CISC 322

Software Architecture

Lecture 15:

Design Patterns 2

Emad Shihab

Material drawn from [Gamma95, Coplien95]

Slides adapted from Spiros Mancoridis and Ahmed E. Hassan

Façade Pattern Motivation

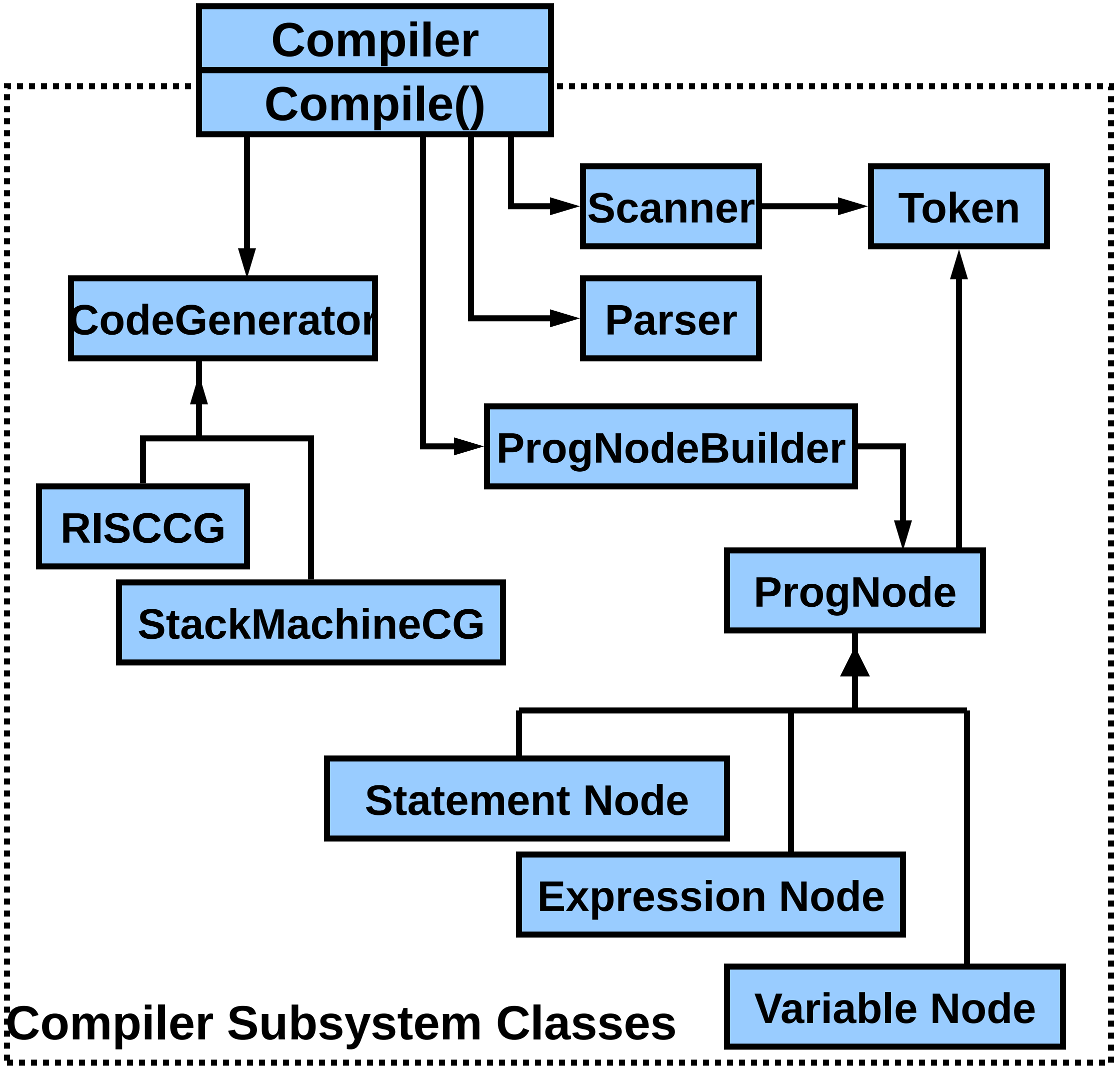
- Structuring a system into subsystems helps reduce complexity
- A common design goal is to minimize the communication and dependencies between subsystems
- Use a facade object to provide a single, simplified interface to the more general facilities of a subsystem

