

Design Patterns

Software Engineering for Engineers
Summer 2009

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Outline

- Design Patterns
 - Usefulness of design patterns, Design Pattern Categories
- Patterns covered in the lecture
 - Composite Pattern: Modeling of dynamic aggregates
 - Adapter Pattern: Interface to old systems (legacy systems)
 - Observer Pattern: Maintain consistency across redundant state, also called Publisher-Subscriber
 - Bridge Pattern: Interfacing to existing and future systems
 - Façade Pattern: Interfacing to subsystems
 - Proxy Pattern: Reduces the cost of accessing objects
 - Strategy Pattern: Interface to a task implemented by different algorithms
 - Not covered in the lecture, but in the backup slides: Template, Abstract Factory, Builder.

Design pattern

A design pattern is...

...a template solution to a recurring design problem

- Look before re-inventing the wheel just one more time

...an example of *modifiable* design

- Learning to design starts by studying other designs

...reusable design knowledge

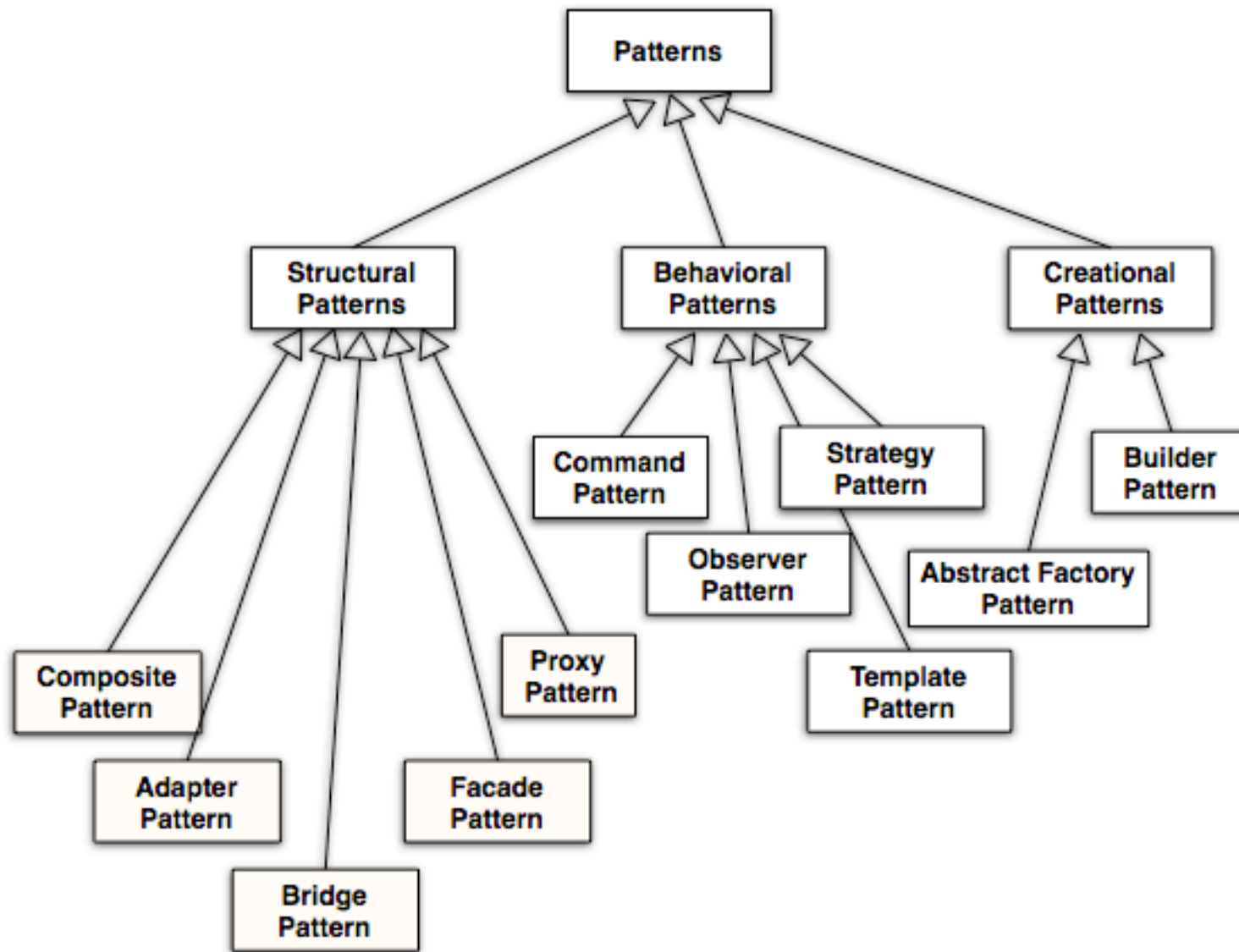
- 7+-2 classes and their associations
- Often actually more 5+-2 classes.

What makes Design Patterns Good?

- They are generalizations of design knowledge from existing systems
- They provide a shared vocabulary to designers
- They provide examples of reusable designs
 - Inheritance (abstract classes)
 - Delegation (or aggregation)

Categorization of Design Patterns

- **Structural Patterns**
 - reduce coupling between two or more classes
 - introduce an abstract class to enable future extensions
 - encapsulate complex structures
- **Behavioral Patterns**
 - allow a choice between algorithms and the assignment of responsibilities to objects (“Who does what?”)
 - characterize complex control flows that are difficult to follow at runtime
- **Creational Patterns**
 - allow to abstract from complex instantiation processes
 - Make the system independent from the way its objects are created, composed and represented.






A Game: Get-15

- Start with the nine numbers 1,2,3,4, 5, 6, 7, 8 and 9.
- You and your opponent take alternate turns, each taking a number
- Each number can be taken only once: If you opponent has selected a number, you cannot also take it.
- The first person to have any three numbers that total 15 wins the game.

