Introduction to Design Patterns

Pr. Jean-Marc Jézéquel

IRISA - Univ. Rennes 1

Campus de Beaulieu
F-35042 Rennes Cedex
Tel : +33 299 847 192 Fax : +33 299 847 171
e-mail : jezequel@irisa.fr
http://www.irisa.fr/prive/jezequel

Outline

⌘1. Context and Origin
⌘2. A Concrete Example: the Observer
⌘3. Using Design Patterns
⌘4. Describing Design Patterns
⌘5. Overview of GoF's catalog
⌘6. More into the details...
⌘7. Conclusion

1. Context and Origin

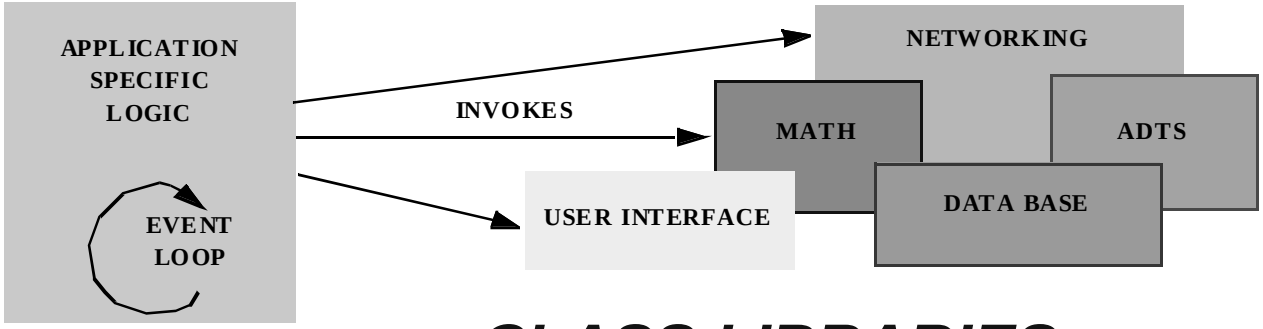
Introduction

- Design Patterns = Tricks of the Trade
  - Recurring Structures of Solutions to Design Problems (both static & dynamic)
  - Semi-formalization of OO design tricks
  - Relative independence from (OO) languages
- Related Notions
  - Analysis Patterns / Architectural Patterns / Code Patterns (idioms)
  - Frameworks (reuse design + code)

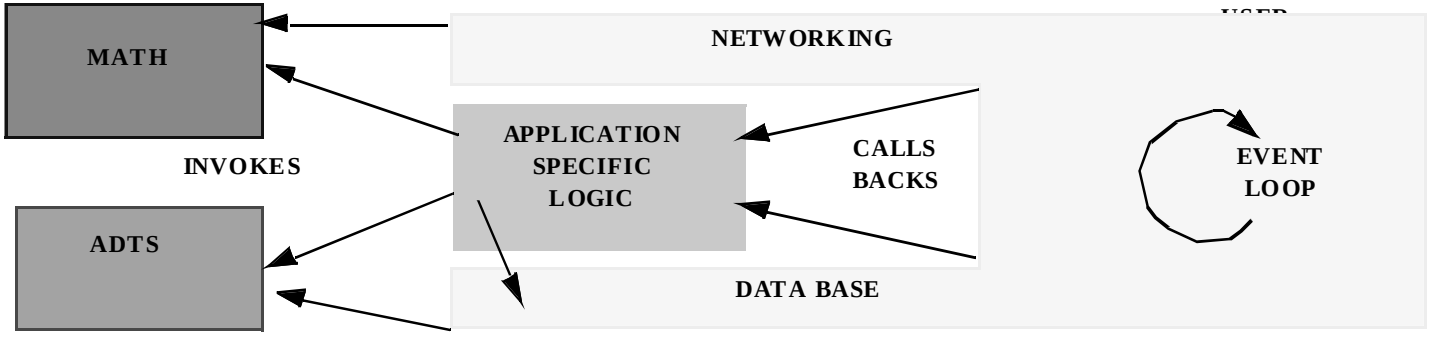
Framework Characteristics

- Provides an integrated set of domain specific functionality
  - e.g., business applications, telecommunications, window systems, databases, distributed applications, OS kernels
- Semi-complete application
  - Complete applications are developed by inheriting from, and instantiating parameterized framework components
- Often exhibit inversion of control at runtime
  - i.e., the framework determines which methods to invoke in response to events