Façade Pattern Intent

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use

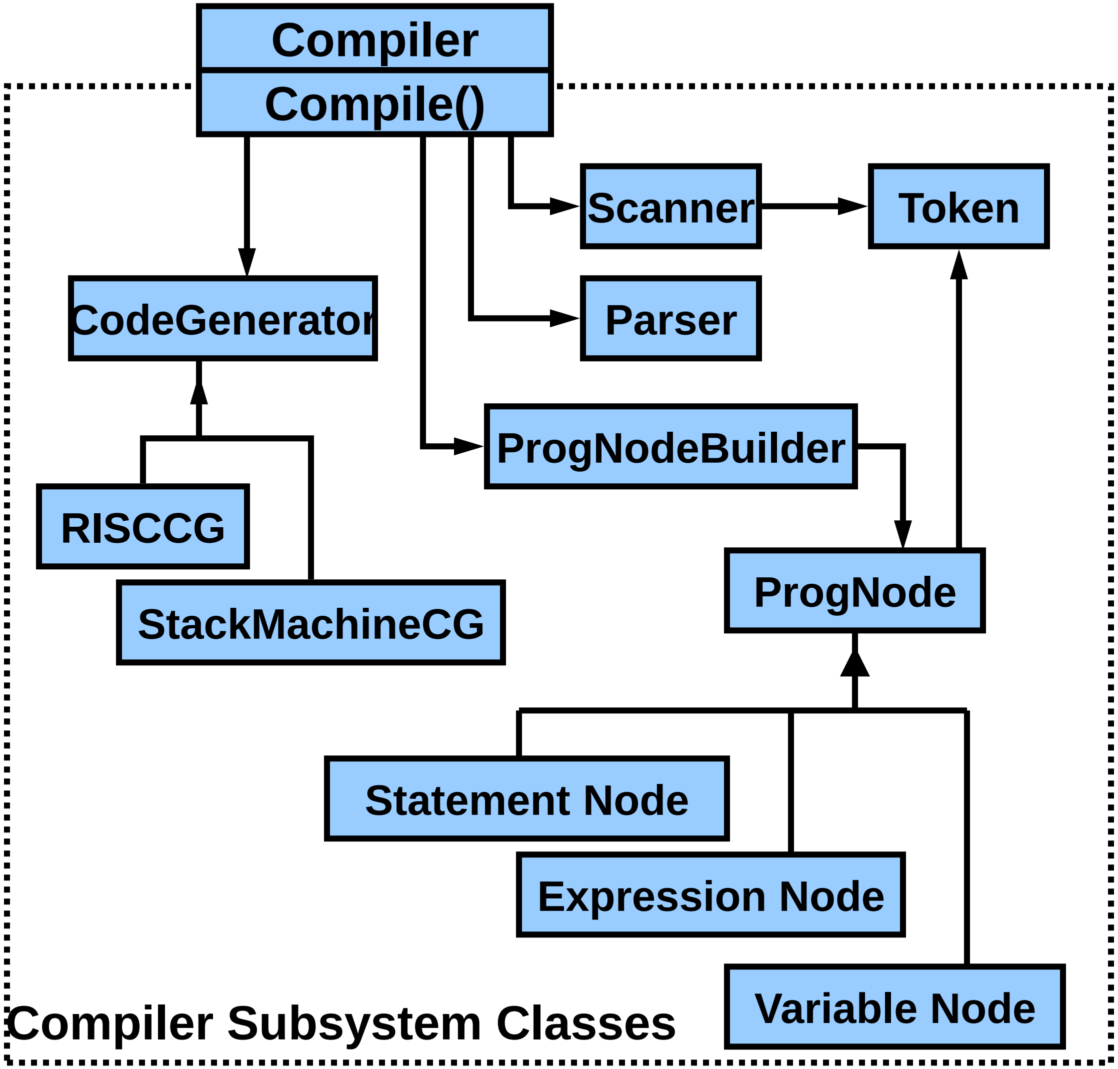
Façade Example – Programming Environment

- Programming environment that provides access to its compiler
- Contains many classes (e.g. scanner, parser)
- Most clients don't care about details like parsing and code generation…just compile my code!
- Low-level interfaces just complicate their task