- Example:

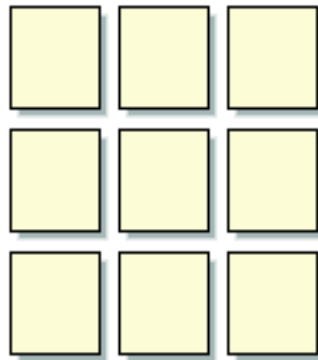
You: 1 5 3 8

Opponent:  9   Opponent Wins!

Characteristics of Get-15

- Hard to play,
- The game is especially hard, if you are not allowed to write anything done.
- Why?
 - All the numbers need to be scanned to see if you have won/lost
 - It is hard to see what the opponent will take if you take a certain number
 - The choice of the number depends on all the previous numbers
 - Not easy to devise an simple strategy

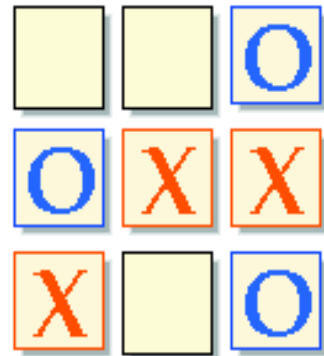
Another Game: Tic-Tac-Toe



YOU ARE 

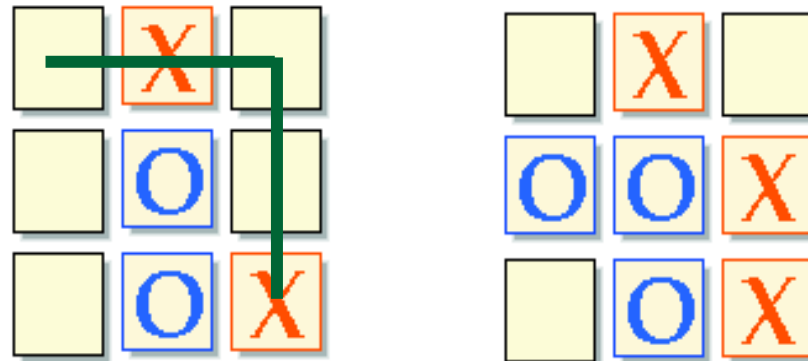
Source: <http://boulter.com/ttt/index.cgi>

A Draw Situation

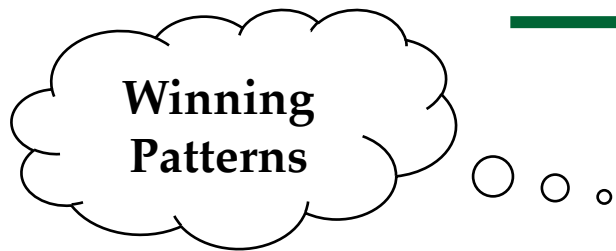
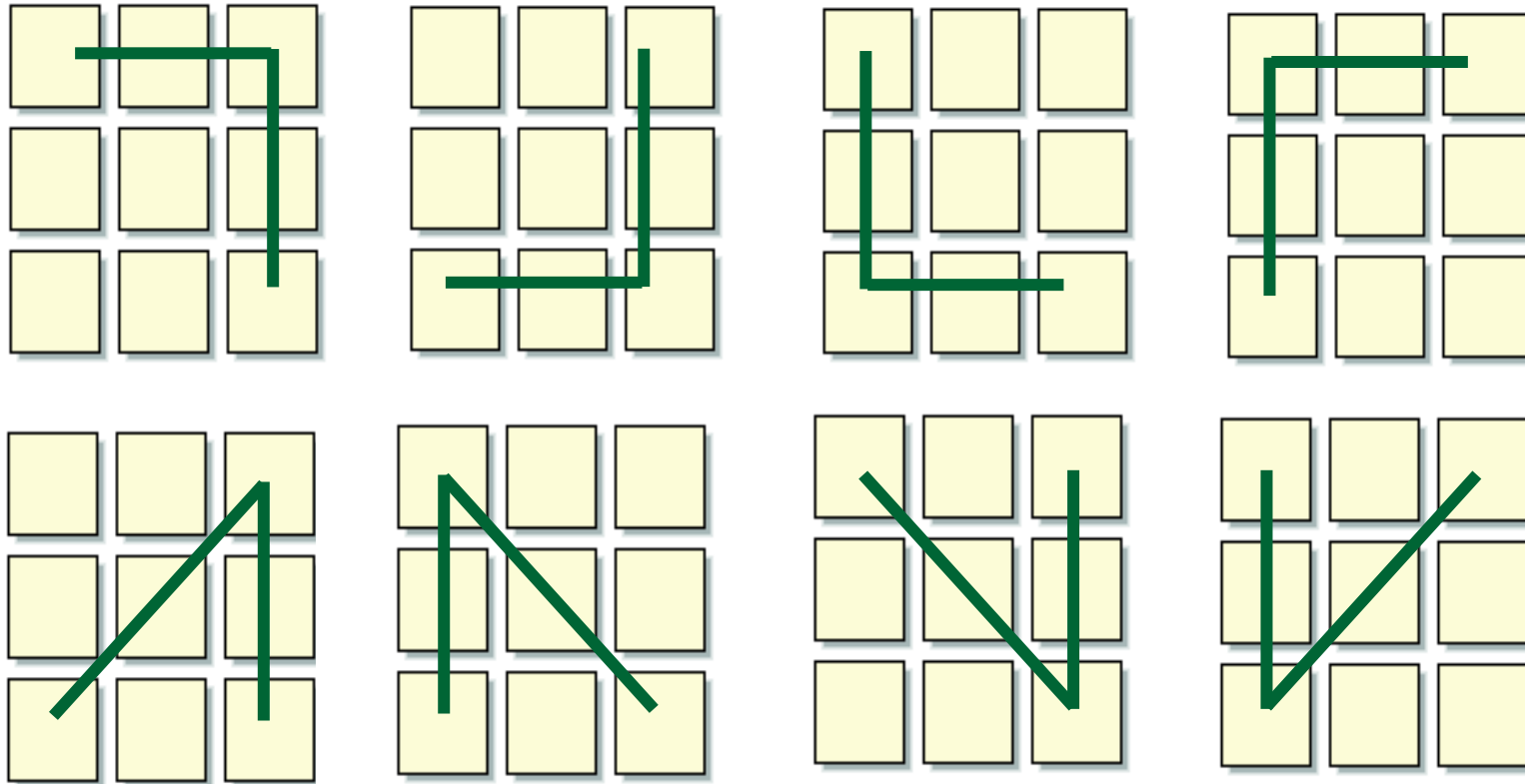


