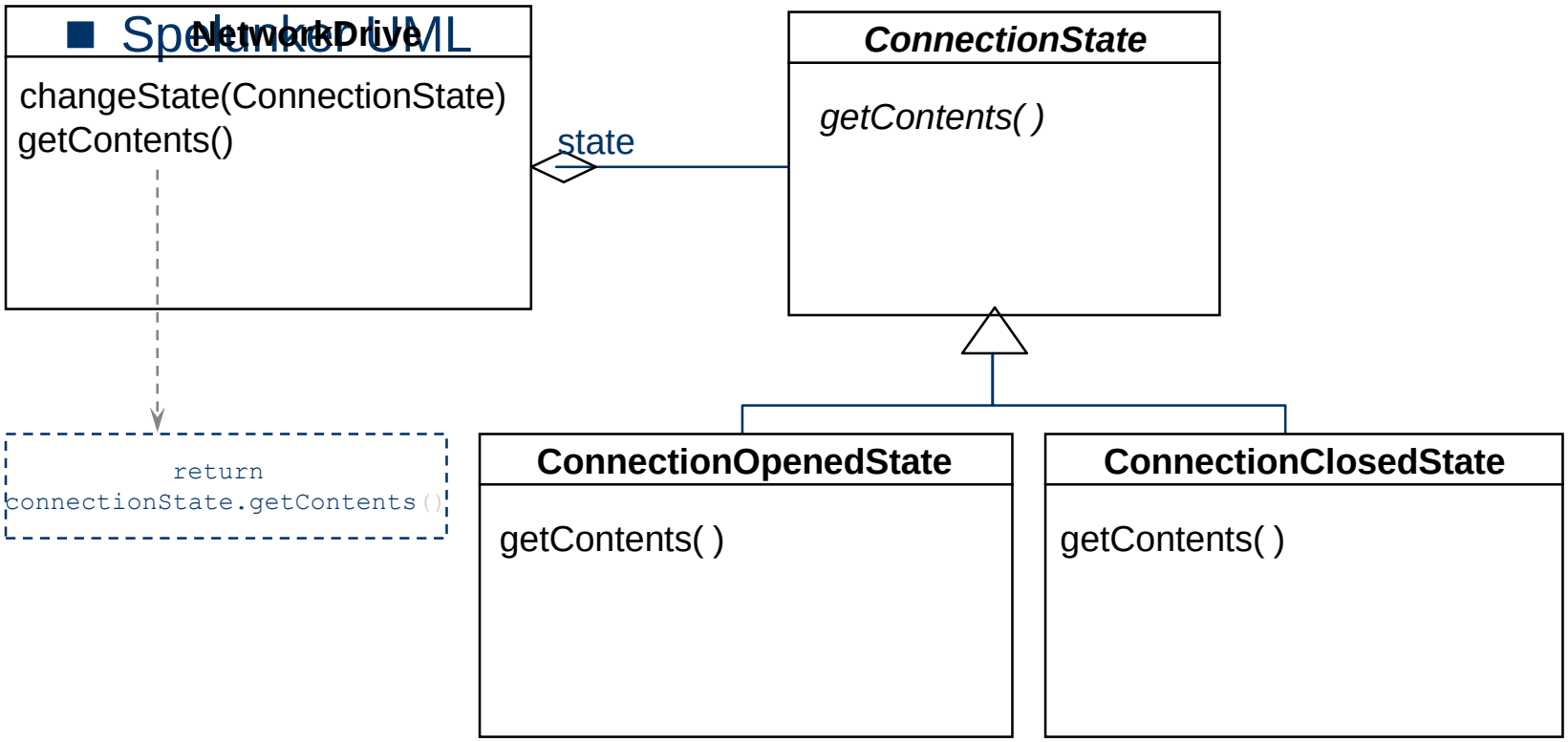
The “State” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “State” pattern

- Use of the pattern in Spelunker
  - The NetworkDrive delegates operations to its ConnectionState.
  - Spelunker UML



The “State” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public class ConnectionClosedState implements ConnectionState {

    public void login() {
        LocalDrive localDrive = null;