Class Libraries vs. OO Frameworks



CLASS LIBRARIES



OBJECT-ORIENTED FRAMEWORK

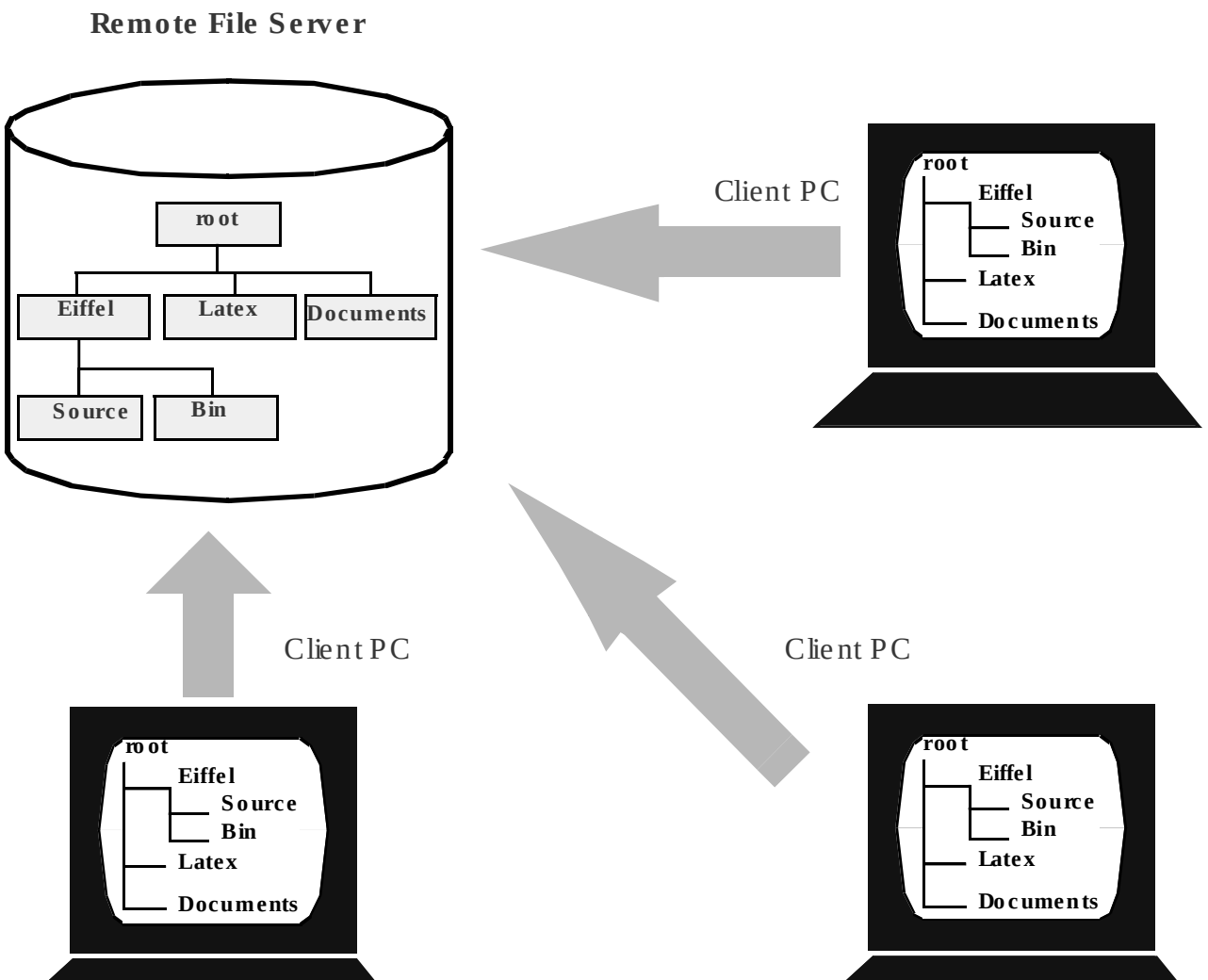
Origin of Design Patterns

- GoF's Book: A catalog
  - Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, Vlissides). Addison Wesley, 1995
- Earlier works by Beck, Coplien and others...
- Origin of Patterns in Architecture (C. Alexander)
  - Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.