YOU ARE 

Strategy for determining a winning move

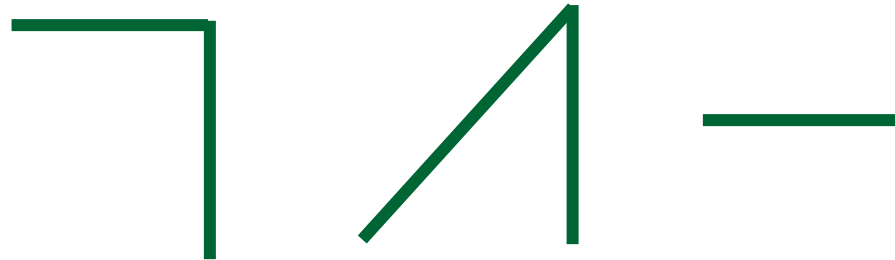


Winning Situations for Tic-Tac-Toe



Tic-Tac-Toe is “Easy”

- Why? Reduction of complexity through patterns and symmetry
- **Patterns:** Knowing the following three patterns, the player can anticipate the opponents move

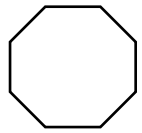


- **Symmetry:**
 - The player needs to remember only these three patterns to deal with 8 different game situations
 - The player needs to memorize only 3 opening moves and their responses

Get-15 and Tic-Tac-Toe are identical problems

- Any Get-15 solution is a solution to a tic-tac-toe problem
- Any tic-tac-toe solution is a solution to a Get-15 problem
- To see the relationship between the two games, we digits into the following pattern

8	1	6
3	5	7
4	9	2



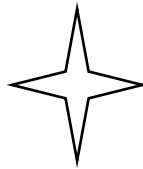
You:

1

5

3

8



Opponent:

6

9

7

2

8	1	6
3	5	7
4	9	2

4			

- During object modeling we do many transformations and changes to the object model
- It is important to make sure the object model stays simple!
- Design patterns are used to keep system models simple (and reusable).

Modeling Heuristics

- Modeling must address our mental limitations:
 - Our short-term memory has only limited capacity (7+-2)
- Good Models deal with this limitation, because they
 - Do not tax the mind
 - A good model requires a small mental effort
 - Reduce complexity
 - Turn complex tasks into easy ones (choice of representation)
 - Use of symmetries
 - Use abstractions
 - Taxonomies
 - Have organizational structure:
 - Memory limitations are overcome with an appropriate representation ("natural model").

What is common between these definitions?

Recursion

- Definition Software System
 - A software system consists of subsystems which are either other subsystems or collection of classes

- Definition Software Lifecycle
 - A software lifecycle consists of a set of development activities which are either other activities or collection of tasks.

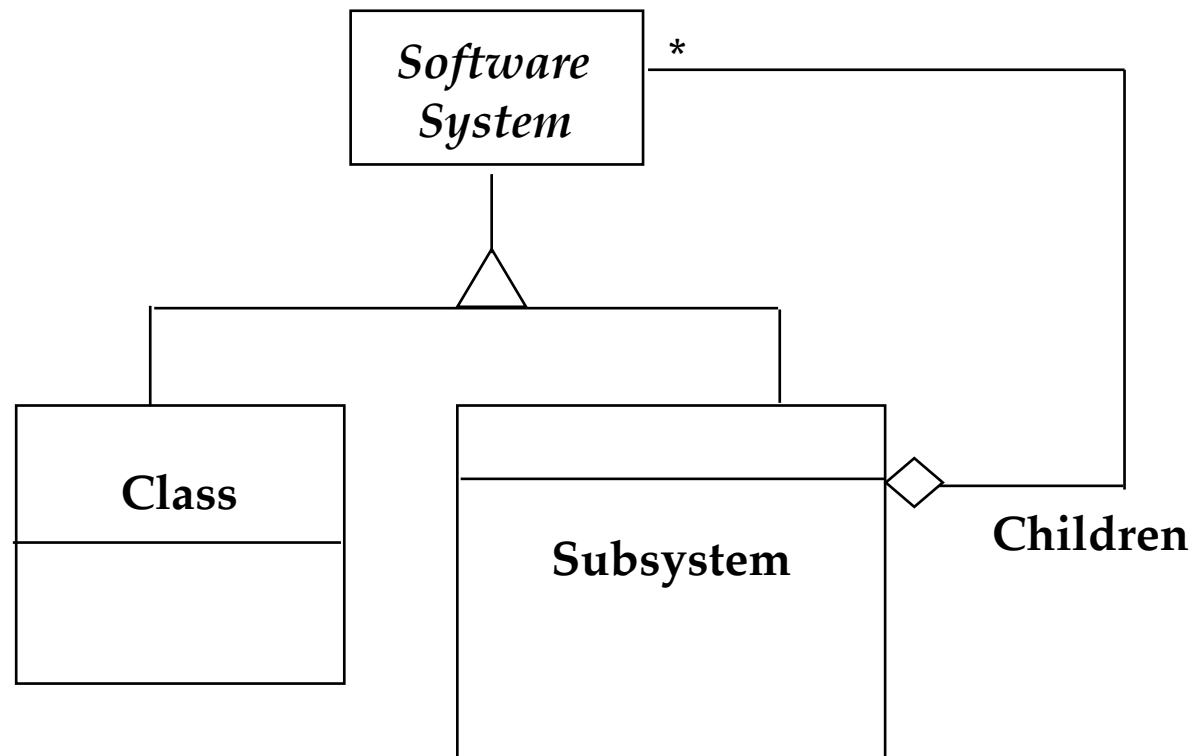
Recursion

- **Recursion**
 - An abstraction being defined is used within its own definition
 - More general: Description of an abstraction based on self-similarity.