        // login and initiate localDrive

        networkDrive.changeState(new ConnectionOpenedState(networkDrive, localDrive));
    }

    public Vector getContents(Folder folder) {
        login();
        return networkDrive.getContents(folder);
    }
}
```

```
public class ConnectionOpenedState implements ConnectionState {
    public void login() {
        // display error
    }

    public Vector getContents(Folder folder) {
        return localDrive.getContents(folder);
    }
}
```

The “Command” pattern

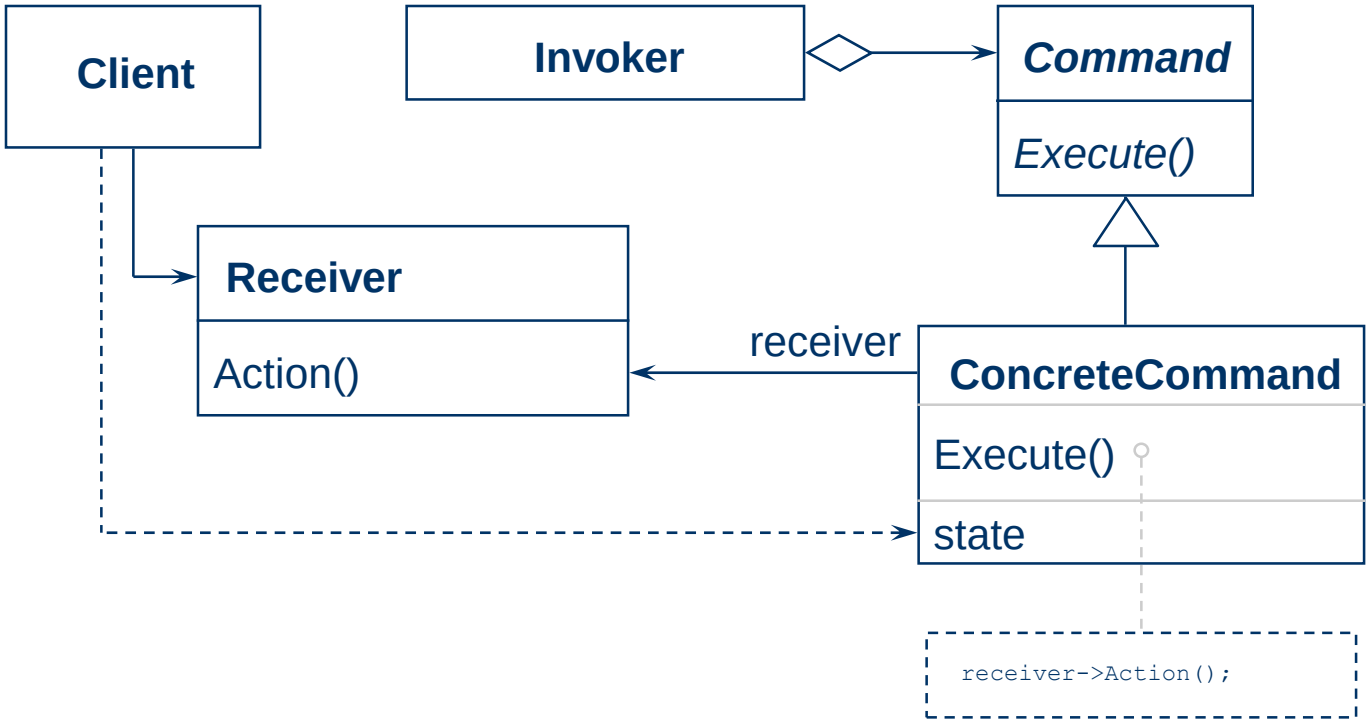
- Problem
  - A request might need access to any number of classes.
  - The initiator of the request should not be tightly coupled to these classes.
  - Requests should be storable to support undoable operations; therefore, requests must be accessible through some common interface.
  - How do we implement requests without coupling them to the initiator or target, or requiring the initiator to know the implementation details of the request ?
