Structural Patterns

Jeffrey Liu

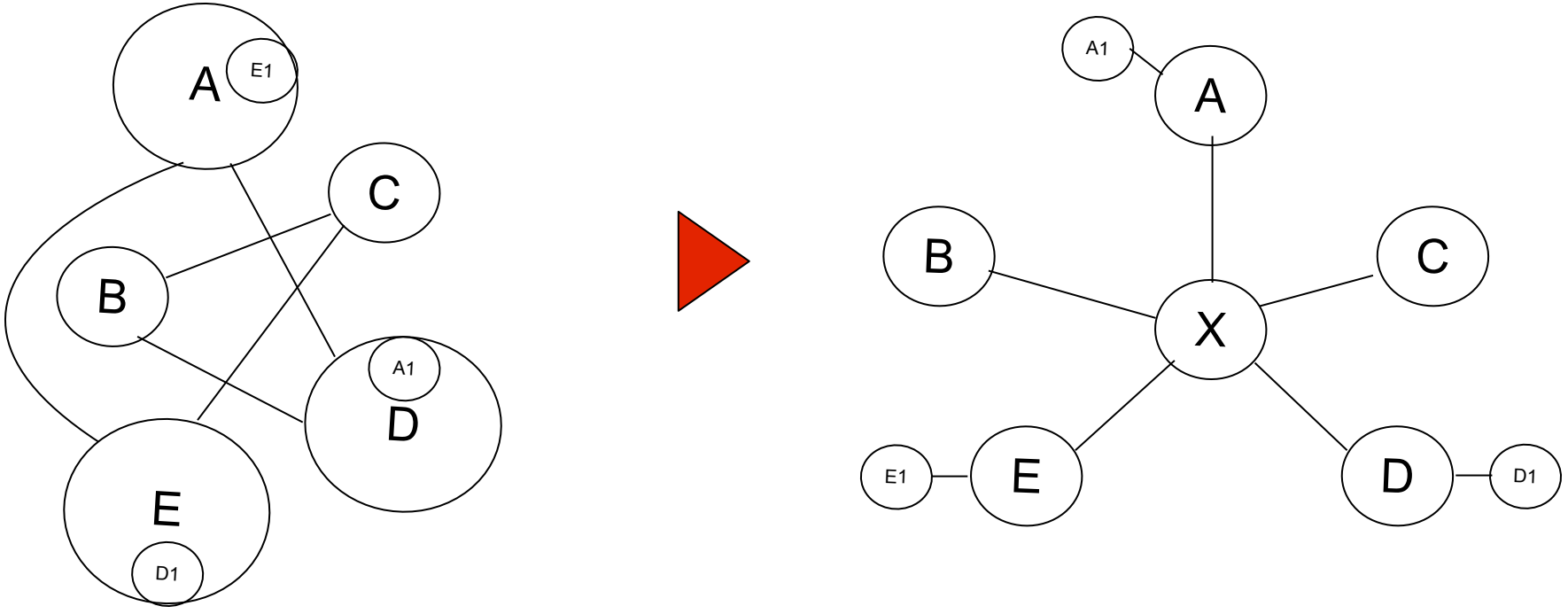
IBM
China Development Lab
Greater China Group

Why structural patterns

- A better way for different entities to work together
- Focus on higher level interface composition and integration.
- Particularly useful for making independently developed libraries to work together

Core Spirits

High Cohesion, Low Coupling



Outline of structural patterns

Adapter

Proxy

Composite

Decorator

Bridge

flyweight

facade

Proxy Pattern

Provide a surrogate or placeholder for another object to control access to it.

Challenge

- Authentication process is quite slow. Is there any way to improve its performance ?
- Can we make the enhancement transparent to existing clients ?