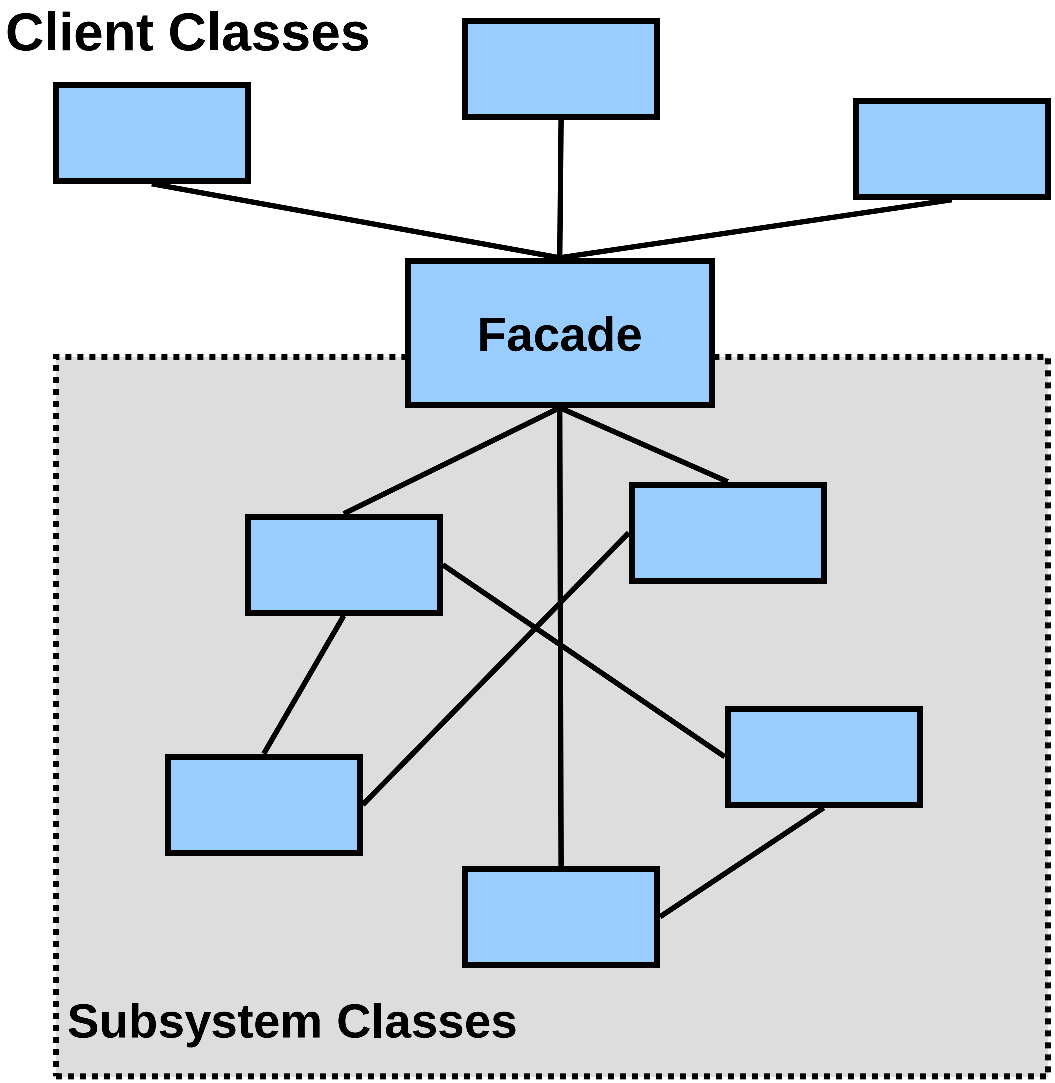


Façade Example – Programming Environment

- Higher-level interface (i.e., Compiler class) shields clients from low level classes
- Compiler class defines a unified interface to the compiler's functionality
- Compiler class acts as a Façade. It offers clients a simple interface to the compiler subsystem



Façade Pattern Structure



Participants of Façade Pattern

- Façade (compiler)
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser,etc..)
  - Implements subsystem functionality
  - Handles work assigned by the façade object

Façade Pattern Applicability

- Use a façade when
  - To provide a simple interface to a complex subsystem
  - To decouple clients and implementation classes
  - To define an entry point to a layered subsystem

Façade Pattern Collaborations

- Clients communicate with the subsystem by sending requests to façade, which then forwards requests to the appropriate subsystems
- Clients that use the façade don’t have access to its subsystem objects directly. However, clients can access subsystem classes if they need to

Composite Pattern Motivation

- Assume you have client code that needs to deal with individual objects and compositions of these objects