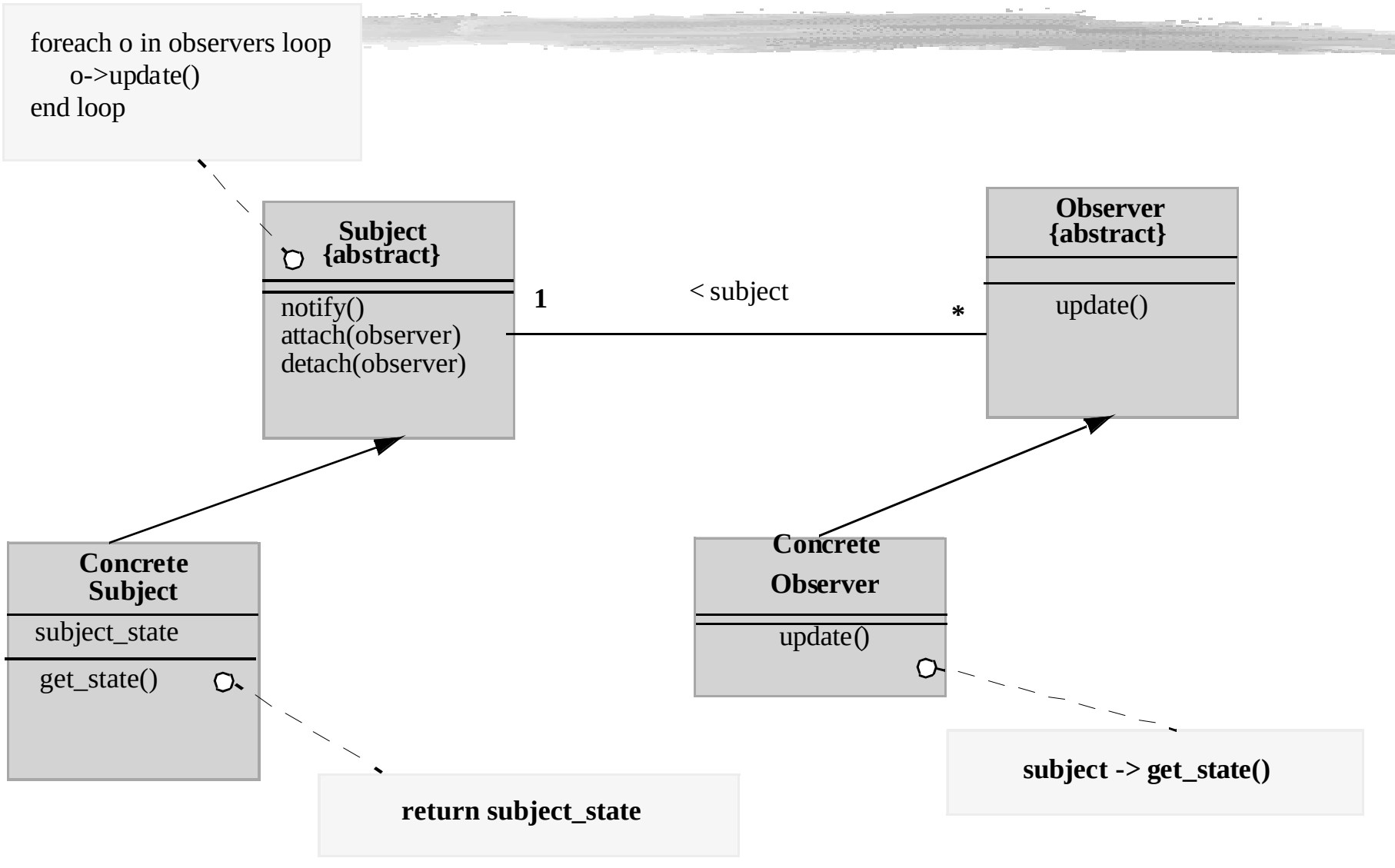
2. Example

The Observer Pattern

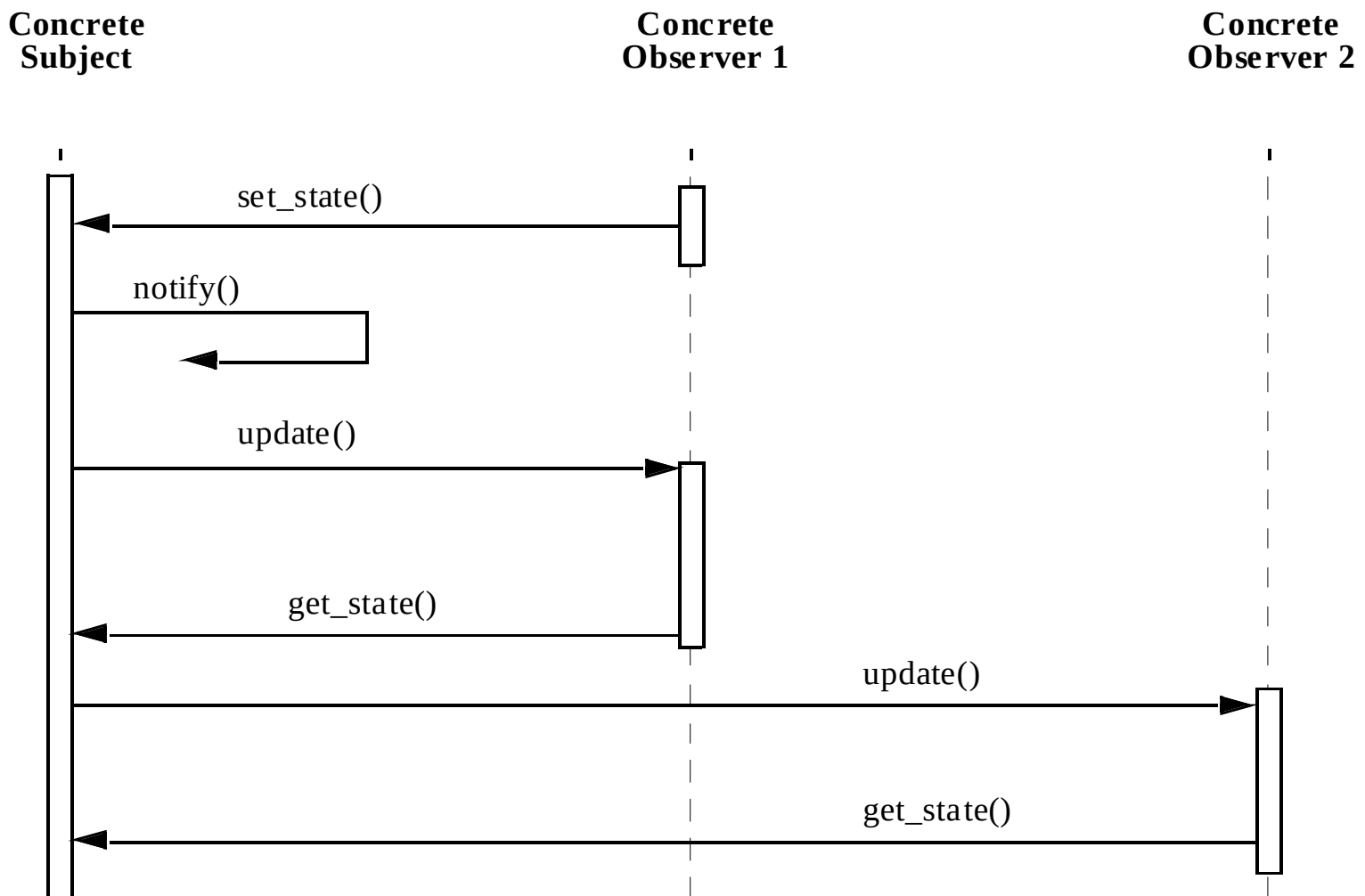
A Distributed File System



Structure of the Observer Pattern



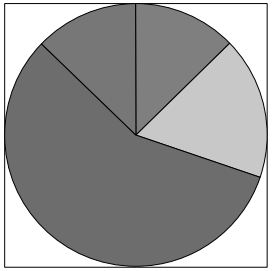
Collaborations in the Observer Pattern

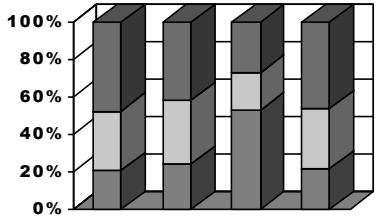


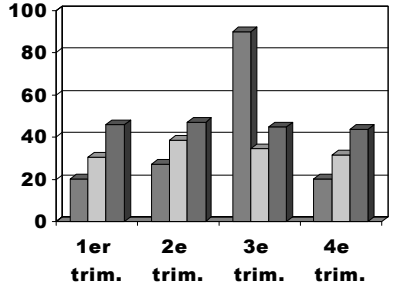
Another Problem…

- Any number of views on a Data Table in a windowing system…
  - close, open views at will…
  - change the data from any view
    - … and the other are updated

	1er trim.	2e trim.	3e trim.	4e trim.
Est	20,4	27,4	90	20,4
Ouest	30,6	38,6	34,6	31,6
Nord	45,9	46,9	45	43,9