  - Implementing the code for all requests in one class would centralize the application and make it difficult to create new requests.

The “Command” pattern

- How the pattern solves the problem
  - Intent
    - “Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.” [GHJV94]
  - Explanation
    - Places the implementation of a request into a separate class.
    - Initiators of the request do not know any implementation details of the request - they simply fire it off by calling the execute() method.
    - The targets of the request do not need to know anything about the request.
    - All requests are accessible through a common interface.
    - The correct implementation is chosen polymorphically.

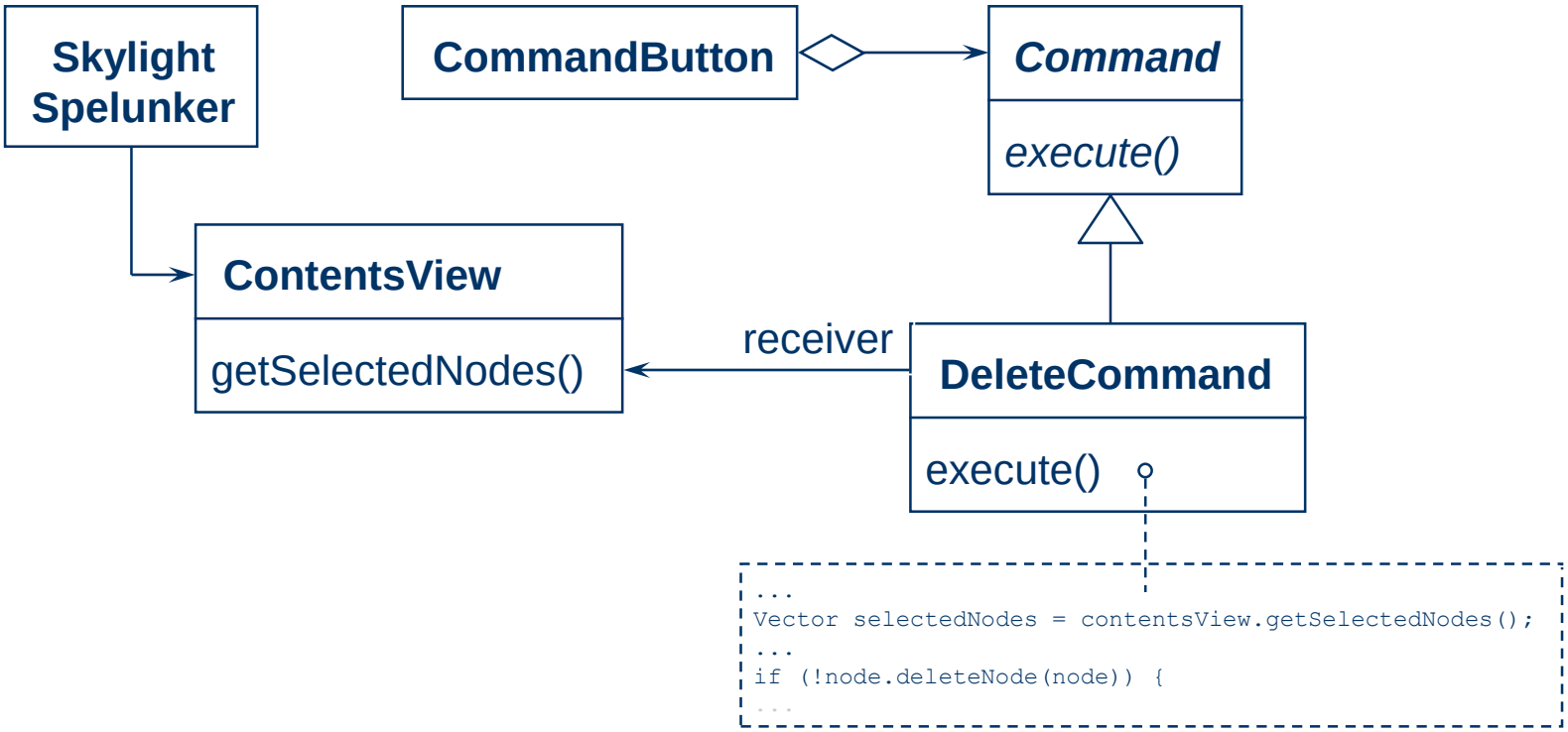
The “Command” pattern

- How the pattern solves the problem, cont.
  - Gang of Four UML [GHJV94]



The “Command” pattern

- Use of the pattern in Spelunker
  - Used to implement user operations on files and folders.
  - Spelunker UML



The “Command” pattern

- Use of the pattern in Spelunker, cont.
  - Code examples