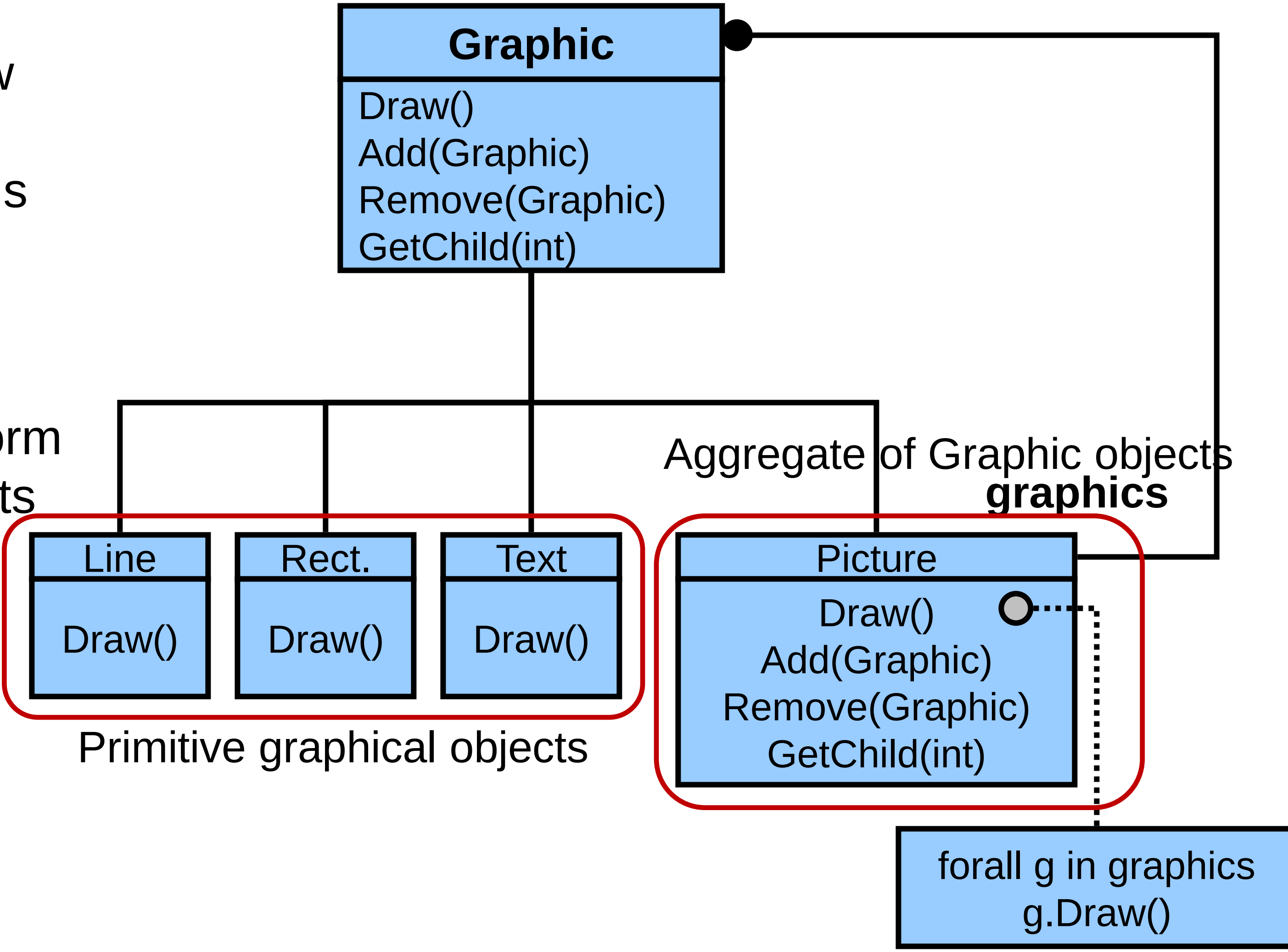
- You would have to treat primitives and container classes differently, making the application more complex than necessary

