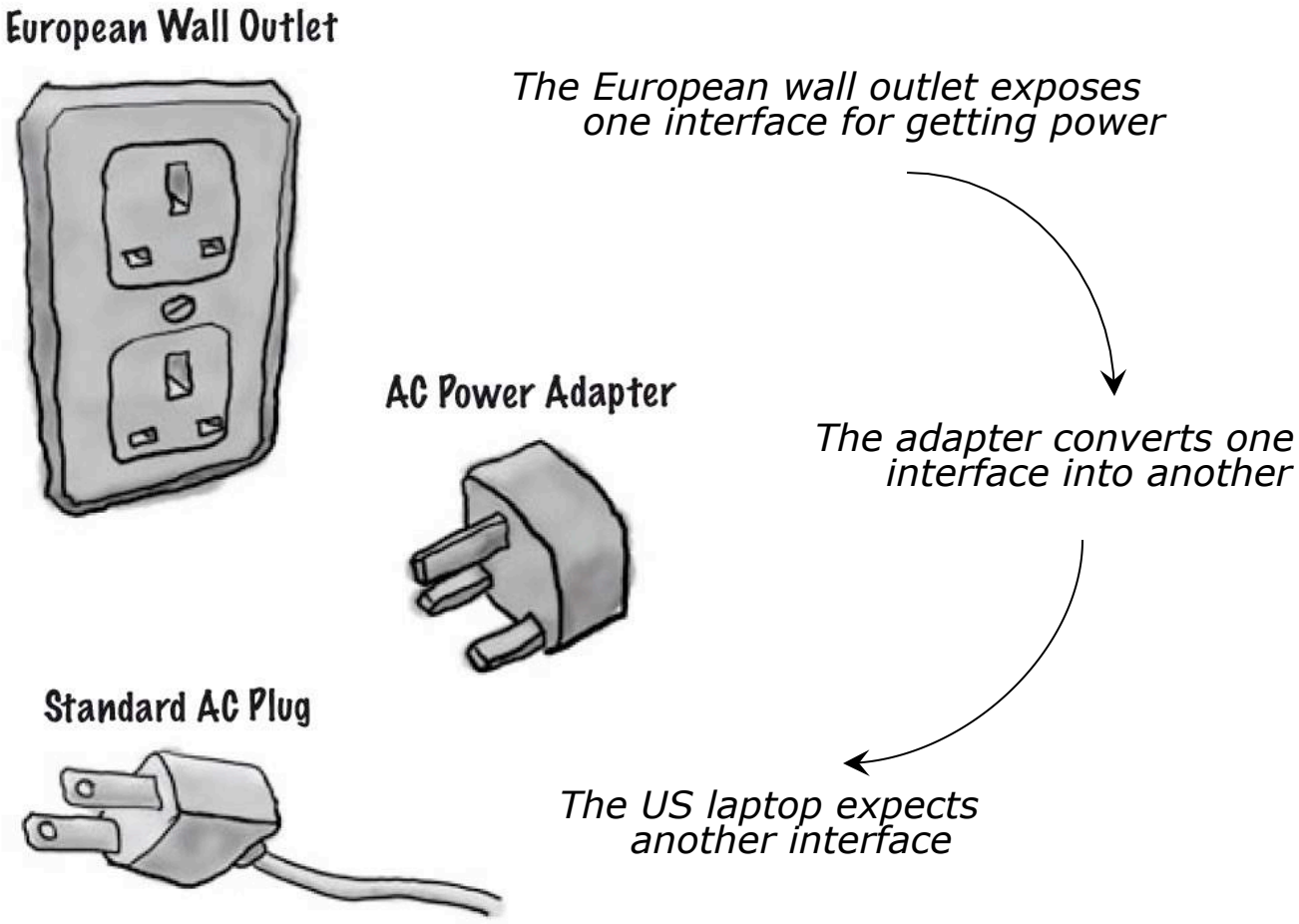
OO Design patterns

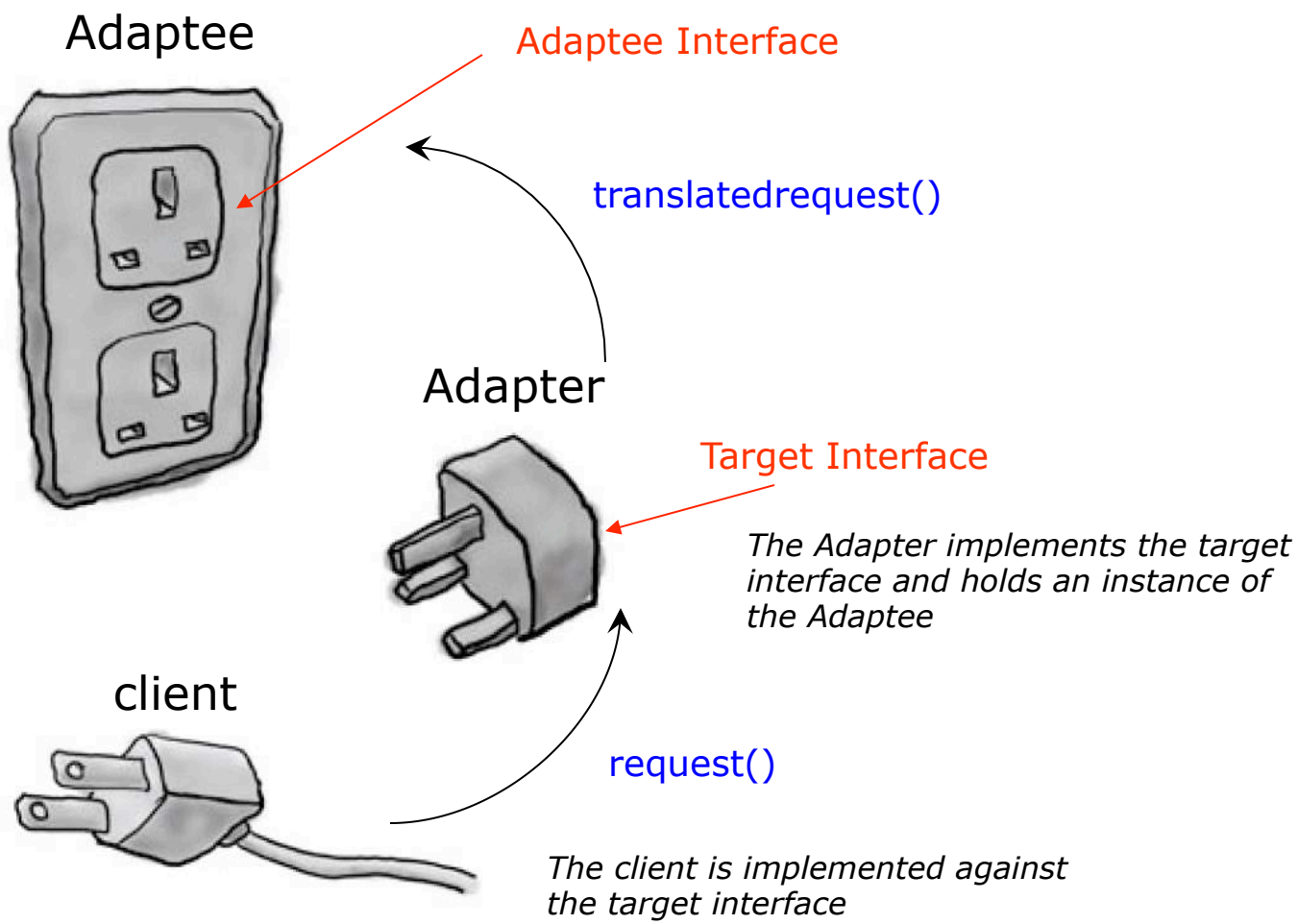
Adapter pattern

Acknowledgement: Freeman & Freeman

Example Scenario



Adapter Pattern Explained



Discussion Questions

- How much "adapting" does an adapter need to do? It seems like if I need to implement a large target interface, I could have a LOT of work on my hands