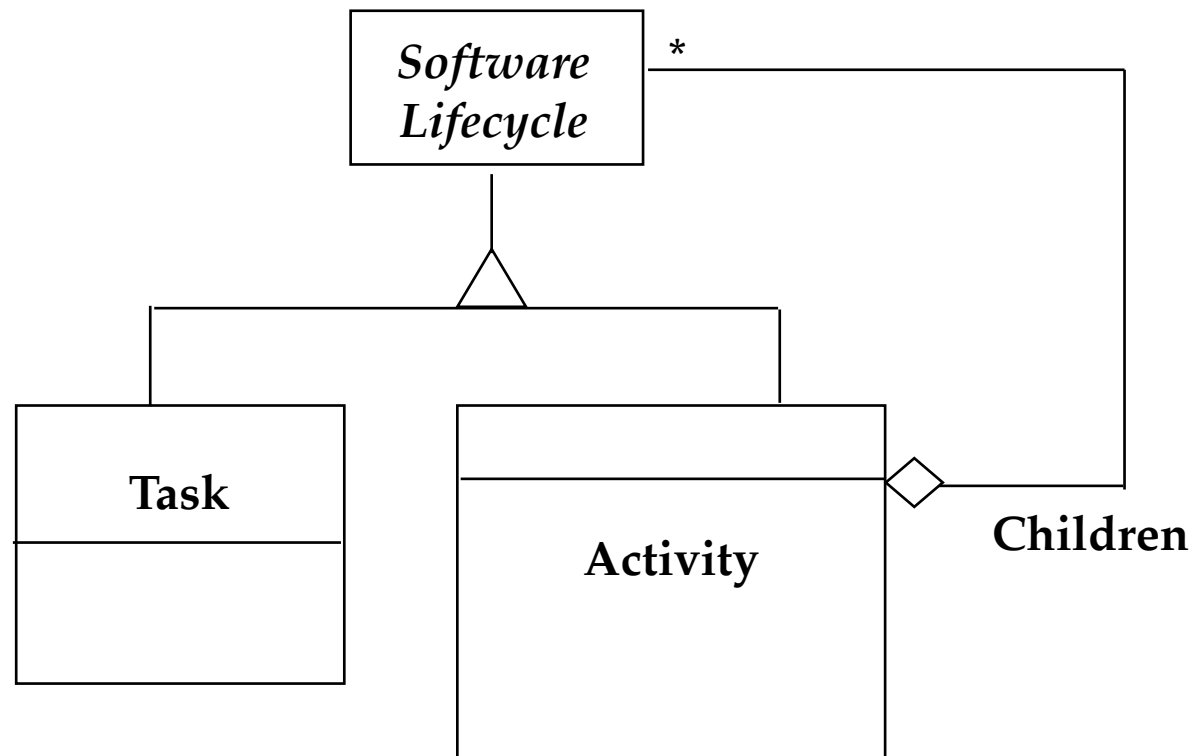
What is common between these definitions?

- Definition Software System
 - A software system consists of subsystems which are either other subsystems or collection of classes
 - **Composite:** Subsystem
 - A software system consists of subsystems which consists of subsystems, which consists of subsystems, which...
 - **Base case:** Class
- Definition Software Lifecycle
 - The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
 - **Composite:** Activity
 - The software lifecycle consists of activities which consist of activities, which consist of activities, which....
 - **Base case:** Task.

