Composite Pattern

Context:

We have an application that uses a variety of components, which look alike.

We want to define a Component that is itself composed of other components, so it behaves like a single component.

Forces:

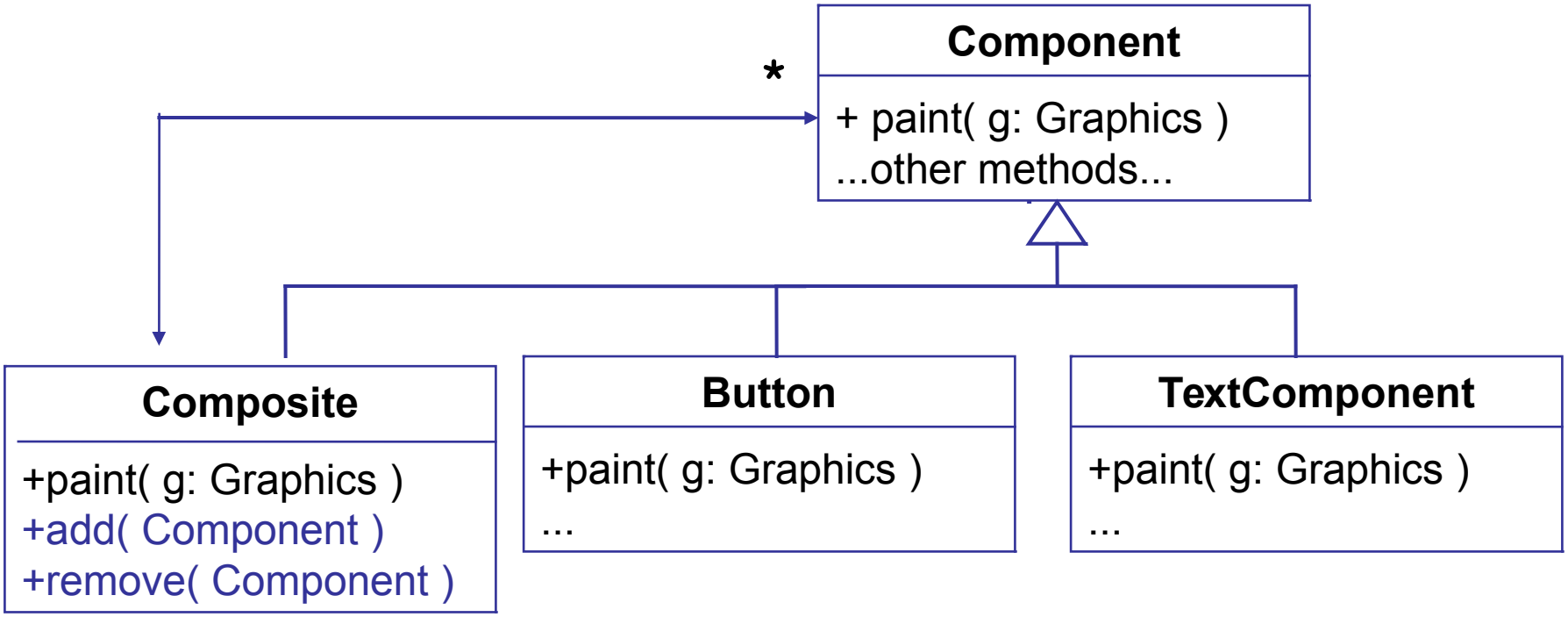
1) we want components to be freely composable.

2) we don't want the application to handle composites as a special case, which would add complexity.

Composite Pattern

Solution:

Define a *composite* that implements the component interface and contains a collection of components. The compposite is responsible for managing components.



Consequences

The application can treat the composite exactly the same as a generic component.

Complexity of managing composite elements is delegated to the composite component.

Example:

In Java GUI (AWT and Swing), a Container is a composite component. A Container is itself a subclass of Component.

JPanel and JWindow are examples of Container.

Applications

Example:

In Java AWT and Swing, a Container is a composite component. A Container is itself a subclass of Component.

Any place that you can use a Component, you can use a Container of many components.

Bundle Item

Context:

A store wants to offer a special price on a "bundle" of Items for sale in the store. The customer gets special price if he buys Items in the bundle (e.g. Beer + Peanuts).

Forces:

The promotions change often. The store doesn't want to modify the software to know about promotions.

Solution:

Define "Bundle" as a LineItem in a sale that contains other Items for purchase.

Bundle Item

UML Diagram

in class

Consequences

Adds complexity to the way items are added to a Sale, and how items are removed from a Sale.