```
public abstract class Command extends Object {
    public abstract void execute();
}
```

```
public class DeleteCommand extends Command {
    private ContentsView contentsView;
    private ResourceTree resourceTree;

    public void execute() {
        (code for retrieving all selected Nodes
                from ContentsView and deleting them)
    }
}
```

```
public class CommandButton extends Button {
    private Command command;

    public CommandButton(String label, Command command) {
        super(label);
        this.command = command;
    }

    public boolean action(Event  e,
                          Object what) {
        command.execute();
        return super.action(e, what);
    }
}
```

Basis Litteratur (Pattern kataloger)

Design Patterns, Elements of Reusable Object-Oriented Software
Gamma et. al. Addison-Wesley, ISBN 0-201-63361-2

Analysis Patterns, Reusable Object Models
Martin Fowler, Addison-Weslwy, ISBN 0-201-89542-0

Pattern-Oriented Software Architecture
F. Buschmann et. al, J. Wiley, ISBN 0-471-95869-7

AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis
W. Brown et. al, J. Wiley, ISBN 0-471-19713-0

http://st-www.cs.uiuc.edu/users/patterns/patterns.html

http://c2.com/ppr/index.html

Tilleggs litteratur

Pattern Languages of Program Design 1
J. Coplien, Douglas Schmidt
Addison-Wesley, ISBN 0-201-60734-4, 1995

Pattern Languages of Program Design 2
J. Vlissides, J. Coplien, N. Kerth
Addison-Wesley, ISBN 0-201-89527-7, 1996

Design Patterns for Object-Oriented Software Development,
Pree, Addison-Wesley, ISBN 0-201-42294-8, 1995

CORBA Design Patterns, T. Mowbray, R. Malveau, j. Wiley, 1997,
ISBN 0-471-15882-8

Communications of ACM, “Software Patterns” - special issue
October 1996, Vol. 39, Number 10

Refactoring - Improving the design of existing code

- 1. Refactoring - a first example
- 2. Principles in refactoring
- 3. Bad Smells in Code
- 4. Building Tests
- 5. Toward a catolog of refactorings
- 6. Composing Methods
- 7. Moving Features between objects
- 8. Organizing data
- 9. Simplifying Conditional Expressions
- 10. Making Method calls simpler
- 11. Dealing with Generalization
- 12. Big Refactorings
- 13. Refactoring, Reuse and Reality
- 14. Refactoring tools

M. Fowler, with K. Beck, J. Brant, W. Opdyke, D. Roberts, Addison-Wesley, August 1999

Refactoring: Improving the design of existing code

Refactoring - What and Why ?

- Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.
- Improving to make it easier to understand and cheaper to modify

When should you refactor

- The rule of three - Three strikes and you refactor
- Refactor when you add function
- Refactor when you need to fix a bug
- Refactor as you do a code review

Why refactoring works

- Programs that are hard to read are hard to modify
- Programs that have duplicated logic are hard to modify
- Program that require additional behaviour that requries you to change running code are hard to modify
- Programs with complex conditional logic are hard to modify

- We want programs that are easy to read, that have all logic specified in one and only one place, do not allow changes to endanger existing behaviour, and allow conditional logic to be expressed as simply as possible