Problem

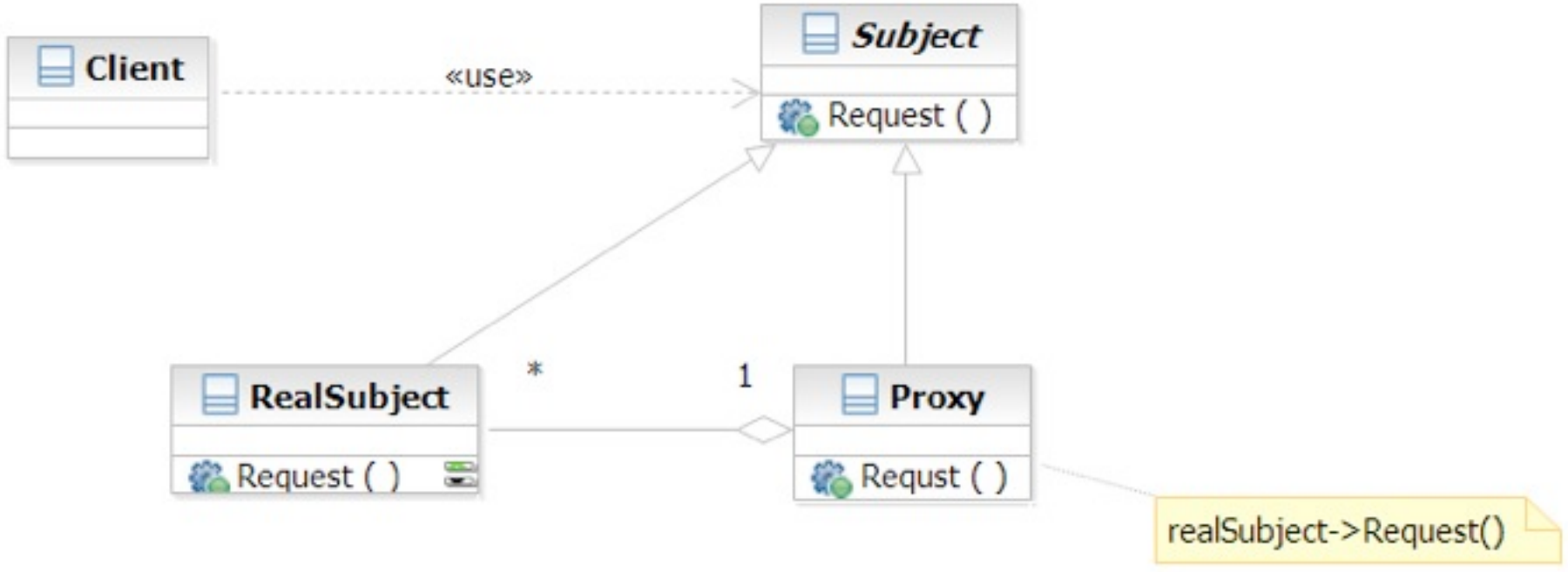
- You want to add a middle-layer between clients and your system.
  - Access control
  - Performance enhancement
- The implementation must be transparent to existing user

Think …

- How to implement the access control?
- Can you do something before client program accesses your resource

- Target:
  - A proxy that can be act as a gate-keeper of existing resource

Structure/Participants

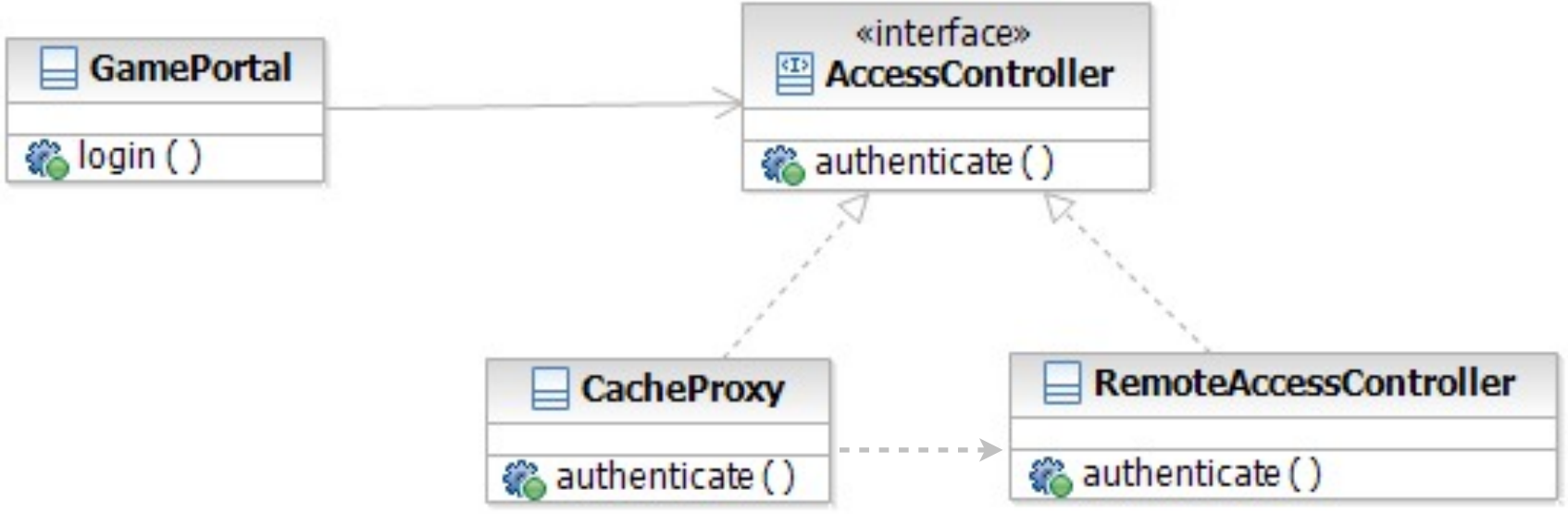


Object Relationship

Applicability

- Uses of Proxy pattern
  - Remote proxy
  - Virtual image proxy
  - Protection proxy

Sample Structure



Consequence

- Indirect access of resources
  - You can always monitor/filter the access request
- Resource control optimization

