Null Object

Problem

- Given that an object reference may be optionally null, and that the result of a null check is to do nothing or use some default value, how can the absence of an object — the presence of a null reference — be treated transparently?

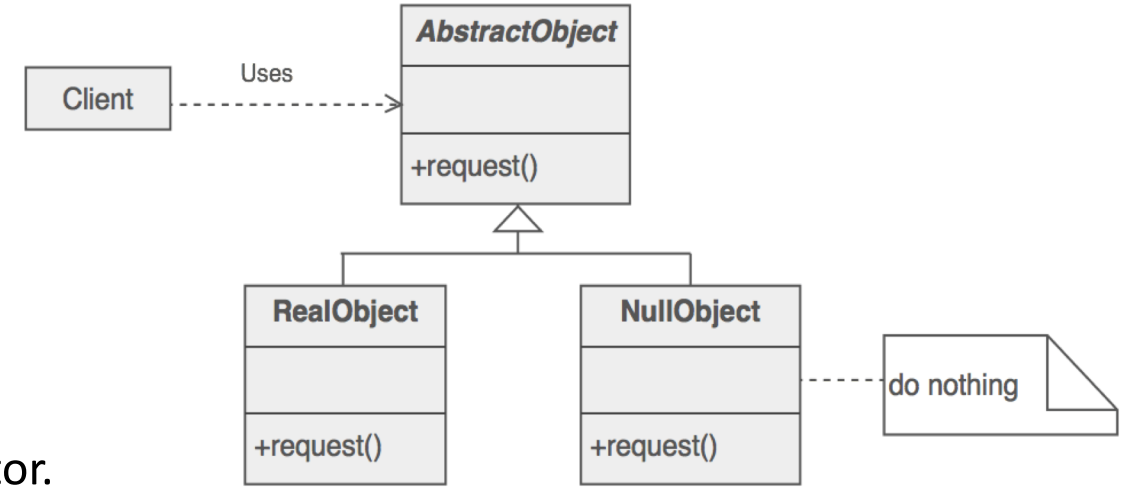
Motivation

Intent

- Provide an object as a surrogate for the lack of an object of a given type. The Null Object provides intelligent “do nothing” behavior, hiding the details from its collaborators.

- Use the Null Object pattern when
  - an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists
  - some collaborator instances should do nothing
  - you want to abstract the handling of null away from the client

Structure and Participants



- Client
  - Requires a collaborator.
- AbstractObject
  - declares the interface for Client's collaborator.
  - implements default behavior for the interface common to all classes, as appropriate.
- RealObject
  - defines a concrete subclass of AbstractObject whose instances provide useful behavior that Client expects.
- NullObject
  - provides an interface identical to AbstractObject's so that a null object can be substituted for a real object.
  - implements its interface to do nothing. What exactly it means to do nothing depends on what sort of behavior Client is expecting.
  - when there is more than one way to do nothing, more than one NullObject class may be required.

Collaborations

- Clients use the AbstractObject class interface to interact with their collaborators. If the receiver is a RealObject, then the request is handled to provide real behavior. If the receiver is a NullObject, the request is handled by doing nothing or at least providing a null result.

Consequences

- Defines class hierarchies consisting of real objects and null objects
- Makes client code simple. Clients can treat real collaborators and null collaborators uniformly
- Encapsulates the do nothing code into the null object. The do nothing code is easy to find.
- Makes the do nothing code in the null object easy to reuse and modify
- Requires creating a new NullObject class for every new AbstractObject class.
- Can be difficult to implement if various clients do not agree on how the null object should do nothing
- Always acts as a do nothing object. The Null Object does not transform into a Real Object.