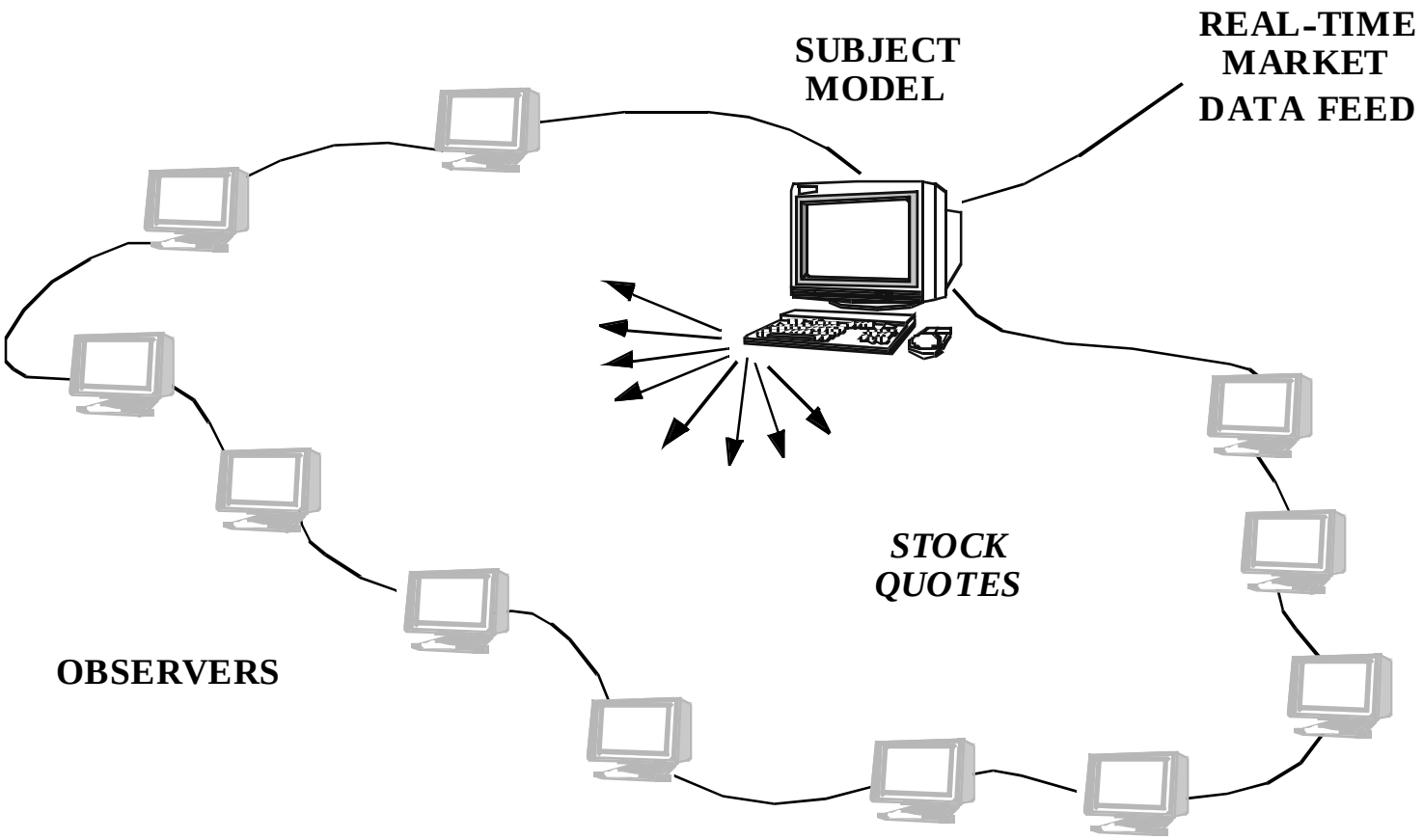




Observer

- Objectif
  - Définir une dépendance de un-vers-plusieurs entre les objets de façon à ce que, quand un objet change d'état, tous ses dépendants sont informés et mis à jour automatiquement
- Contraintes clés
  - Il peut y avoir plusieurs observateurs
  - Chaque observateur peut réagir différemment à la même information
  - Le sujet devrait être aussi découplé que possible des observateurs (ajout/suppr. dynamique d'observateurs)

Yet Another Problem...



3. Using Design Patterns

In a Software Engineering Context...

What Design Patterns are all about

- As much about problems as about solutions
  - pairs problem/solution in a context
- Not about classes & objects but collaborations
- About non-functional forces
  - reusability, portability, and extensibility…
- Embody architectural know-how of experts

Key Points of Design Patterns

- Identification of reusable micro-architectures
  - codifying good design
  - suitable for classification (see GoF's catalog)
- Definition of a vocabulary for thinking about design at a higher level of abstraction
    - Analogy with chess playing (borrowed from Doug Schmidt)