Related Patterns

- Decorator
  - Proxy focus on resource control instead of adding feature**s** to existing component dynamically

Decorator Pattern

Attach additional responsibilities to an object dynamically

Challenge

- The basic access control system has been implemented, but we need to come up with a general approach so that we can add new features dynamically…

Problem

- You need to **attach/detach** features **dynamically**
- You can't implement various combinations of feature by using subclasses

Think…

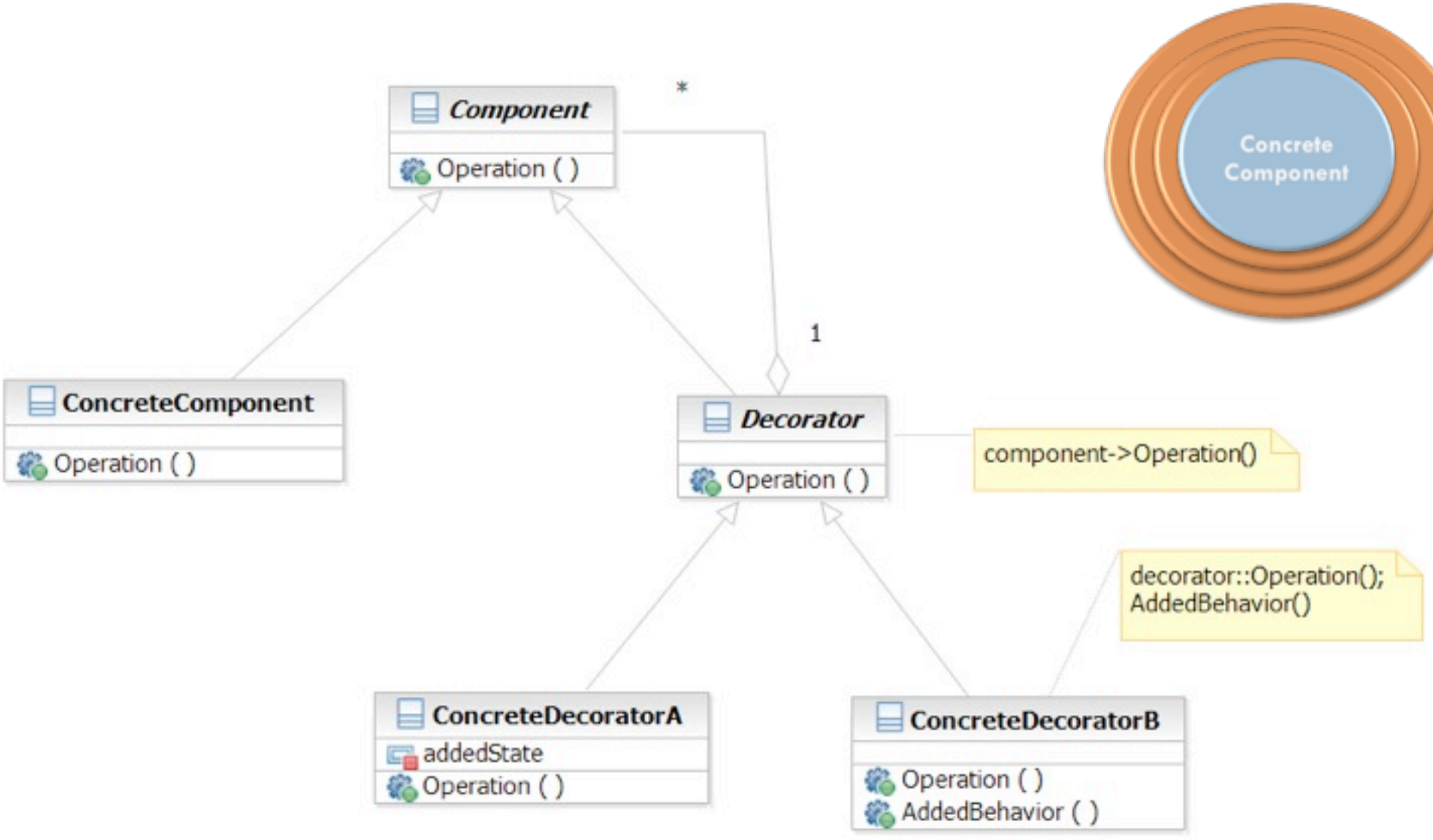
- How to add/remove new features to an "object" dynamically?
- Instead of subclassing, are there any other alternatives?

- Target:
  - Dynamic feature composition.
  - Chain of responsibility.