Modeling a Software System

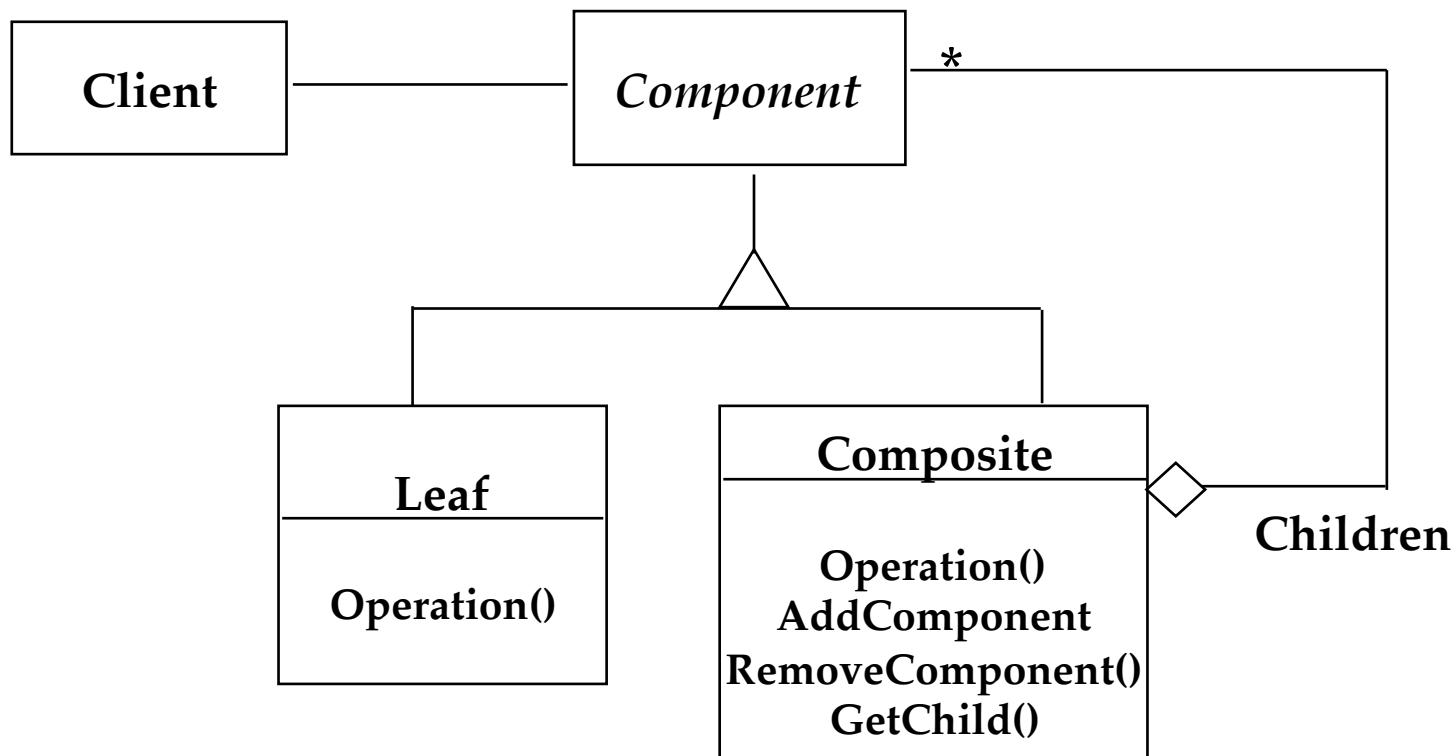


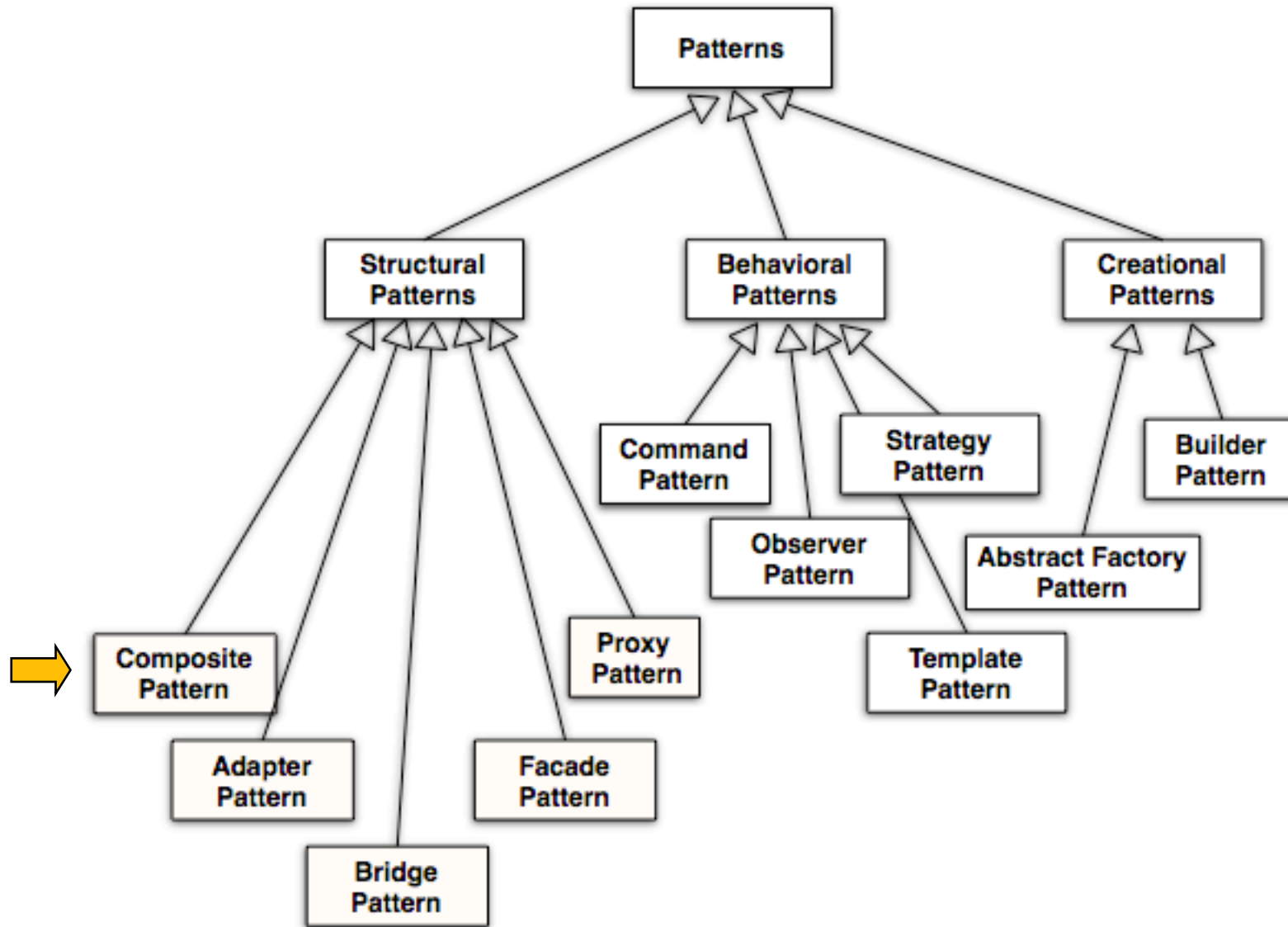
Modeling the Software Lifecycle



Introducing the Composite Pattern

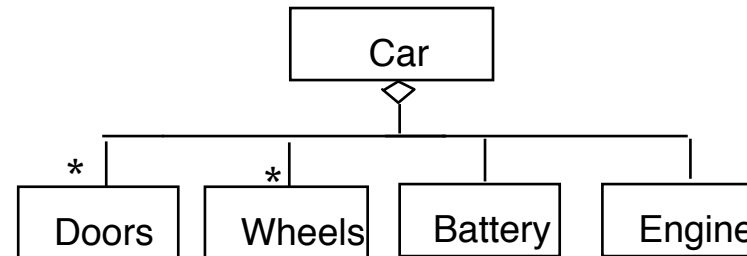
- The pattern models tree structures that represent hierarchies of objects with arbitrary depth and width
- The Composite Pattern lets a client treat individual objects and compositions of these objects uniformly