Composite Pattern Intent

- Lets clients treat individual objects and compositions of objects uniformly

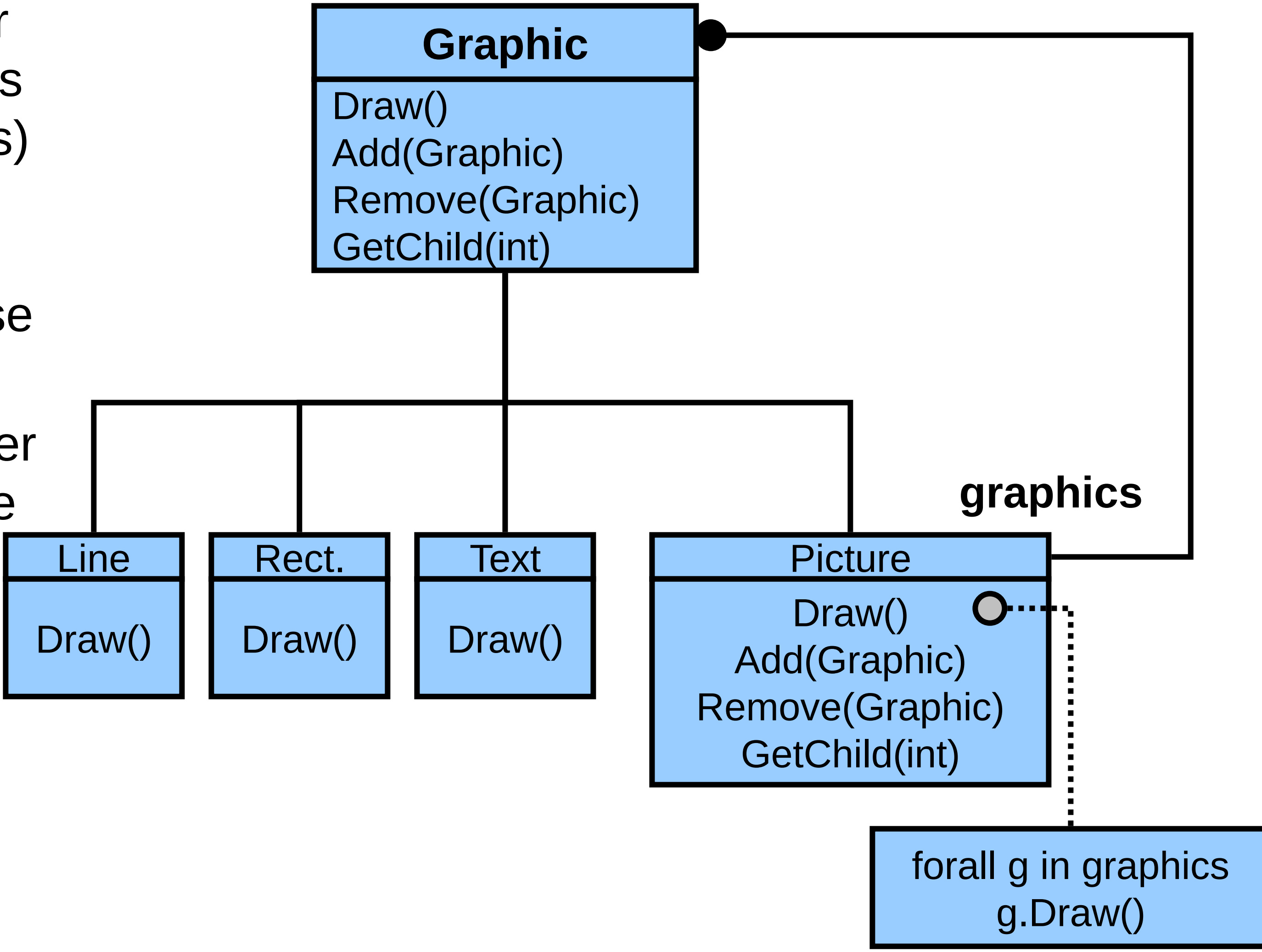
Composite Pattern Example

- Graphic applications allow users to build complex diagrams out of simple components
- Users group components to form larger components