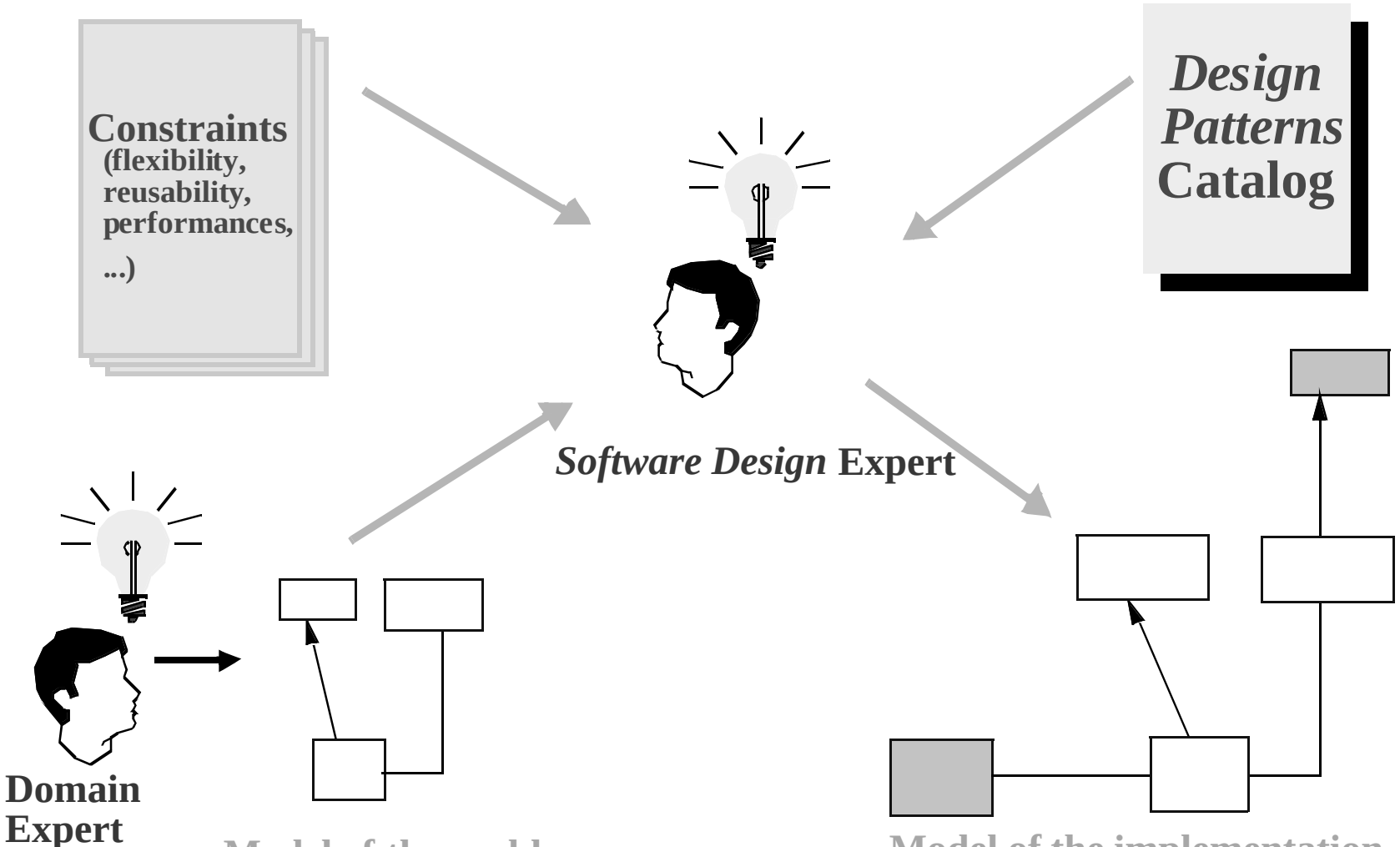
Becoming a Chess Master

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn principles
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- However, to become a master of chess, one must study the games of other masters
  - These games contain patterns that must be understood, memorized, and applied repeatedly
- There are thousands upon thousands of these patterns

Becoming a Master Software Designer

- First one learns the rules
  - e.g., the algorithms, data structures and languages of software
- Later, one learns the principles of software design
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- But to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are thousands upon thousands of these patterns

Design Patterns in the Software Life-Cycle



4. Describing Design Patterns

In a Software Engineering Context...

Interest of Documenting Design Patterns

- Communication of architectural knowledge among developers
- Provide a common vocabulary for common design structures
  - Reduce complexity
  - Enhance expressiveness, abstractness
- Distill and disseminate experience
  - Avoid development traps and pitfalls that are usually learned only by experience

Interest of Documenting Design Patterns (cont.)

- Improve documentation
  - Capture and preserve design information
  - Articulate design decisions concisely
- Build a Pattern Language
  - A cohesive collection of patterns that forms a vocabulary for understanding and communicating ideas

- Need of a more or less standard form...

Design Patterns (Alexandrian Format)

- Name
- Problem & Context
- Force(s) addressed
- Solution (structure and collaborations)
- Examples
- Positive & negative consequences of use
- Rationale
- Related Patterns & Known Uses

Design Pattern Descriptions (GoF Format)

- Name (& Aliases)
- Intent
- Motivation & Applicability
- Structure
- Participants & Collaborations