- Does an adapter always wrap one and only one class?
- What if I have old and new parts of my system, the old parts expect the old vendor interface, but we've already written the new parts to use the new vendor interface? It is going to get confusing using and adapter here and the unwrapped interface there. Wouldn't I be better of just writing my older code and forgetting the adapter?

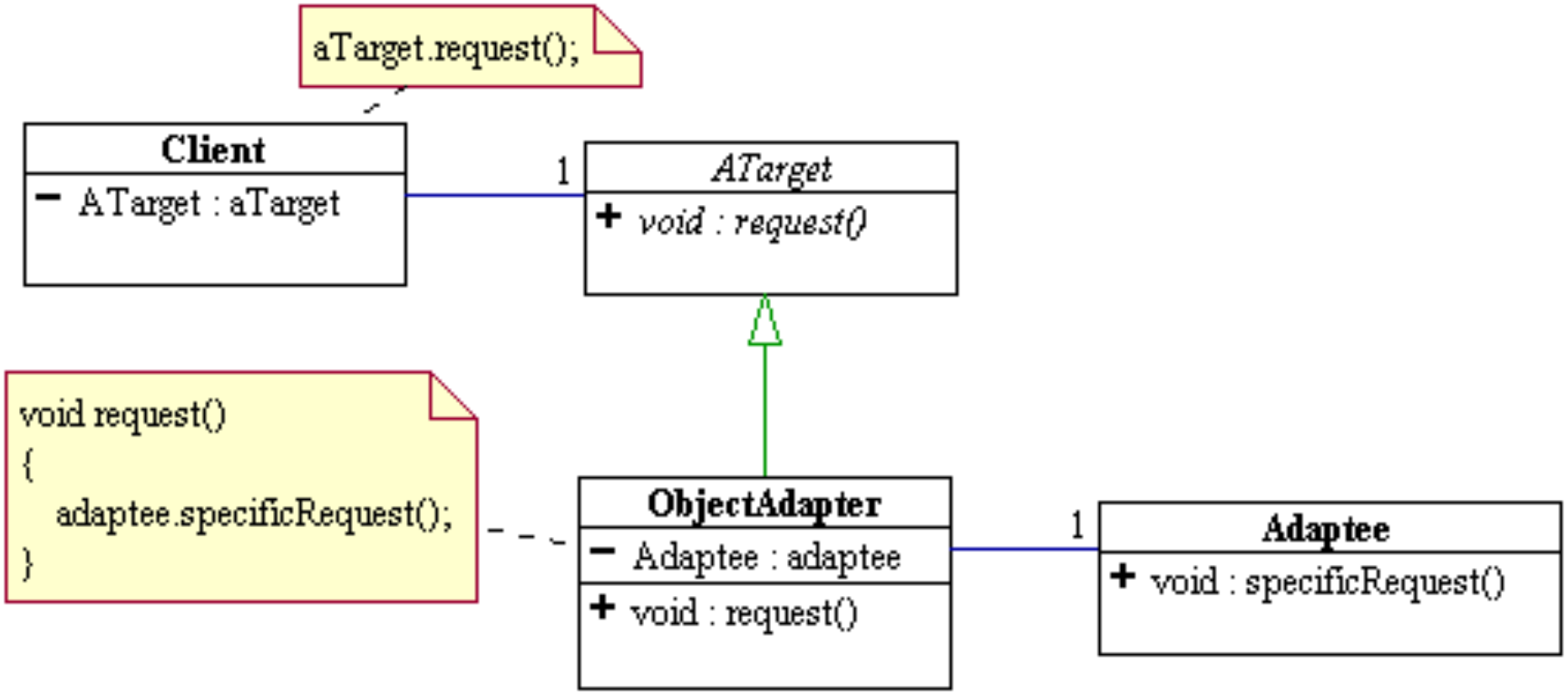
Adapter Pattern

- An adapter pattern converts the interface of a class into an interface that a client expects
- Adapters allow incompatible classes to work together
- Adapters can extend the functionality of the adapted class
- Commonly called “glue” or “wrapper”

