

Composite

A nice and common design pattern

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone



Outline

- Motivating examples
- Presentation of the Composite design pattern
- Discussions on Composite



File entry examples

Pharo.image

MOOC_Files/
Pharo.image
Pharo.changes

MOOC_Files/
src/
doc/
images/
Pharo.image
Pharo.changes



File entries

An entry is:

- a file
- or a folder with entries as children



Documents

A document is composed of:

- a title
- a table of contents
- chapters

A chapter is composed of:

- sections

A section is composed of:

- paragraphs
- figures
- lists
- sub-sections



Diagram

- A diagram is composed of elements
- An element is:
 - a circle
 - a segment
 - a text
 - a group of elements (i.e, diagram)



Now the question!

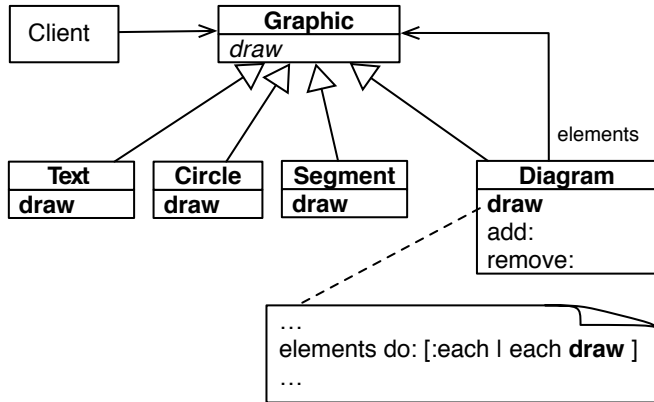
- How do we draw diagram elements?
- How do we draw a diagram?

We do not want to have to check if we are talking to an element or to a diagram composed of elements!



Composite motivation

Elements and diagrams should offer the same API!



Composite: Intent

- Compose objects into tree structures to represent **part-whole** hierarchies
- Let clients treat **individual** objects and **compositions** of objects **uniformly**

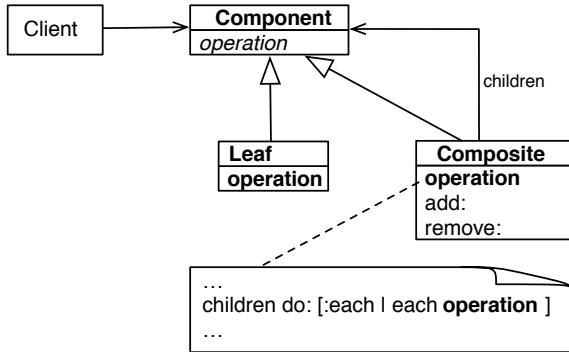
Client's code:

```
aGraphic draw
```

aGraphic **being** a Text, Circle, ... **or even** Diagram (group of Graphics)



Essence of the Composite design pattern



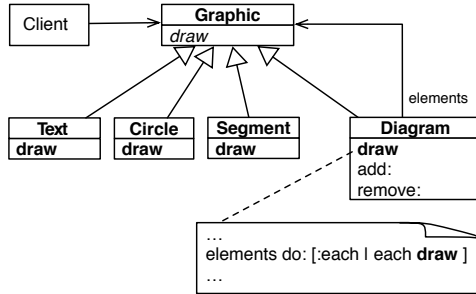
Essence of the Composite design pattern

What is key:

- Leaves offer the **same** API as the composite
- Each leaf do something **different** but with the **same** API (polymorphism)
- The composite element offers the same API and some functionality to manage children
- Leaves and the composite are **substituable**
 - Clients do not have to check

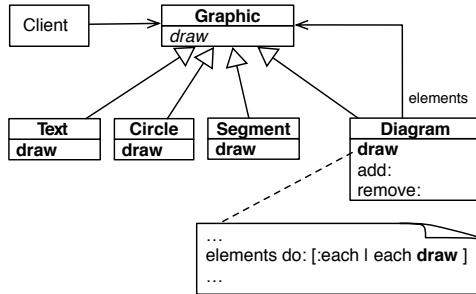


Composite participants: Client



Client manipulates objects in the composition through the **Component** interface (here **Graphic**)