- Consequences
- Implementation, Sample Code and Usage
- Known Uses & Related Patterns

5. GoF Design Patterns

- Creational patterns
  - Deal with initializing and configuring classes and objects
- Structural patterns
  - Deal with decoupling interface and implementation of classes and objects
- Behavioral patterns
  - Deal with dynamic interactions among societies of classes and objects

Creational Patterns

- Abstract Factory
  - Interface for creating families of objects without specifying their concrete classes
- Builder
  - Factory for building complex objects incrementally
- Factory Method
  - Lets a class defer instantiation to subclasses.
- Prototype
  - Factory for cloning new instances from a prototype
- Singleton
  - Access to the unique instance of class

Structural Patterns (1)

- Adapter
  - Convert the interface of a class into another interface clients expect.
- Bridge
  - Decouple an abstraction from its implementations
- Composite
  - Recursive aggregations letting clients treat individual objects and compositions of objects uniformly
- Decorator
  - Extends an object functionalities dynamically.

Structural Patterns (2)

- Facade
  - Simple interface for a subsystem
- Flyweight
  - Efficiently sharing many Fine-Grained Objects
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns (1)

- Chain of Responsibility
  - Uncouple request sender from precise receiver on a chain.
- Command
  - Request reified as first-class object
- Interpreter
  - Language interpreter for a grammar
- Iterator
  - Sequential access to elements of any aggregate

Behavioral Patterns (2)

- Mediator
  - Manages interactions between objects
- Memento
  - Captures and restores object states (snapshot)
- Observer
  - Update observers automatically when a subject changes
- State
  - State reified as first-class object

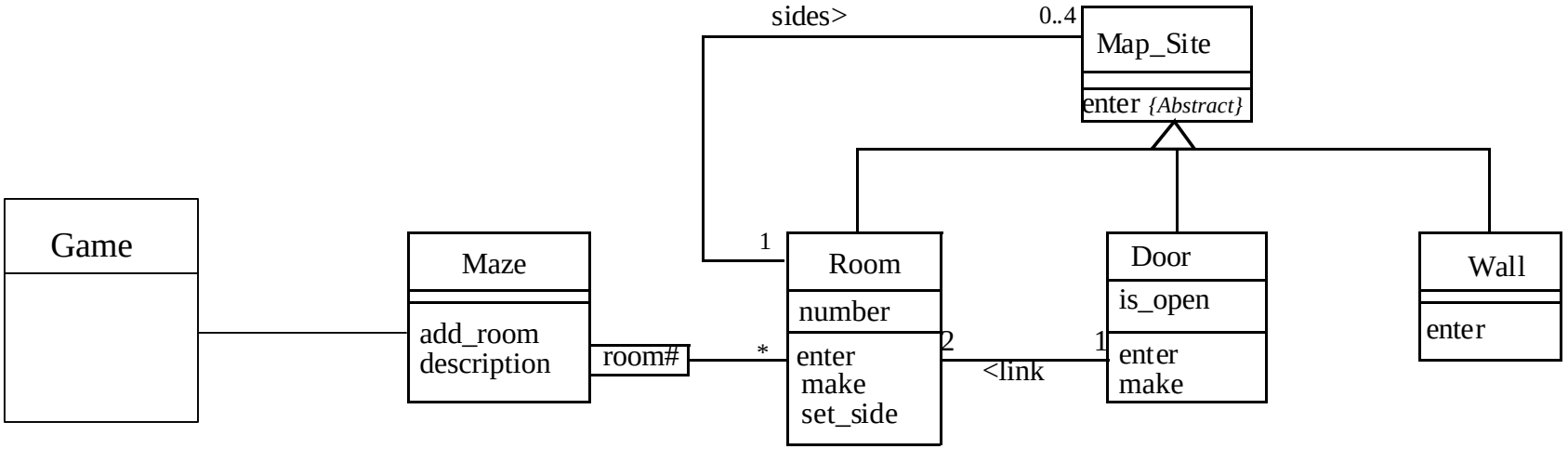
Behavioral Patterns (3)