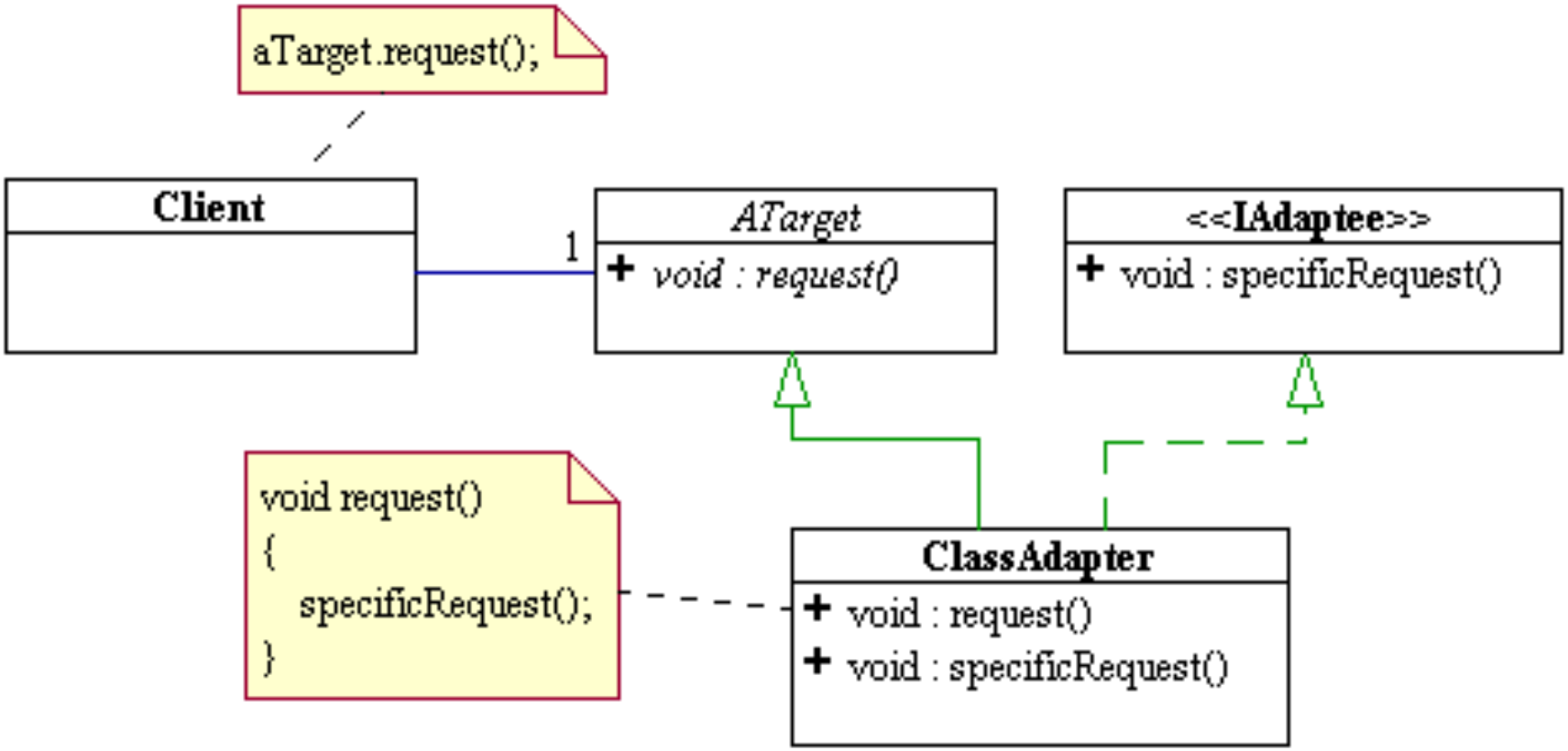
When to Use

- Need to adapt the interface of an existing class to satisfy client interface requirements
  - Adapting Legacy Software
  - Adapting 3rd Party Software