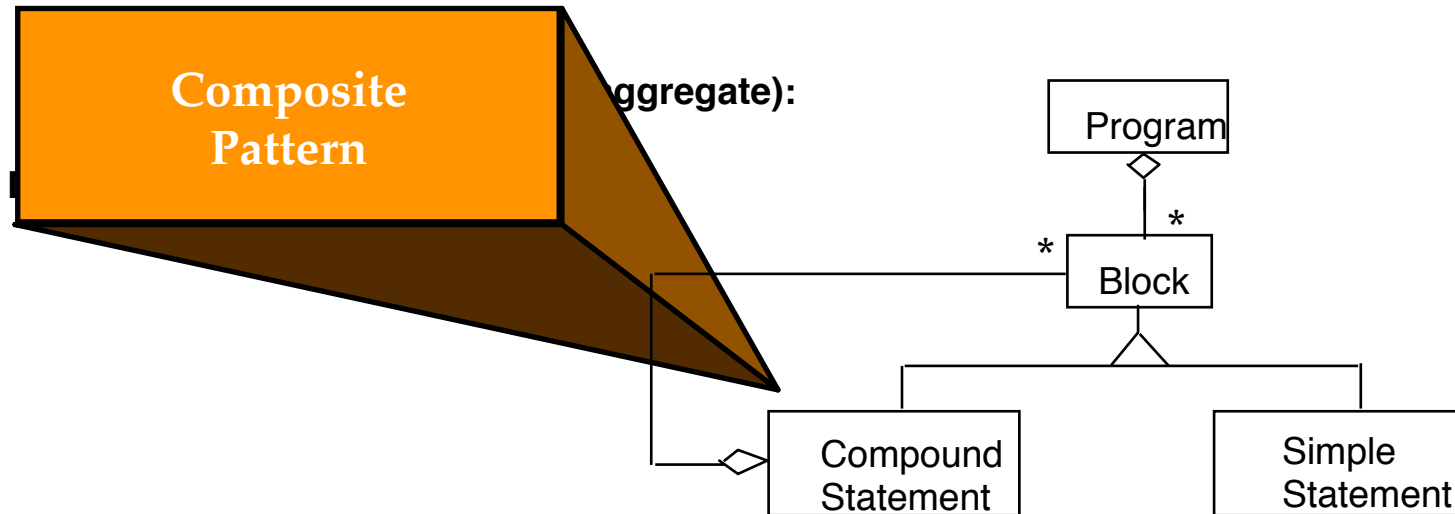
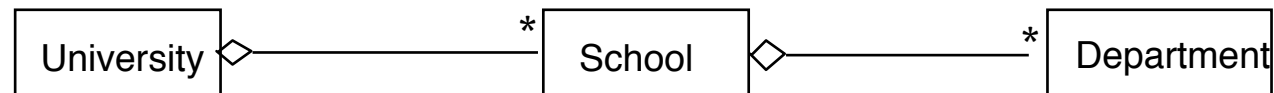


The Composite Patterns models dynamic aggregates

Fixed Structure:

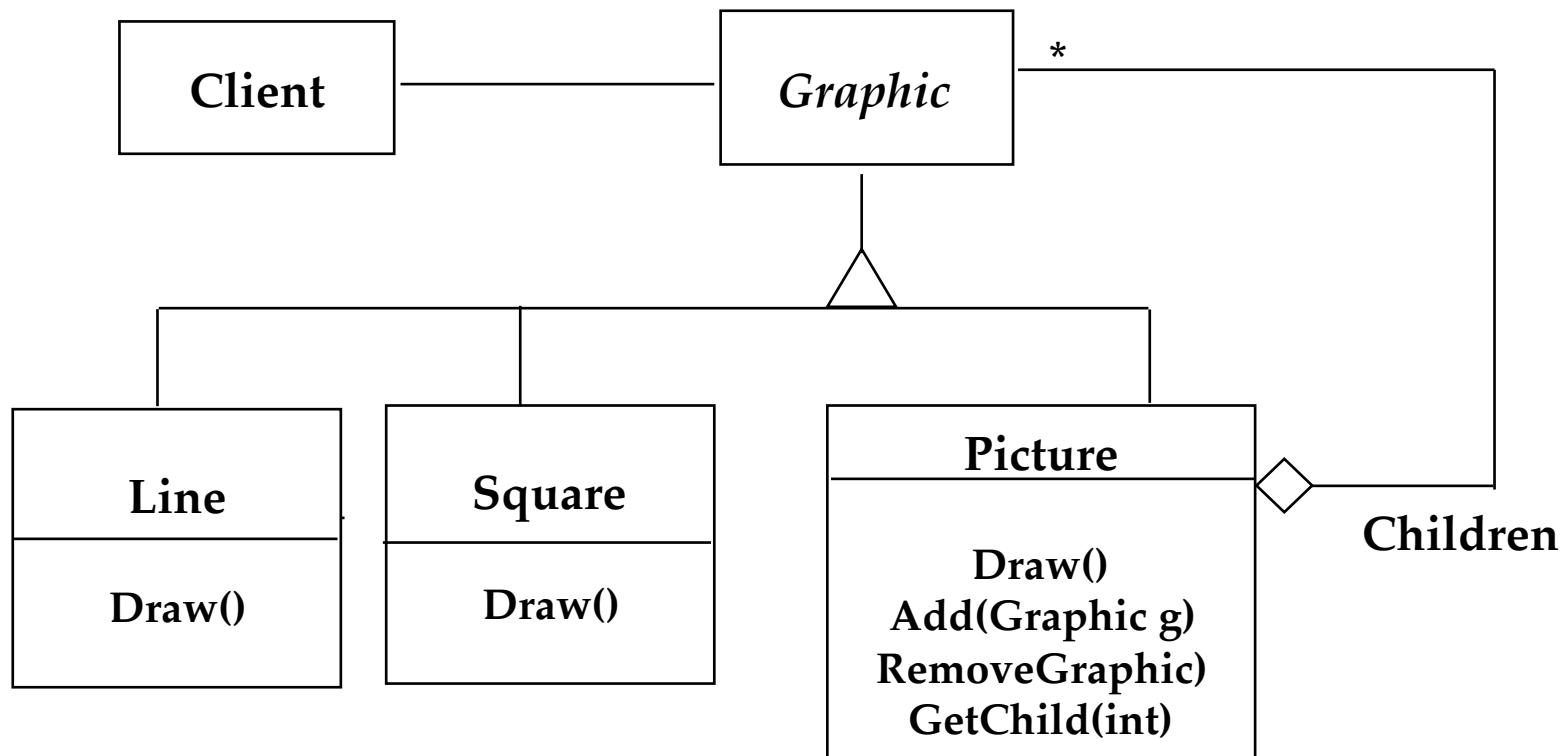


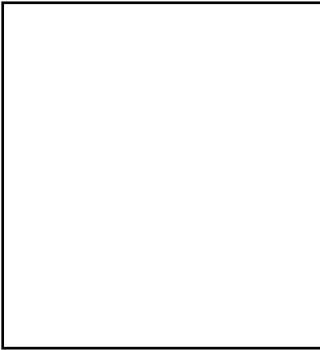
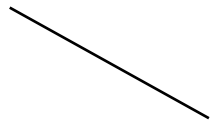
Organization Chart (variable aggregate):

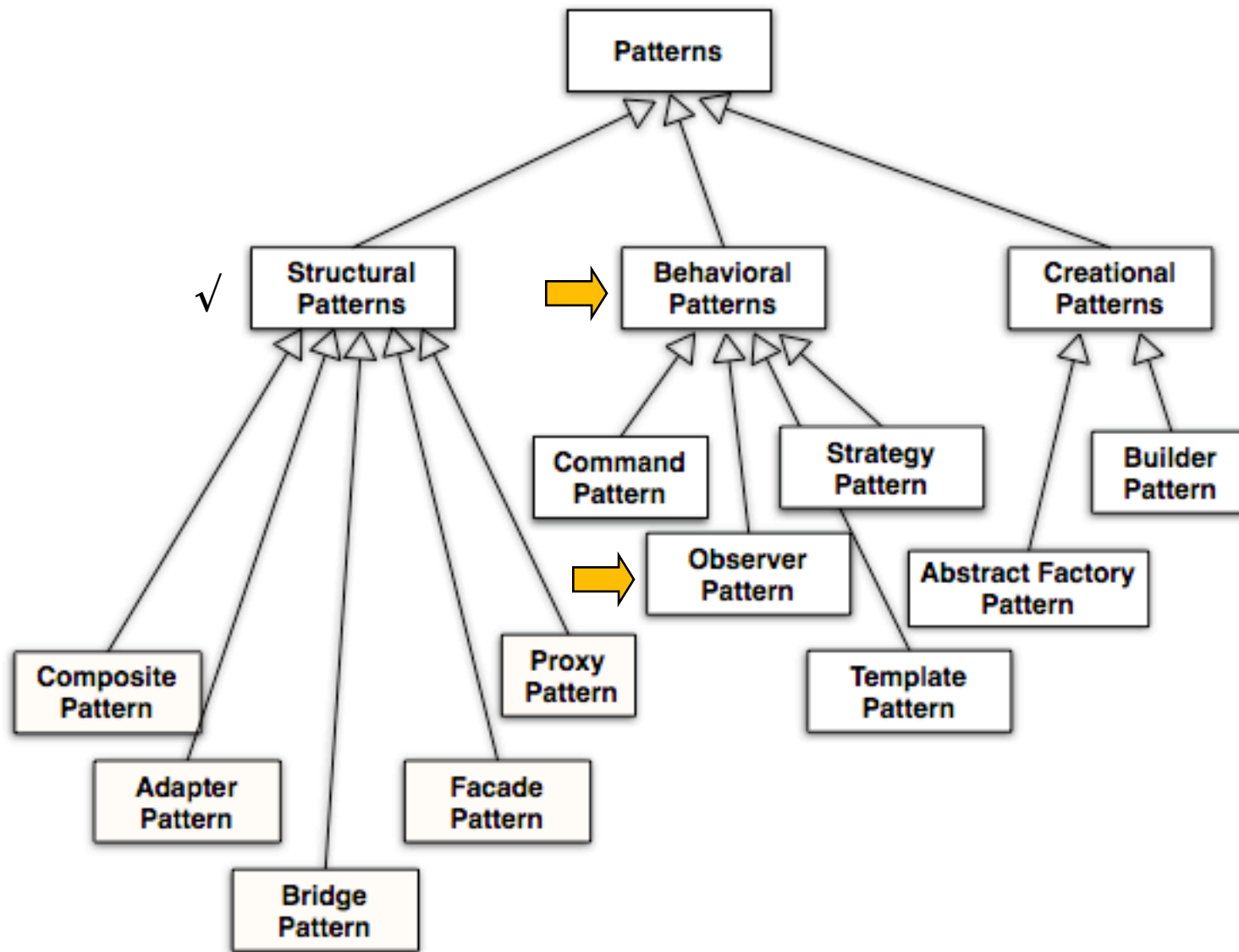


Graphic Applications also Composite Patterns

- The *Graphic* Class represents both primitives (Line, Square) and their containers (Picture)