The Open-Closed Principle

Classes should be open for extension, but closed for modification

Structure/Participants



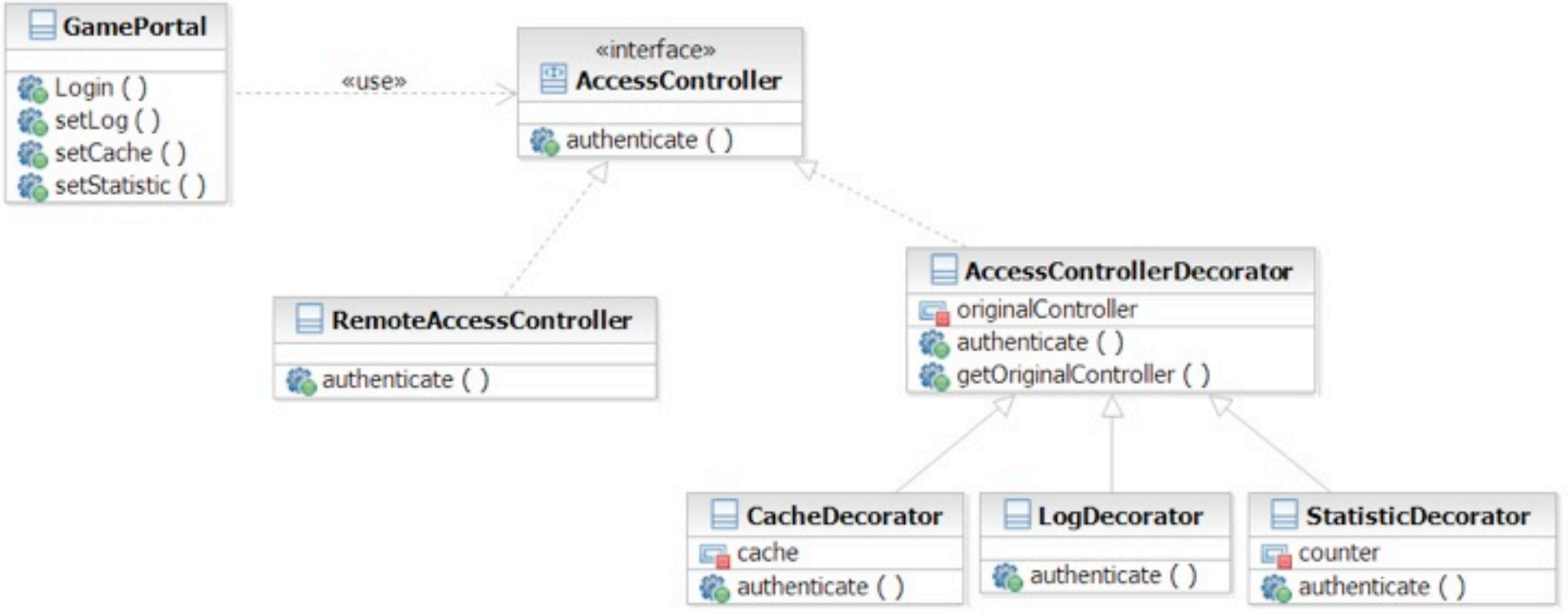
Applicability

- Use Decorator pattern
  - When you want to associate a new feature to an existing object "dynamically" and "transparently"
  - When subclassing is impractical

Implementation

- Minimize the operation exposed by "component"
- Change skin (decorator) V.S change guts (strategy)
  - Transparency
  - Controllability

Sample Structure



Consequence

- More flexible than subclassing
- Avoid feature-overloaded parent class
- Minimize the impact of adding new nodes

Related Patterns

- Composite
  - Structurally similar, but decorator allow adding new feature (responsibility)
- Strategy
  - Change skin V.S change guts
  -

Composite Pattern

Compose objects into tree structures to represent part-whole hierarchies

Problem

- Implement a nested structure for various objects (e.g. team/subteam relationship)
- The interface needs to be unified so that you don't need to worry about which object you are currently dealing with