Object Adapter Pattern



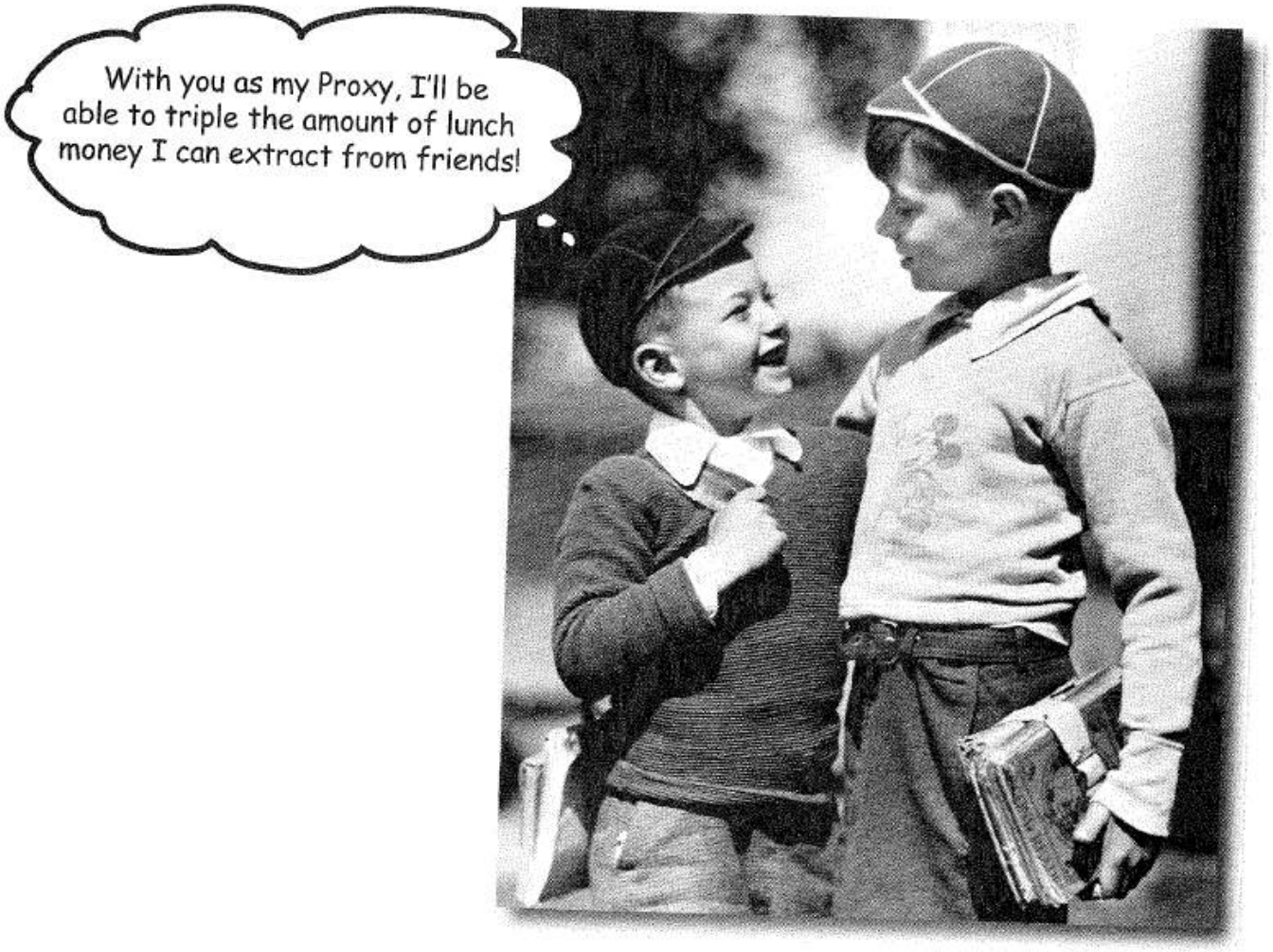
Class Adapter Pattern



Proxy pattern

Acknowledgement: Freeman & Freeman

I like proxies…



The Problems

- Expensive & inexpensive pieces of state
  - Example: Large image
  - Inexpensive: size & location of drawing
  - Expensive: load & display
- Remote objects (e.g., another system)
  - Want to access it as if it were local
  - Want to hide all the required communications