- Strategy
  - Flexibly choose among interchangeable algorithms
- Template Method
  - Skeleton algo. with steps supplied in subclass
- Visitor
  - Add operations to a set of classes without modifying them each time.

6. More into the details...

With Abstract Factory

The Maze Example



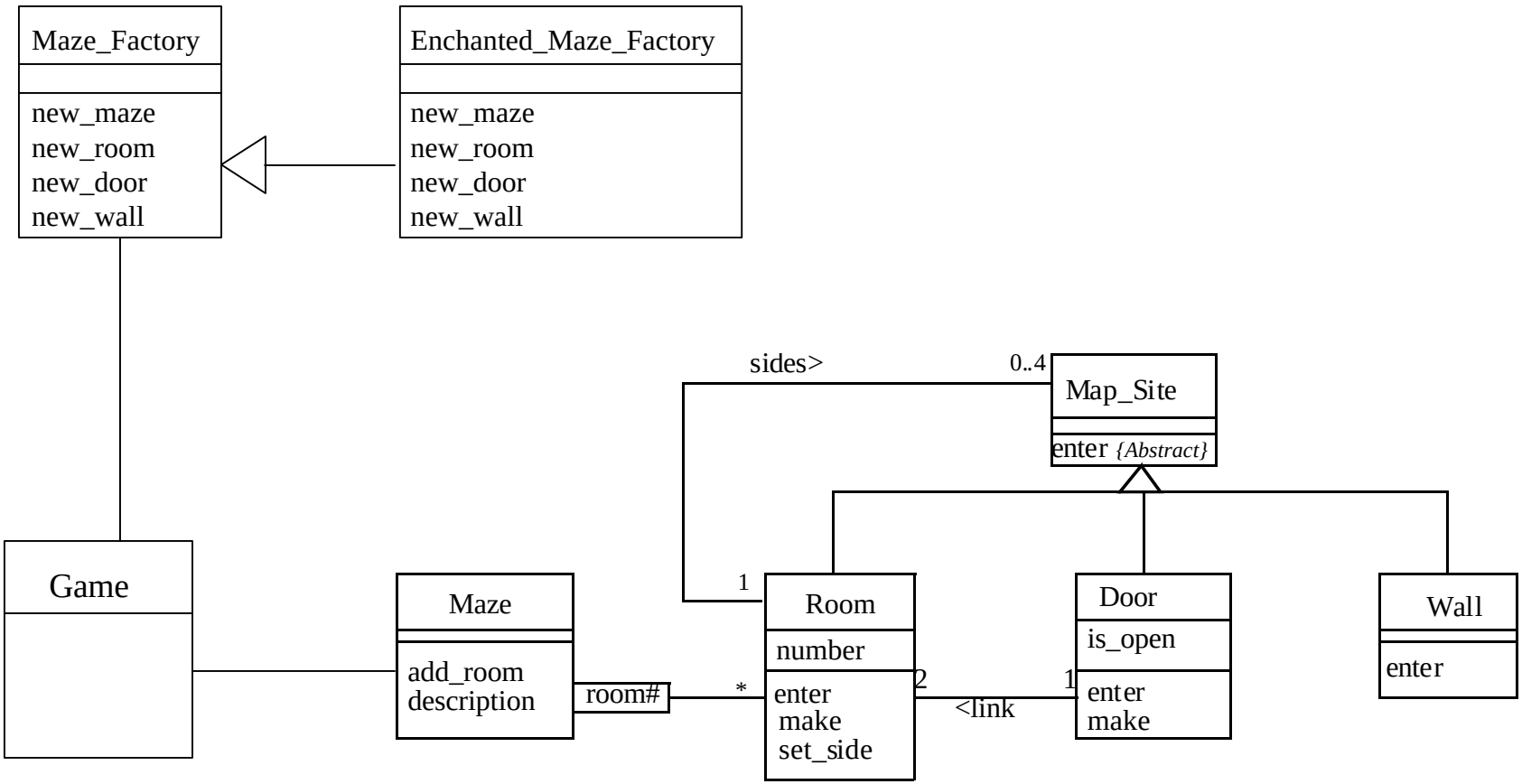
Create a basic game

```
class BASEGAME
creation make
feature -- Creation
  make is
    -- Program entry point
    do
      my_maze := create_maze
      my_maze.describe
    end -- make
feature
  create_maze : MAZE is
    -- 2 rooms connected through a door: [r1|r2]
    local
      r1, r2 : ROOM
      door : DOOR
      wall : WALL
    do
      !!Result.make_empty
      !!r1.make(1); !!r2.make(2); !!door.make(r1,r2)
      Result.add_room(r1); Result.add_room(r2)
      -- Now set up r1
      !!wall; r1.set_north_side(wall)
      r1.set_east_side(door)
      !!wall; r1.set_south_side(wall)
      !!wall; r1.set_west_side(wall)
      -- Now set up r2
      !!wall; r2.set_north_side(wall)
      !!wall; r2.set_east_side(wall)
      !!wall; r2.set_south_side(wall)
      r2.set_west_side(door)
    end -- create_maze
feature {NONE} -- Private
  my_maze : MAZE
end -- BASEGAME
```