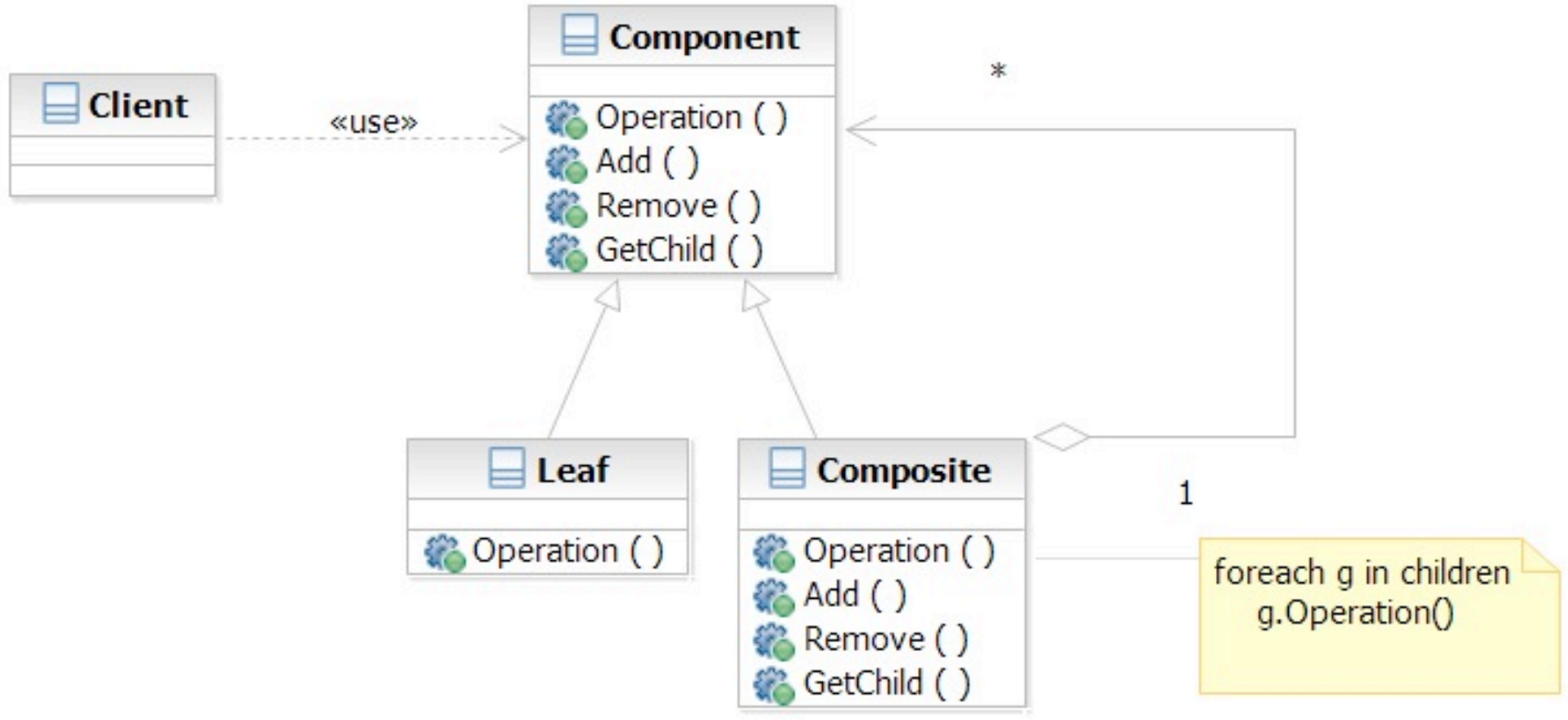
First Attempt

- Implement classes to represent root node/intermediate note/child node separately
- Each kind of node has different interface to reflect its role and responsibility

Think…

- How to represent a tree-like/recursive structure in your code?
- Target:
  - Leverage the beauty of recursive
  - Apply your changes (commands) to the system as a whole

Structure/Participants



Applicability

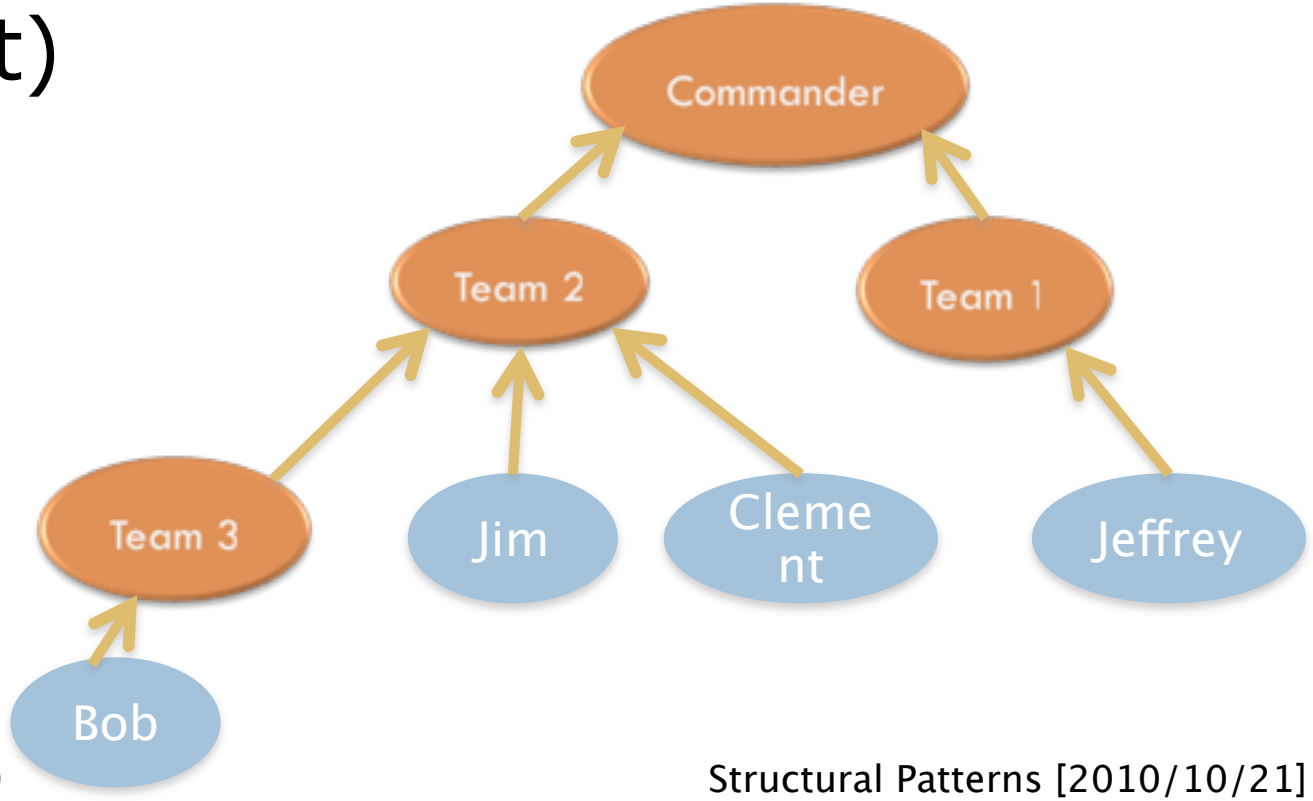
- Use Composite pattern
  - When you need to represent a nested, whole-part relation
  - You want to provide a uniform interface for each node in the system