Refactoring strategies

- Composing Methods
- Moving features between objects
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Dealing with generalization
- Big refactorings

Composing Methods

- Extract method
- Inline method
- Inline temp
- Replace temp with query
- Introduce explaining variable
- Split temporary variable
- Remove assignments to parameters
- Replace method with method object
- Substitute algorithm

Moving Features between objects

- Move method
- Move field
- Extract Class
- Inline Class
- Hide Delegate
- Remove middle man
- Introduce foreign method
- Introduce local extension

Bad Smells in Code (1/4)

- Duplicated Code ***(extract method, extract class, pull up method, form template method)***
- Long Method **(*extract method, replace temp with query, replace method with method object, decompose conditional)***
- Large Class ***(extract class, extract subclass, extract interface, replace data value with object)***
- Long ParameterList ***(replace parameter with method, introduce parameter object, preserve whole object)***
- Divergent Change ***(extract class)***

Bad Smells in Code (2/4)

- Shotgun Surgery ***(move method, move field, inline class)***
- Feature Envy ***(move method, move field, extract field)***
- Data Clumps ***(extract class, introduce parameter object, preserve whole object)***
- Primitive Obsession **(*replace data value with object, extract class, introduce parameter object, replace array with object, replace type code with class/subclasses, replace type code with state/strategy)***
- Switch Statements ***(replace conditional with polymorphism, replace type code with subclasses/state/strategy, replace parameter with explicit methods, introduce null object)***

Bad Smells in Code (3/4)

- Parallell Inheritance Hierarchies *(**move method, move field)***
- Lazy Class ***(inline class, collapse hierarchy)***
- Speculative Generality ***(collapse hierarchy, inline class, remove parameter, rename method)***
- Temporary Field ***(extend class, introduce null object)***
- Message Chains ***(hide delegate)***
- Middle Man *(**remove middle man, inline method, replace delegation with inheritance)***
- Inappropriate Intimacy ***(move method, move field, change bidirection to unidirectional)***

Bad Smells in Code (4/4)

- Alternative classes with different interfaces ***(rename method, move method)***
- Incomplete Library Class ***(introduce foreign method, introduce local extension)***
- Data Class ***(move method, encapsulate field, encapsulate collection)***
- Refused Bequest ***(replace inheritance with delegation)***
- Comments ***(extract method, introduce assertion)***

Organizing data

- Self encapsulate field
- Replace data value with object
- Change value to reference
- Change reference to value
- Replace array with object
- Duplicate observed data
- Change unidirectional association to bidirectional
- Change bidirectional association to unidirectional
- Replace magic number with symbolic constant
- Encapsulate field
- Encapsulate collection