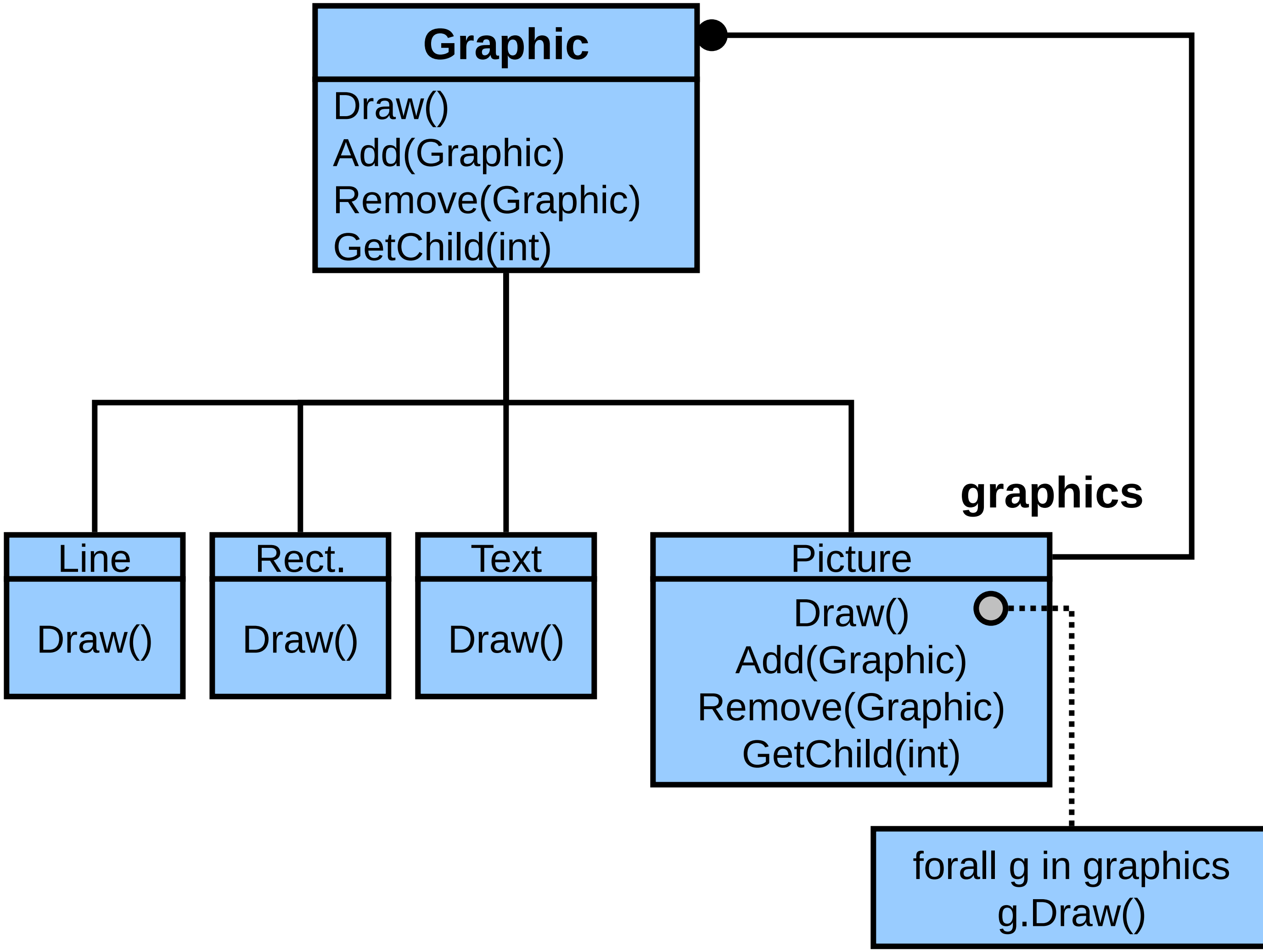
Composite Pattern Example

- A simple implementation defines classes for graphical primitives (e.g. Text and lines) plus other classes that act as containers for these primitives
- The problem is user must treat primitive and container objects differently, making the applications more complex

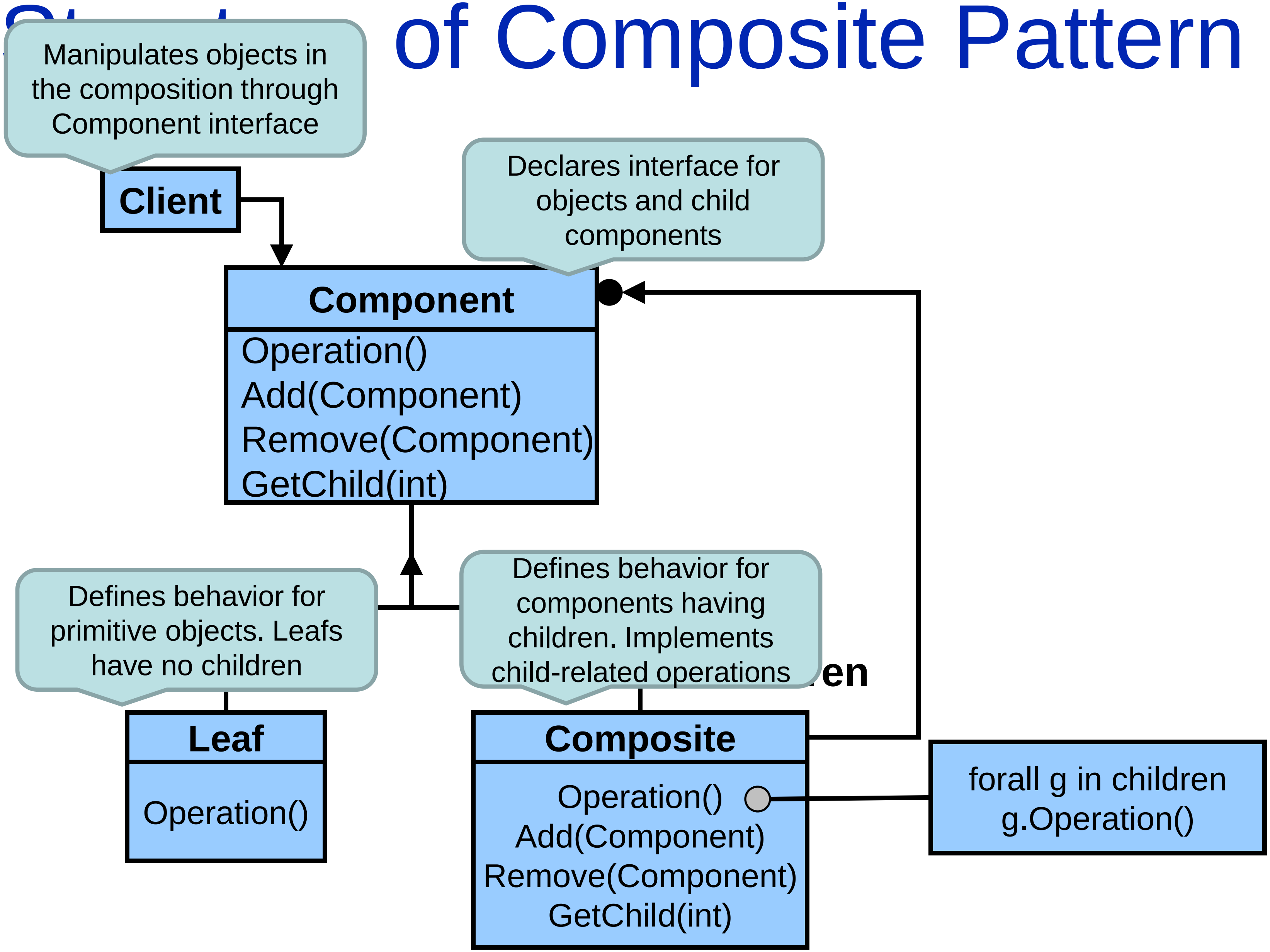


Composite Pattern Example

- Key is an abstract class that represents both primitives and their containers
- Graphic declares operations such as draw that are specific to graphical objects
- Also operations for accessing and managing children