Implementation

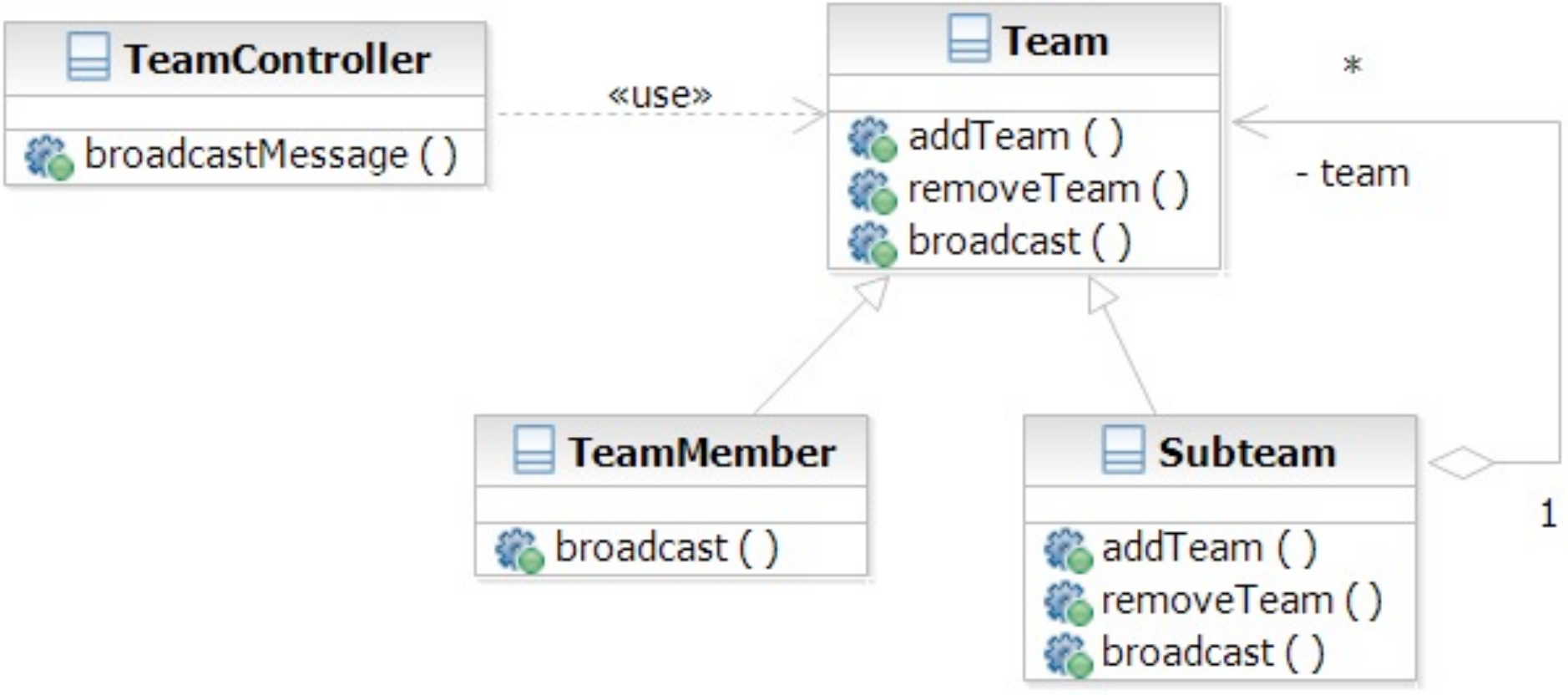
- Reference to parent
- Focus on node manipulation methods
  - Transparency v.s strong type checking
- Relative order between nodes
  - Leverage Iterator/visitor pattern

Sample Scenario

- You want to build up a structure that can represent team/subteam/member relationship
- You want an action to apply to all members (e.g. broadcast)



Sample Structure



Consequence

- A composited structure that has no clear line between composite nodes and leaf nodes
- Reduce client's knowledge about internal structure
- Minimize the impact of adding new nodes

Related Patterns

- Decorator
  - Often used with composite pattern. It implements the same interface of composite so both patterns can be seamlessly integrated
- Iterator
  - Support various of ways to traverse the nested structure
- Visitor
  - Move a specific operation to a visitor instead of complicating the general composite interface

Facade Pattern

Provide a unified high-level interface to a set of interfaces in a subsystem

Challenge

- There are a lot of fine-grained components in our system. Does that mean our client needs to deal with these details?