- Replace record with data class
- Replace type code with class/sublasses
- Replace type code with state/strategy
- Replace subclass with fields

Simplifying Conditional Expressions

- Decompose conditional
- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Remove control flag
- Replace nested conditional with guard clauses
- Replace conditional with polymorphism
- Introduce null object
- Introduce assertion

Making Method calls simpler

- Rename method
- Add parameter
- Remove parameter
- Separate query from modifier
- Parameterize method
- Replace parameter with explicit methods
- Preserve whole object
- Replace parameter with method
- Introduce parameter object
- Remove setting method
- Hide method
- Replace constructor with factory method
- Encapsulate downcast
- Replace error code with exception
- Replace exception with test

Dealing with Generalization

- Pull up field
- Pull up method
- Pull up constructor body
- Push down method
- Push down field
- Extract subclass
- Extract superclass
- Extract interface
- Collapse hierarchy
- Form template method
- Replace inheritance with delegation
- Replace delegation with inheritance

Big refactorings

- Tease apart inheritance
- Convert procedural design to objects
- Separate domain from presentation
- Extract hierarchy

The rhythm of refactoring ...

- test, small change, test, small change, test, ….
- … allows refactoring to move quickly and safely

AntiPatterns

- Refactoring Software, Architectures , and Projects in Crisis: W. Brown, R. Malveau. H. McCormick, T. Mowbray, Wiley, 1998
- AntiPattern: A commonly occuring patterns or solution that generates decidely negative consequences. An AntiPatterns may be a pattern in the wrong context. When properly documented, an AntiPattern comprises a paired AntiPattern solution with a refactored solution.

Software Development AntiPatterns

- The Blob (from the film)
- Continuous Obsolescence
- Lava Flow
- Ambiguous Viewpoint
- Functional Decomposition
- Poltergeists
- Boat Anchor
- Golden Hammer
- Dead End
- Spaghetti Code
- Input Kludge
- Walking through a Minefield
- Cut-and-Paste Programming
- Mushrooom management

Software Architecture AntiPatterns

- Autogenerated Sovepipe
- Stovepipe Enterprise
- Jumble
- Stovepipe System
- Cover your Assets
- Vendor Lock-In
- Wolf Ticket
- Architecture by Implication
- Warm bodies
- Design by Committee
- Swiss Army Knife
- Reinvent the Wheel
- The Grand Old Duke of York

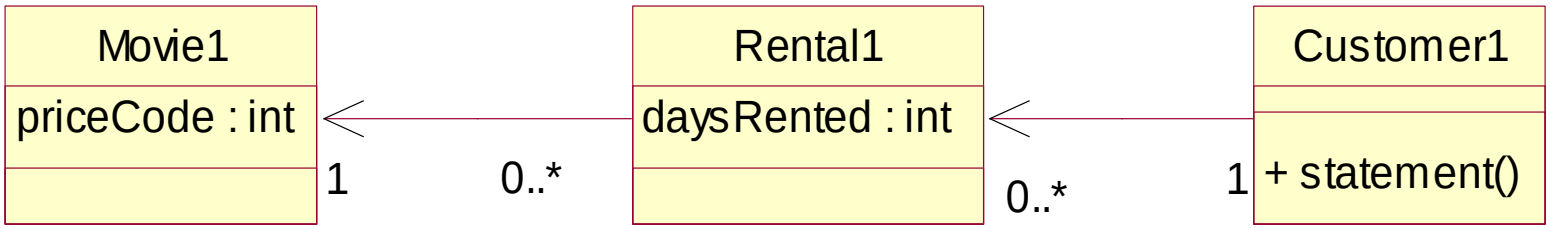
Software Project Management AntiPatterns

- Blowhard Jamboree
- Analysis Paralysis
- Viewgraph Engineering
- Death by Planning
- Fear of Success
- Corncob
- Intellectual Violence
- Irrational Management
- Smoke and Mirrors
- Project Mismanagement
- Throw it over the wall
- Fire Drill
- The Feud
- E-mail is dangerous

Practical Refactoring exercise

Example:Video rental

- Bad smells:
- Long Method
- Feature Envy
- Switch statements
- Temporary Fields
- Support change ?: *Add HTML statement, change classification of films*



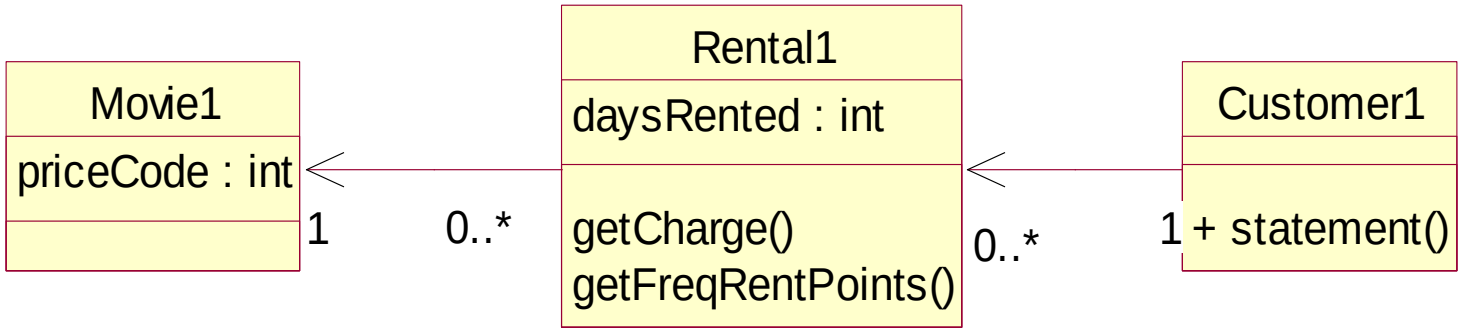
extract from statement()

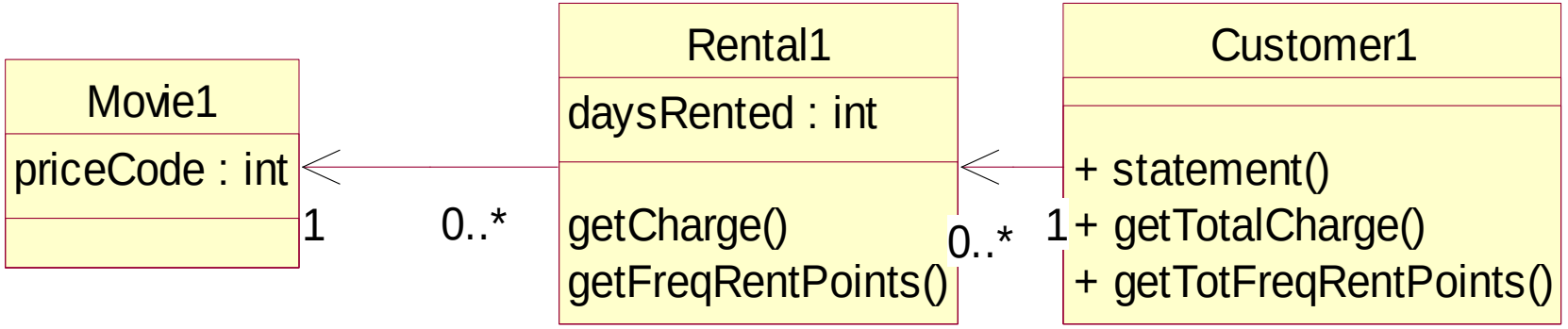
```
public String statement() {