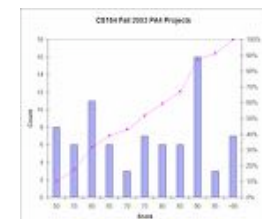
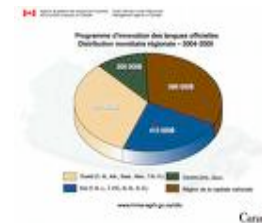
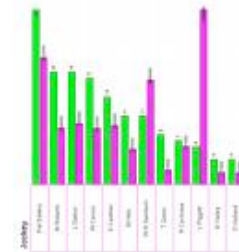
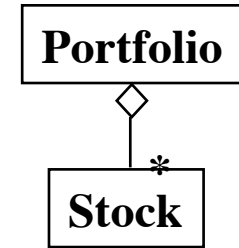






Observer Pattern Motivation

- Problem:
 - We have an object that changes its state quite often
 - Example: A Portfolio of stocks
 - We want to provide multiple views of the current state of the portfolio
 - Example: Histogram view, pie chart view, time line view, alarm
- Requirements:
 - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
 - The system design should be highly extensible
 - It should be possible to add new views without having to recompile the observed object or the existing views.

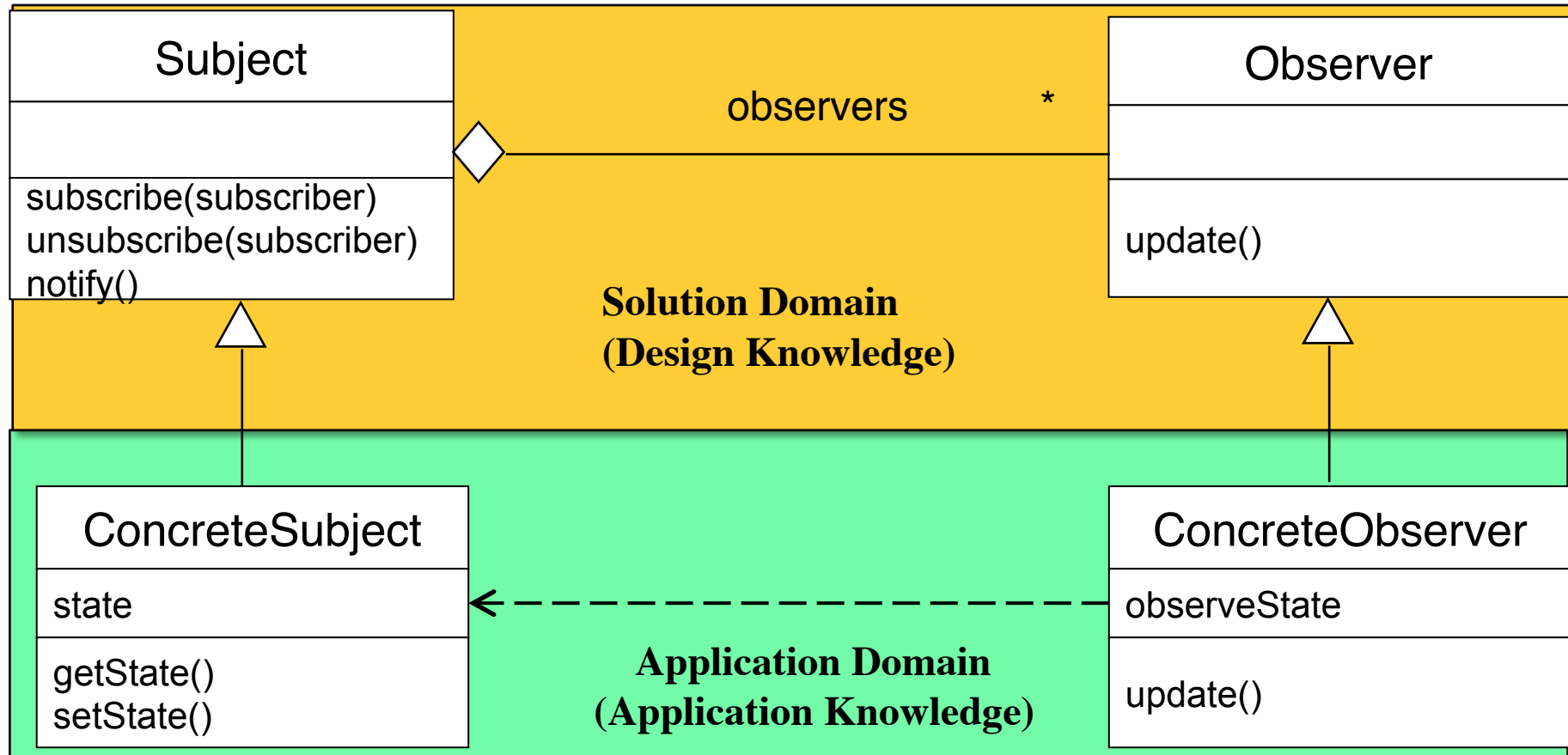


Example: The File Name of a Presentation

3 Possibilities to change the File Name

The screenshot shows a Mac OS X desktop environment. On the left is the 'Info' window for the file 'L9_DesignPatterns2.ppt', showing details like size (3.7 MB), location, and permissions. The main window displays a file list in 'List View' with columns for Name and Date Modified. The file 'L9_DesignPatterns2.ppt' is selected. Below the file list is a preview window showing a PowerPoint slide titled 'Example: The File Name of a Presentation'. The slide content is: 'What happens if I change the file name of this presentation in List View to foo?'. The slide footer includes '©2007 Bernd Brügge', 'Introduction Into Software Engineering, Summer 2007', and '59'. Three red callouts are present: 'List View' points to the file list, 'Powerpoint View' points to the slide preview, and 'Info View' points to the file information sidebar.

Observer Pattern: Decouple Object from its Views



- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object
 - The **state** is contained in the subclass **ConcreteSubject**
 - The state can be **obtained and set** by subclasses of type **ConcreteObserver**.