- Also, someone already proposed an enhancement request for one particular component, which means the component is subject to change. How to make this change transparent to client in the future ?

Problem

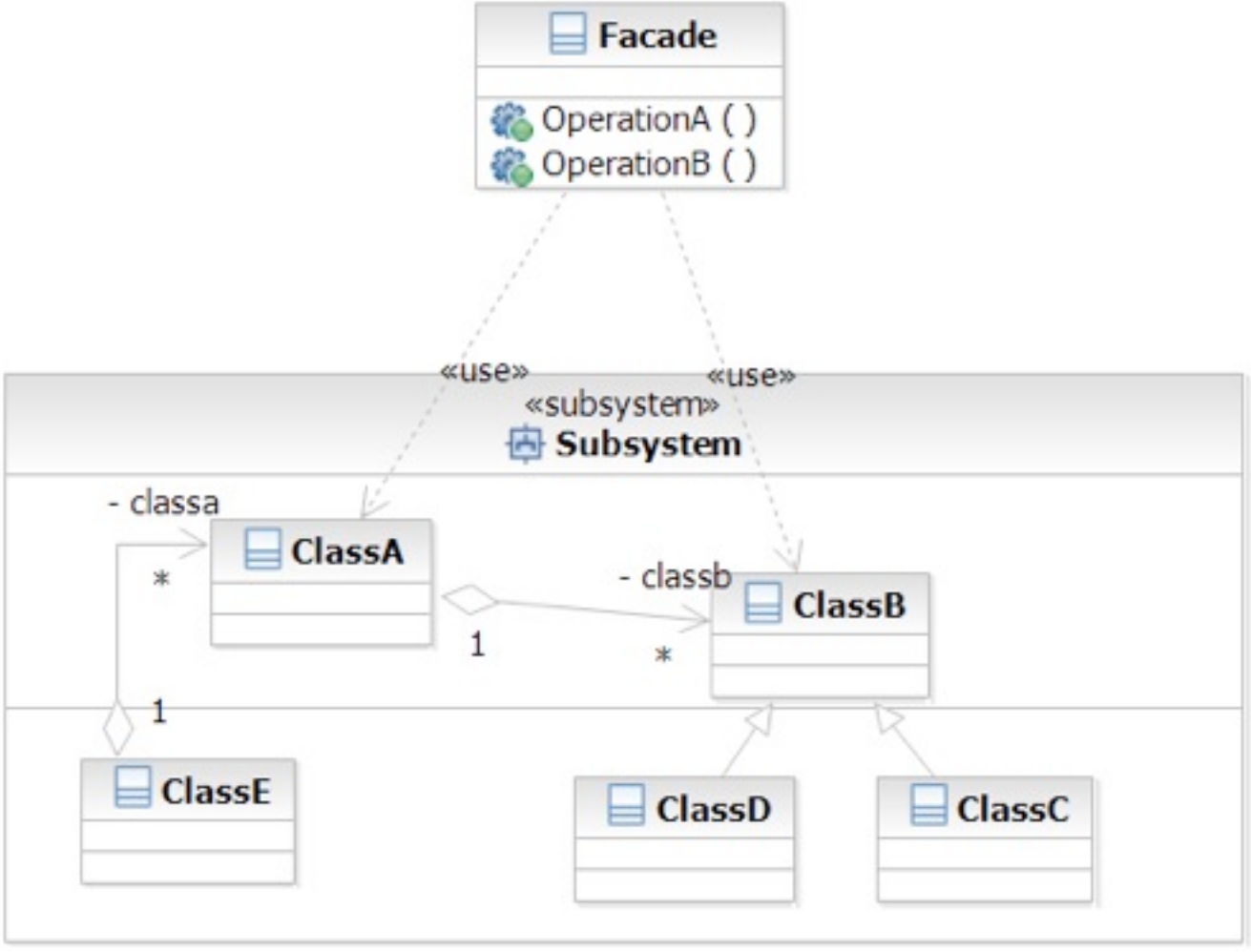
- Each sub-system has its own unique class hierarchy, programming conventions, and usage caveats
- You don't want to have strong binding with a particular class which is subject to be changed

Think …

- How to encapsulate internal details and provide a high-level interface to other **sub–systems**?
- How do you set up the interface contract appropriately ?