/ ….. Determine amounts for each rental
  Switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR
     thisAmount += 2;
   …..


// add frequent renter points
frequentRenterPoints ++;

if (each.getMovie().getPriceCode() -----

// show figures for this rental

// add footer lines

}
```

Refactorings:Video rental

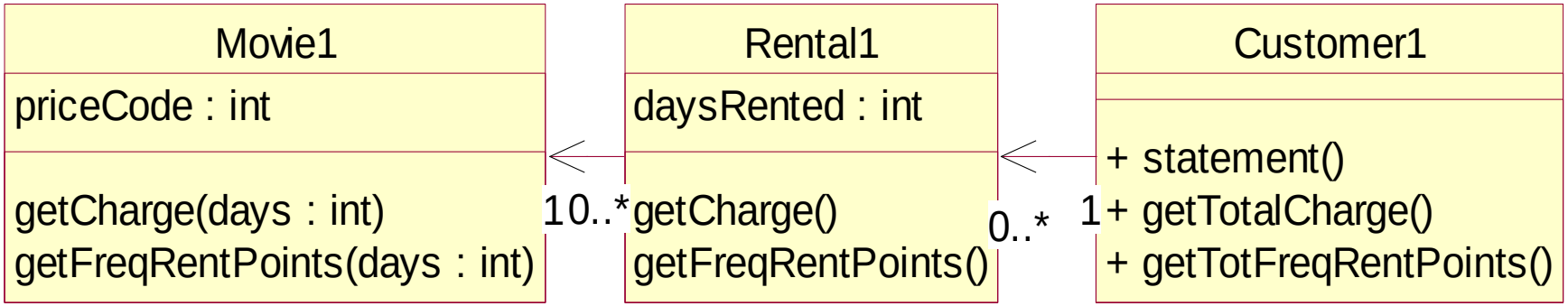
- *Create Tests to check refactoring correctness*
- Decomposing and redistributing the statement method (extract method, moving the amount calculation amountFor() (move the method from customer to rental), rename variables (I.e each -> aRental)
- similar: extracting frequent renter points, removing temps (totals) replaceTempWithQuery totalAmount/freqRentPoint,

Extracting and Moving methods



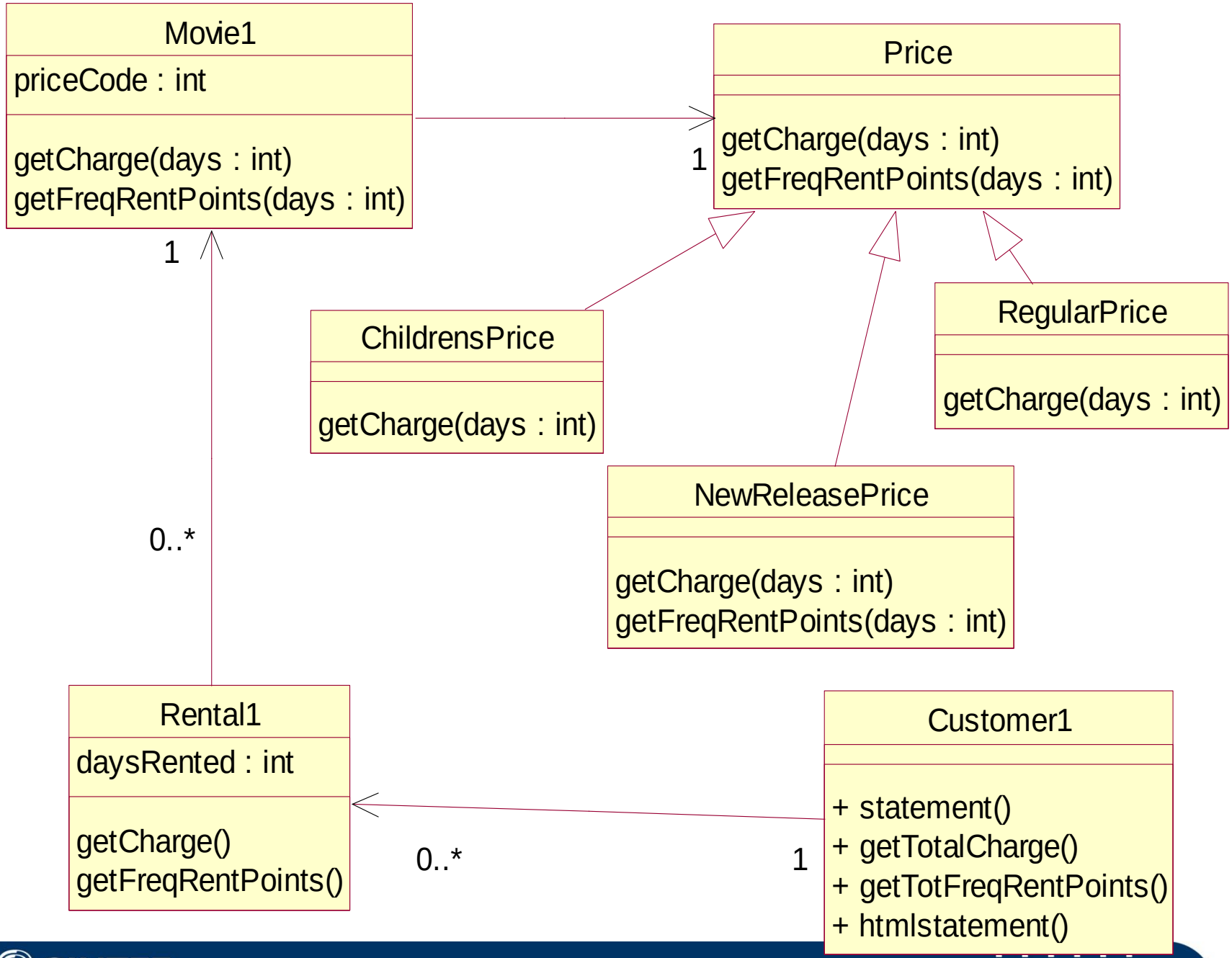


Move calculation of charge and points to the “expert”



Refactorings:Video rental

- replace conditional logic on price code with polymorphism, using inheritance - problem: a movie can change its classification during its lifetime -> use the state pattern for price code object (or strategy) -> replace type code with state/strategy, move method (switch into price class), replace conditional with polymorphism to eliminate switch


Movie1
priceCode : int
getCharge(days : int)
getFreqRentPoints(days : int)
1
Price
getCharge(days : int)
getFreqRentPoints(days : int)
1
ChildrensPrice
getCharge(days : int)
RegularPrice
getCharge(days : int)
NewReleasePrice
getCharge(days : int)
getFreqRentPoints(days : int)
0..*
Rental1
daysRented : int
getCharge()
getFreqRentPoints()
0..*
1
Customer1
+ statement()
+ getTotalCharge()
+ getTotFreqRentPoints()
+ htmlstatement()