# of Composite Pattern



Manipulates objects in the composition through Component interface

Client

Declares interface for objects and child components

Component

Operation()
Add(Component)
Remove(Component)
GetChild(int)

Defines behavior for primitive objects. Leafs have no children

Defines behavior for components having children. Implements child-related operations

en

Leaf

Operation()

Composite

Operation()
Add(Component)
Remove(Component)
GetChild(int)

forall g in children
g.Operation()

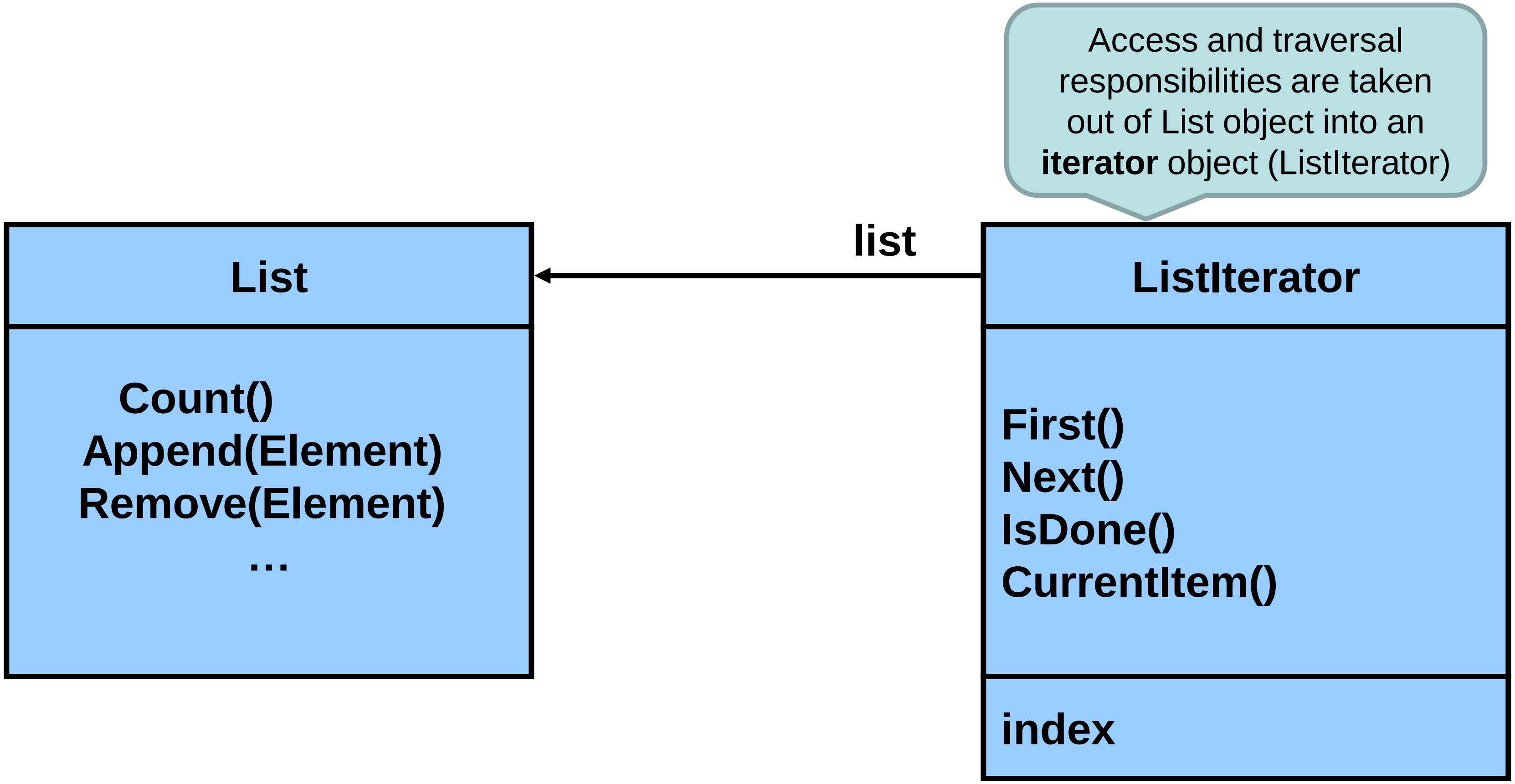
Iterator Pattern Motivation

- Aggregate objects (e.g. list) should give you a way to access its elements without exposing its internal structure
- You might want to traverse an aggregate object in different ways
- Sometimes cannot decide on all ways to traverse the aggregate object apriori
- Should not bloat the interface of aggregate objects with different traversals