- Target:
  - Implement a class whose exposed methods can represent the essential functions of the whole system

Structure/Participants



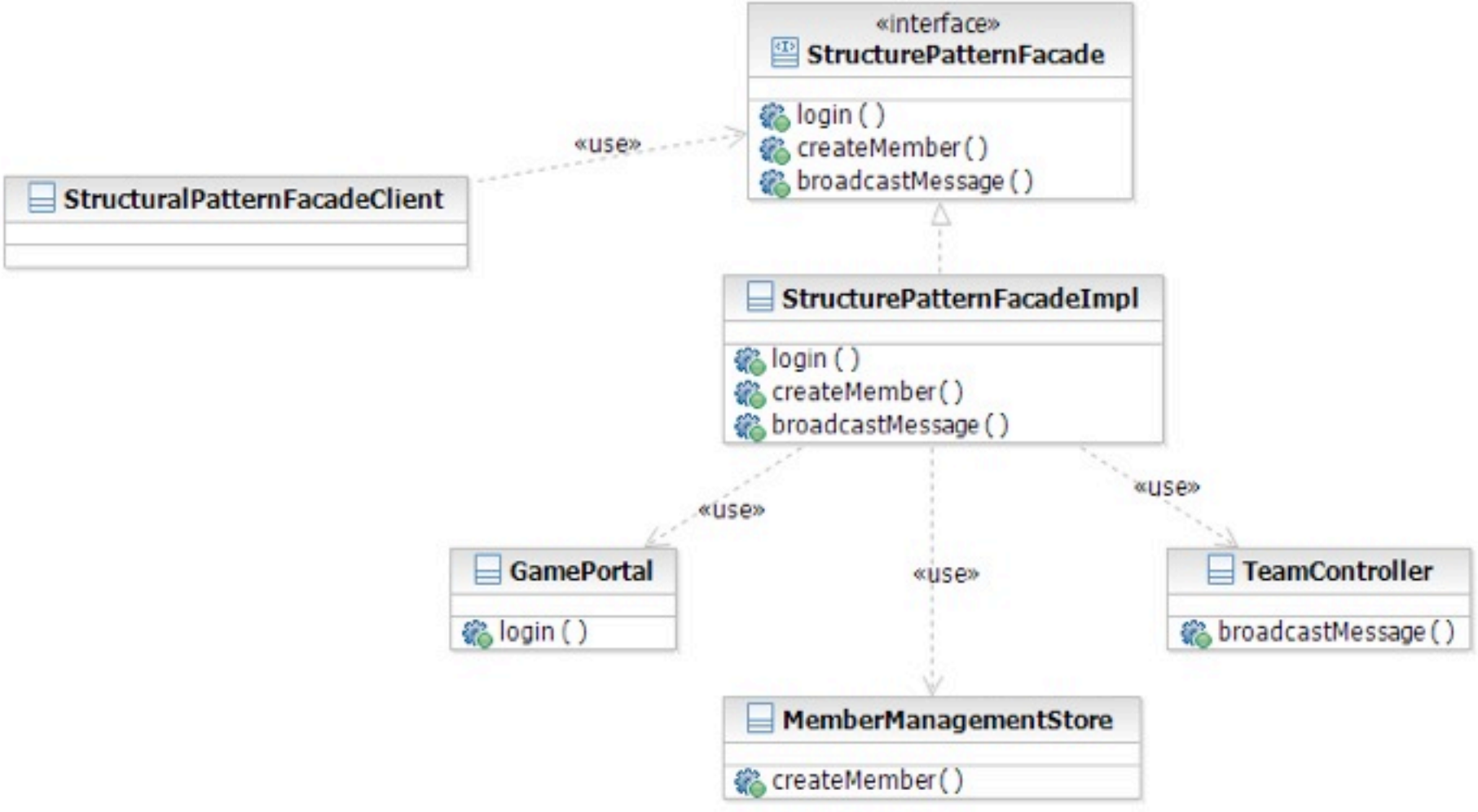
Applicability

- Use Facade pattern
  - When the interface of the class in the sub-system are too complicated to follow
  - When using top-down approach
  - Reduce class dependency

Sample Scenario

- You want to expose various functions of you subsystem
  - Membership management
  - Access Control
  - Team-based operations

Sample Structure



Consequence

- Make sub-system easy to use
- Reduce code dependency among sub-systems
- Design by contract, then stick with the contract

Related Patterns

- Singleton
  - You only need one facade instance most of the time
- Mediator
  - Façade and mediator both abstract the functionality of existing classes
  - Mediator focus on how to abstract the way arbitrary classes communicate with each other
- Proxy
  - The gateway between internal and external system

Structural patterns review

- Use Proxy pattern to serve as a middle layer between two components
- Use decorator pattern when you want to attach/detach features with existing component dynamically

Structural patterns review

- Use Composite pattern to represent nested structure in a flexible way
- Use Façade pattern to provide a higher level of abstraction of your subsystem