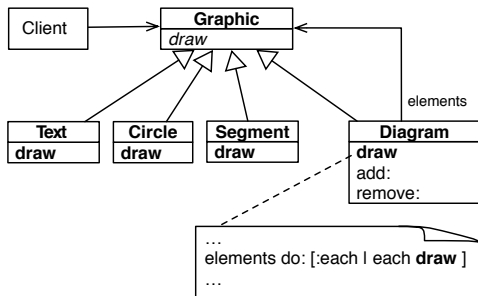
Composite participants: Component



Component (here Graphic)

- declares the interface for objects in the composition
- **may** implement a default behavior for common interface
- **may** declare an interface for accessing and managing its child components
 - see Lecture on “Polymorphic objects”

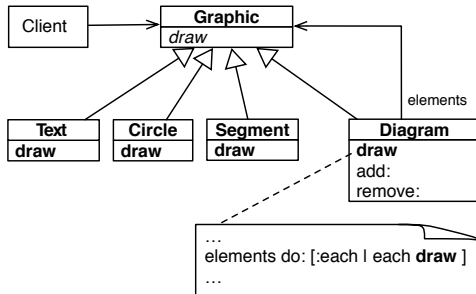
Composite participants: Leaf



Leaf (here Circle, Segment, Text, ...)

- represents leaf objects in the composition
- usually has no children
- defines behavior for primitive objects in the composition using a **polymorphic API**

Composite participants: Composite



Composite (here Diagram)

- defines behavior for components with children via a **polymorphic** API (here `draw`)
- stores child components
- implements child-related operations (`add`, `remove`, ...)

Composite consequences

- Leaves and composite objects expose a common polymorphic API
- Clients do not have to check and treat composite and leaves objects uniformly
- Adding new leaves is simple



Important!

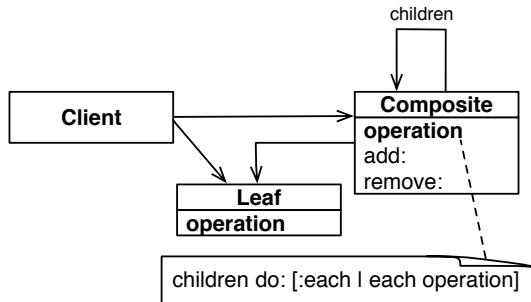
A Design Pattern:

- is a name and an intent
- can have multiple implementations (pros/cons)



Composite in dynamically-typed languages

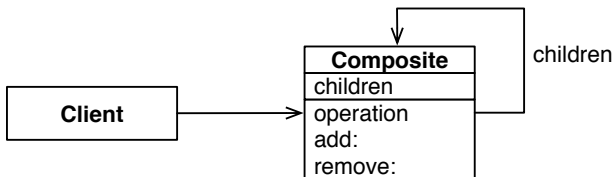
- Polymorphism results from compatible API and not compile-time types checking (see Lecture on Polymorphic objects)
- So, composite and leaves do **not** have to inherit from a common ancestor
 - more difficult to recognize the composite but it works



Composite: extreme implementation

Extreme Composite implementation:

- a single class
- the components (leaves) are composite with no children
- the gain of such an implementation is unclear



Frequently Asked Questions

Can Composite contain any type of child?

- Yes if they implement the common API
- Wrappers or adapters can help with third-party objects

Can we limit the depth of a composite object (number of children)?

- Yes

Can we have different Composites within the same system?

- Yes and each Composite can have a different constraints, behavior, ...



About Composite behavior

Variations on Composite behavior:

- **Simple forward** sends the message to all the children and merges the results without performing any other behavior
- **Selective forward** conditionally forwards the message to some children
- **Extended forward** adds an extra behavior and delegates to leaves
- **Overriding** does not delegate to leaves



Composite and other design patterns

Composite and Visitors

- Visitors walk on structured recursive objects e.g. composites
- see Lectures on Visitor

Composite and Factories

- Factories can create composite elements



Conclusion

- Composite is about composing objects into tree structures to represent part-whole hierarchies
- Composite provides a uniform API to clients for leaves and composite
- Basis for complex treatments expressed as Visitors
 - see Lectures on Visitor



Produced as part of the course on <http://www.fun-mooc.fr>

Advanced Object-Oriented Design and Development with Pharo

A course by

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone



Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>