Iterator Pattern Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

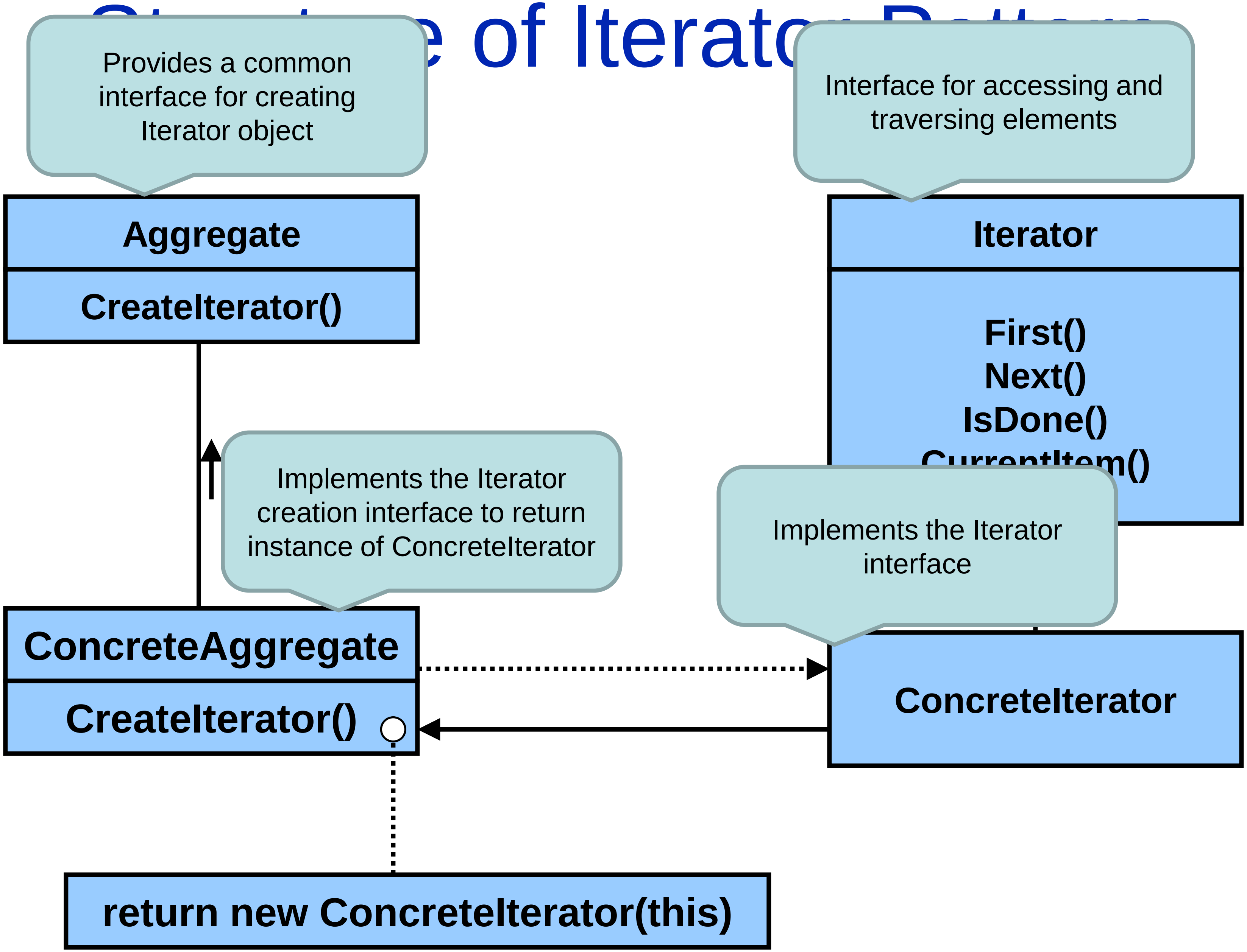
Iterator Pattern Example



Access and traversal responsibilities are taken out of List object into an **iterator** object (ListIterator)

Can define different traversal policies without enumerating them in the List interface

Structure of Iterator Pattern



Provides a common interface for creating Iterator object

Aggregate

CreateIterator()

Interface for accessing and traversing elements

Iterator

First()
Next()
IsDone()
CurrentItem()

Implements the Iterator creation interface to return instance of ConcreteIterator

Implements the Iterator interface

ConcreteAggregate

CreateIterator()

ConcreteIterator

return new ConcreteIterator(this)