Evolution

- I want the same maze, but with specialized rooms (enchanted)
- Do I need to re-write the basic game?
  - danger of cut and paste...
- Use an Abstract Factory
  - for creating maze components

With Abstract Factory



Maze Factory

```
class MAZE_FACTORY
inherit
  ABSTRACT_FACTORY
      rename new_product as new_maze
      redefine new_maze end;
feature {ANY} -- Public
  new_maze : MAZE is
    do
      !!Result.make_empty
    end -- new_maze
  new_wall : WALL is
    do
      !!Result
    ensure created: Result /= Void
    end -- new_wall
  new_room (number : INTEGER) : ROOM is
    do
      !!Result.make(number)
    ensure created: Result /= Void
    end -- new_room
  new_door (r1, r2 : ROOM) : DOOR is
    do
      !!Result.make(r1,r2)
    ensure created: Result /= Void
    end -- new_door
end -- MAZE_FACTORY
```

```
class ENCHANTED_MAZE_FACTORY
inherit
  MAZE_FACTORY
      redefine new_room, new_door end; -- (co-variant redefinition)
feature {ANY} -- Public
  new_room (number : INTEGER) : ENCHANTED_ROOM is
    -- Creates an ENCHANTED_ROOM
    do
      cast_a_spell
      !!Result.make(number,last_spell_cast)
    end -- new_room
  new_door (r1, r2 : ROOM) : LOCKED_DOOR is
    do
      !!Result.make(r1,r2)
    end -- new_door
feature {NONE} -- Private
  last_spell_cast : SPELL
  cast_a_spell is do !!last_spell_cast end
end -- ENCHANTED_MAZE_FACTORY
```

Game with Abstract Factory

```
class GAME_WITH_ABSTRACT_FACTORY
creation make
feature -- Creation
  make is
    -- Program entry point
    local maze_factory : MAZE_FACTORY
    do
      !!maze_factory
      -- A normal MAZE
      my_maze := create_maze (maze_factory)
      my_maze.describe
      !ENCHANTED_MAZE_FACTORY!maze_factory
      my_maze := create_maze (maze_factory)
       -- A MAZE with enchanted ROOMs
      my_maze.describe
    end -- make
feature
  create_maze (factory : MAZE_FACTORY) : MAZE is
    -- Create a new maze
    local
      r1, r2: ROOM
      door : DOOR
    do
      Result := factory.new_maze
      r1 := factory.new_room(1); r2 := factory.new_room(2)
      door := factory.new_door(r1,r2)
      Result.add_room(r1); Result.add_room(r2)
      -- Now set up r1
      r1.set_north_side(factory.new_wall)
      r1.set_east_side(door)
      r1.set_south_side(factory.new_wall)
      r1.set_west_side(factory.new_wall)
      -- Now set up r2
      r2.set_north_side(factory.new_wall)
      r2.set_east_side(factory.new_wall)
      r2.set_south_side(factory.new_wall)
      r2.set_west_side(door)
    end -- create_maze
feature {NONE} -- Private
  my_maze : MAZE
end -- GAME_WITH_ABSTRACT_FACTORY
```

Conclusion

- Design Patterns have raised the level at which most OO designs are done now
- Useful thing in the designer's toolkit
  - But no silver bullet…
- Ongoing systematic efforts to catalog DPs
  - maybe towards the software engineering manual.

Further References

- Design Patterns and Contracts ……..
  - Addison-Wesley 1999. ISBN 0-201-30959-9
- Design Patterns: Elements of Object-Oriented Software
  - Gamma, Helm, Johnson, Vlissides. Addison Wesley, 1995
- Pattern Oriented Software Architecture, A System of Patterns
  - Buschmann, Meunier, Sommerland, Stal. Wiley & Sons, 1996
- Pattern Languages of Program Design
  - PLoPD 1, 2, 3 & 4. Addison-Wesley
- http://hillside.net/patterns

And...

Object-Oriented Software Engineering with Eiffel

by Jean-Marc Jézéquel

Addison-Wesley Eiffel in Practice Series
ISBN 0-201-63381-7 * Paperback
368 pages * ©1996
http://www.irisa.fr/pampa/EPEE/book.html