Using Design Patterns

- A brief introduction to Design Patterns
- Example: The Composite Pattern
  - Definition
  - Examples
- Example: The Publisher-Subscriber Pattern
  - Definition
  - Example
- How Together can help

Re-using experience

- How do we decide what makes **a good object model**?
  - What guidance do we have about the kind of object that we expect to find in a object model?
  - It is always much easier to solve a problem if you have previously solved similar problems or, at least, have access to a solution to a similar problem.
- An expert is a person with previous experience that they are able to use when solving problems.
  - How can we enable (less experienced) designers to benefit from expert experience?
- A major topic in object modelling is **design patterns**

Design Patterns

- UML designers' definition:

  "A pattern is a common solution to a common problem in a given context."
- Wikipedia's definition:

  "In **software engineering**, a design pattern is a **general reusable solution** to a **commonly occurring problem** in software design. ... is **not a finished** design ... it ... is a description or **template** for how to solve a problem that can be used in many different situations.

  Object-oriented design patterns typically show relationships and interactions between classes or objects."

Design Patterns

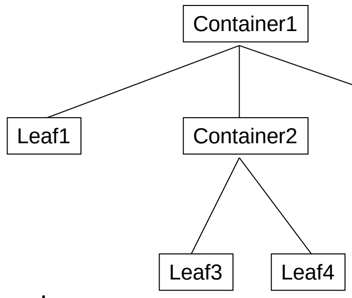
- *[1995] Gamma, Helm, Johnson & Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley*
  - This book made design patterns popular in computer science
  - Authors: the "Gang of Four"
- Main classification:
  - **Creational** (creating objects, eg. Factory Method),
  - **Structural** (class/objects structure, eg. Adapter, Composite)
  - **Behavioural Patterns** (interaction, eg. Publisher-Subscriber)
- Lots of documentation online...
- Pattern *catalogues* have been produced
- Look at a few examples of patterns...

The Composite Pattern

- Intent: To manage part-whole *hierarchies*
  - It uses tree structures
- Motivation:
  - Similar objects/components can be composed in a hierarchical structure
  - Single instances and groups of similar instances can be dealt with uniformly
- The idea:
  - We will consider a **`Container`** object that contains...
  - ... an arbitrary number of **`Component`** objects...
  - ... each of which might itself be a **`Container`**

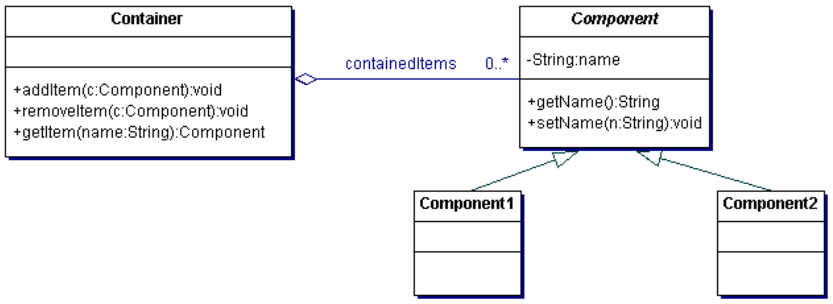
Composite Pattern - Example I

- Consider the following set of objects:



- Concrete examples:
  - An organization with divisions/subdivisions/... and undivided teams
  - A graphical design with groups/subgroups/... and elements
  - A file store with folders and files

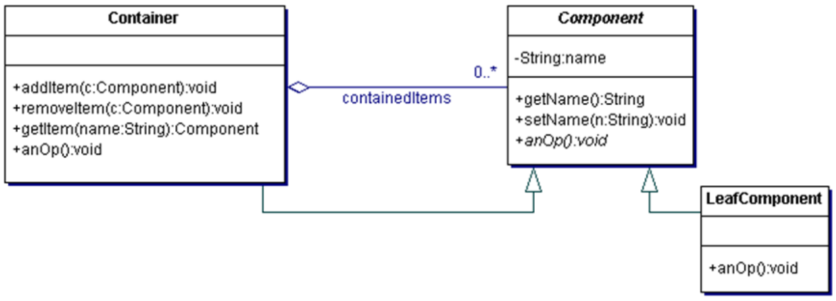
- Structure (first approximation):
  - Consider a **Container** object that contains …
  - … an arbitrary number of …
  - … potentially different kinds of **Component** objects
  - The contained objects are "uniform"



- Structure:
  - A **Container** object is an aggregate that can contain …
  - … uniform **Component** objects,
  - which are all instances of *subclasses* of the **Component** superclass
- Some of the subclasses are **LeafComponents**
  - Simple objects, no extension of the hierarchy
- However, some of the **Component** objects **may themselves be** **Container** objects
  - **Container** is therefore a *subclass* of class **Component**
  - These extend the hierarchy to further contained structure