Composite

Motivation

Intent

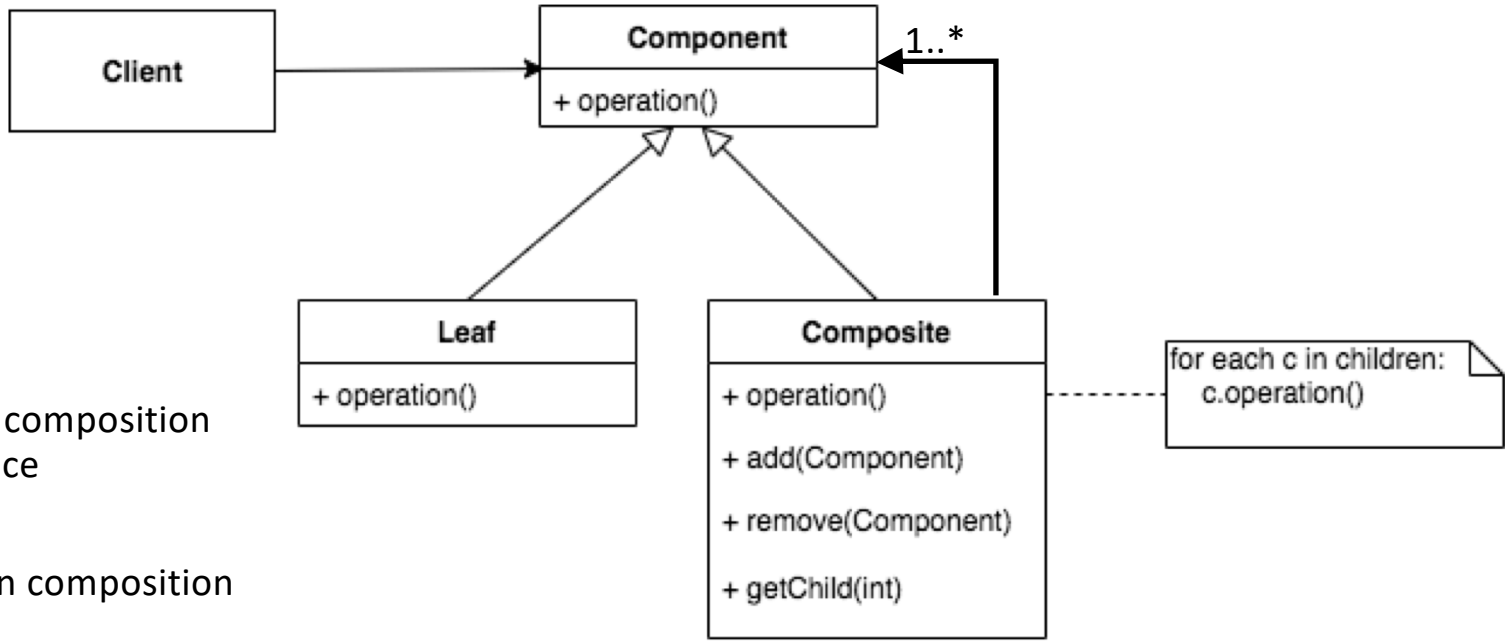
- Compose objects into tree structures to represent part-whole hierarchies

- Treat individual objects and compositions of objects uniformly

Applicability

- **Use the Composite pattern when:**
  - You want to represent hierarchies of objects
  - There should be no distinction between individual and composed elements
  - Objects in the structure can be treated uniformly

Structure and Participants



- **Client**
  - Manipulates the objects in the composition through the component interface
- **Component**
  - Specifies interface for objects in composition
  - Can define default behavior
- **Leaf**
  - Represents the leaf objects in the composition. A leaf has no children
  - Implements the specific behavior for primitive objects in the composition
- **Composite**
  - Stores child components
  - Specifies behavior for components having children
  - Can implement child-related operations (for managing its children)

Consequences

- Defines a class hierarchy consisting of *primitive* and *composite* classes
  - Defines a recursive data structure
  - Primitive objects can be composed into more complex objects
  - And complex objects can also be composed into more complex objects
- Makes client code simple
  - *Composite* and *Leaf* objects treated uniformly
- Supports new types of *Components* without breaking existing code
  - Follows the Open/Closed principle
- It might be difficult to provide a common interface for classes whose functionality differs too much
  - May need to overgeneralize the component interface, making it harder to comprehend
- Can make your design overly general
  - Sometimes we want to restrict the components of a composite

Variants

Design for type safety

- Where define child-related operations?
  - Compromise between transparency and safety
- Child-related operations defined only in Composite
- Clients treat *Leaf* and *Composite* objects differently
- But type-safety is assured

Design for uniformity

- Child-related operations defined in Component
  - Can specify a default behavior for Leaf
- Clients treat Leaf and Component uniformly
- But type-safety is lost