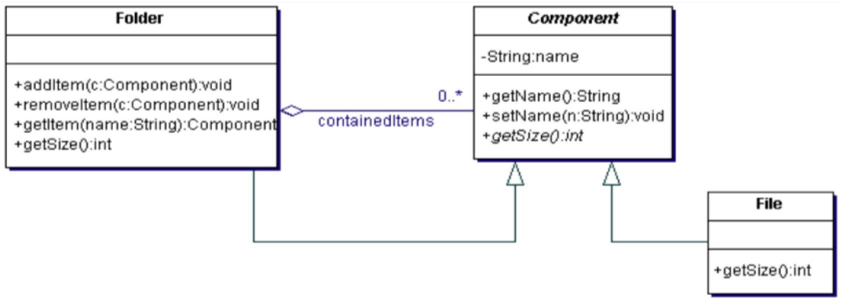
Composite Pattern – final structure

- A **Container** object is an aggregate that can contain …
  - … uniform **Component** objects…
  - … each of which can be either a **LeafComponent** or a **Container**
- **anOp** is introduced as typical operation offered in a specialized way by all **Components** – e.g. **getSize**, **print**



Composite Pattern - Example II

- An example of a Composite is a file store **Folder**:
  - It can contain different kinds of **File** objects.
  - BUT, a folder can also contain **Folder** objects.
  - Hence, a **Folder** is not just a **Container**, it is also a **Component**



- This pattern describes a not-so-natural structure in which the container, or aggregate, is a subclass of a component.
  - This pattern is useful in the design of systems for hierarchical structures such as the ones in the examples
  - That is the advantage of patterns; **once one person has invented and documented the pattern, we can all use it even though we might not have thought of it ourselves**
  - How are patterns created?
    Usual way is that lots of object models have been examined to identify common structures that occur in different contexts, then patterns are proposed, discussed, listed in a catalogue, ...

The Publisher-Subscriber Pattern

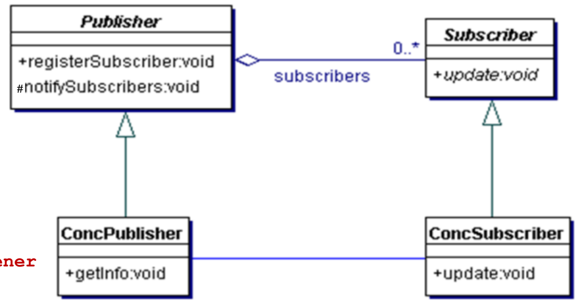
- Motivation (scenario):
  - Subscribers register with a publisher
  - When the publisher has some new information, all its subscribers are informed
  - The subscribers can then access the new information
- We have the abstract super-classes `Publisher` and `Subscriber` and the concrete subclasses `ConcPublisher` and `ConcSubscriber`
- Concrete example:
  - A Java Swing main program can register as a listener with a `JSlider`
  - When the `JSlider` is adjusted, it calls `stateChanged` in its listeners...
  - ... which can then interrogate the slider using `getValue`

•Structure:

- A **ConcSubscriber** object registers with a **ConcPublisher** object using the **registerSubscriber** operation defined in the superclass **Publisher**
- Each (**Conc**)**Publisher** object maintains a list of all the (**Conc**)**Subscriber** objects that have registered with it. Managing the list is done by methods already defined in the **Publisher** superclass

JComponent
listenerList

ChangeListener
stateChanged

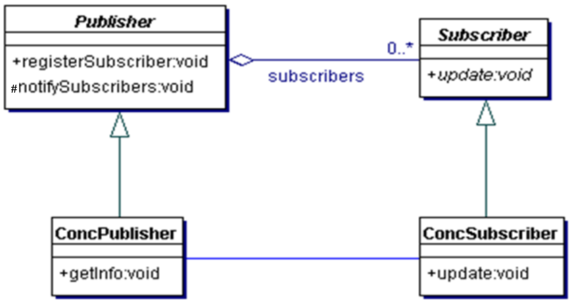


JSlider
addChangeListener
getValue

MyProgram
stateChanged

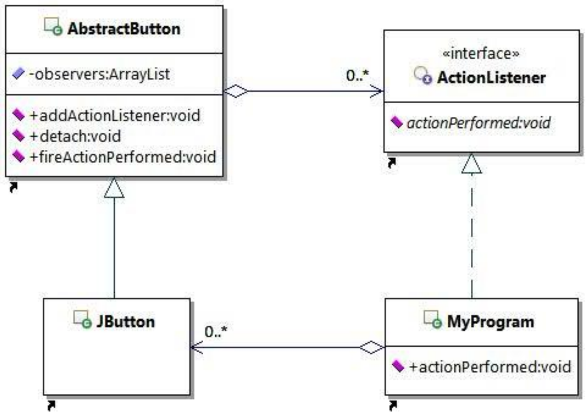
•Operation:

- When a **ConcPublisher** object has new information it calls its **notifySubscribers** method (from its superclass)…
- … which calls **update** in each of its registered (**Conc**)**Subscribers**
- Each **ConcSubscriber's update** can interrogate the **ConcPublisher** object by calling its **getInfo** operation



Publisher-Subscriber Example

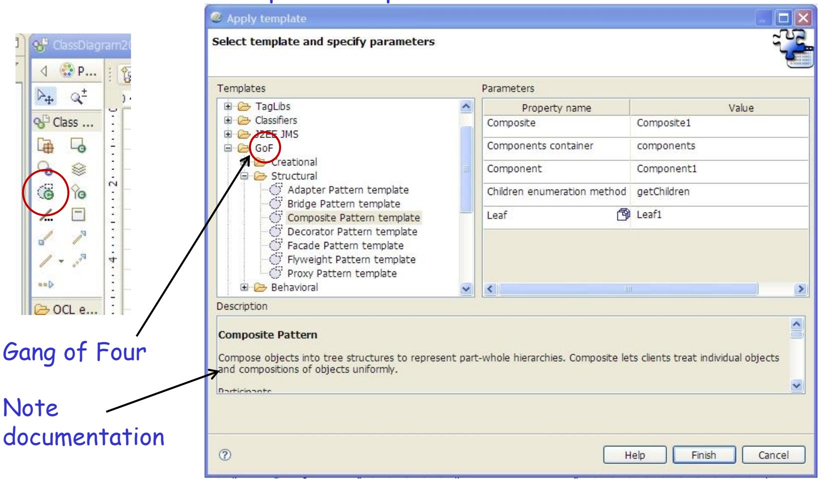
- An interactive graphical user interface:
  - A Java main program with Swing GUI JButtons



(Built using the Together wizard – next slides)

How Together can help

- The class diagram tools contain a "class by template" button
  - A template wizard pops up when a class is drawn
  - ... Which has pattern options



Property name	Value
Composite	Composite1
Components container	components
Component	Component1
Children enumeration method	getChildren
Leaf	Leaf1

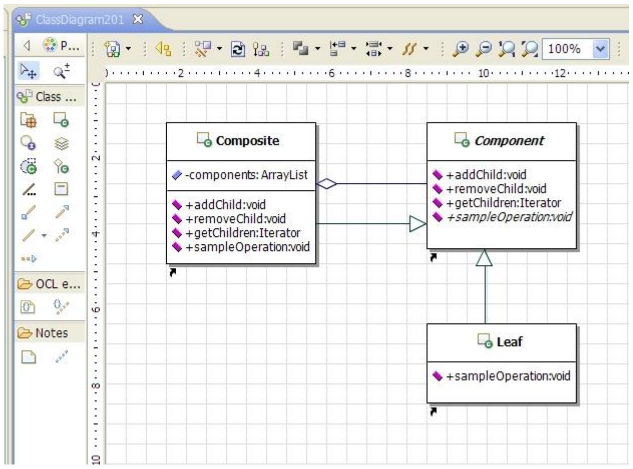
Composite Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

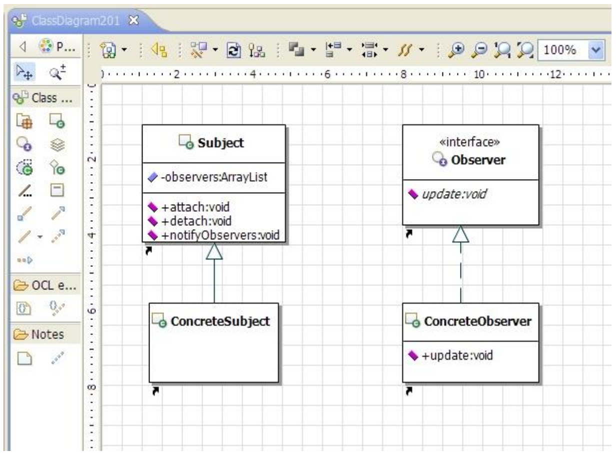
Gang of Four

Note documentation

- For example, selecting Composite Pattern Template, and accepting the default labelling gives:



- And, selecting Observer Pattern Template gives:



End of lecture