  - Example: Java RMI
- Object with varying access rights
  - Some clients can access anything
  - Other clients have subset of functionality available

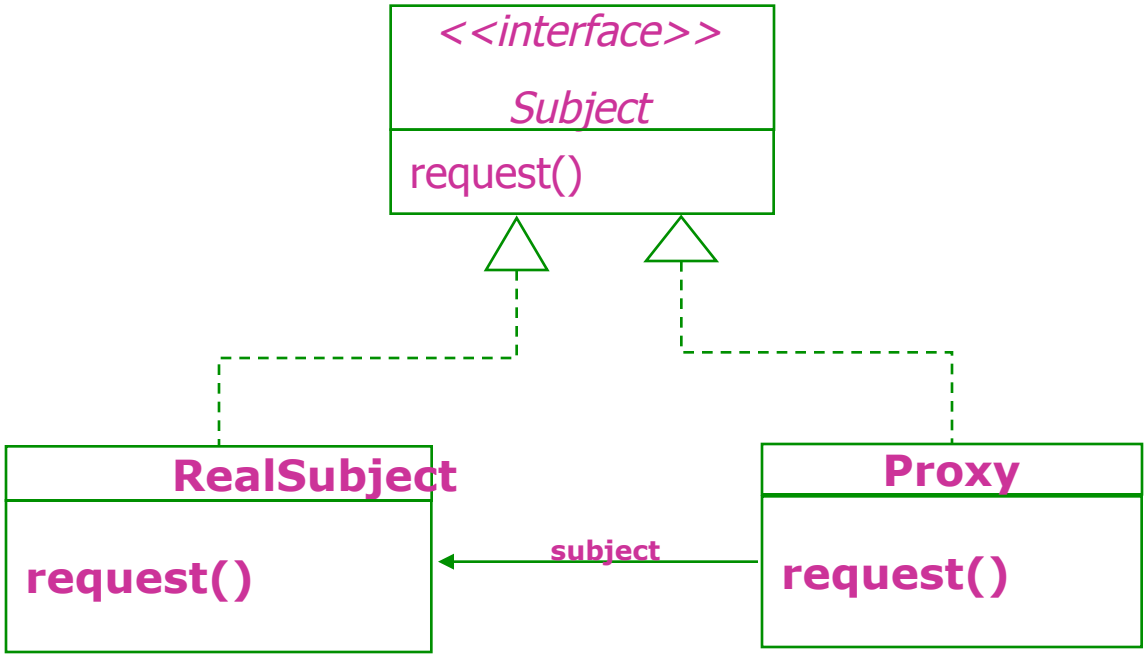
The Design Goal

- In all these cases desire access to object as if it is directly available
- For efficiency, simplicity, or security, put a *proxy* object in front of the real object
- We have a stand-in for the real object to control how the real object behaves

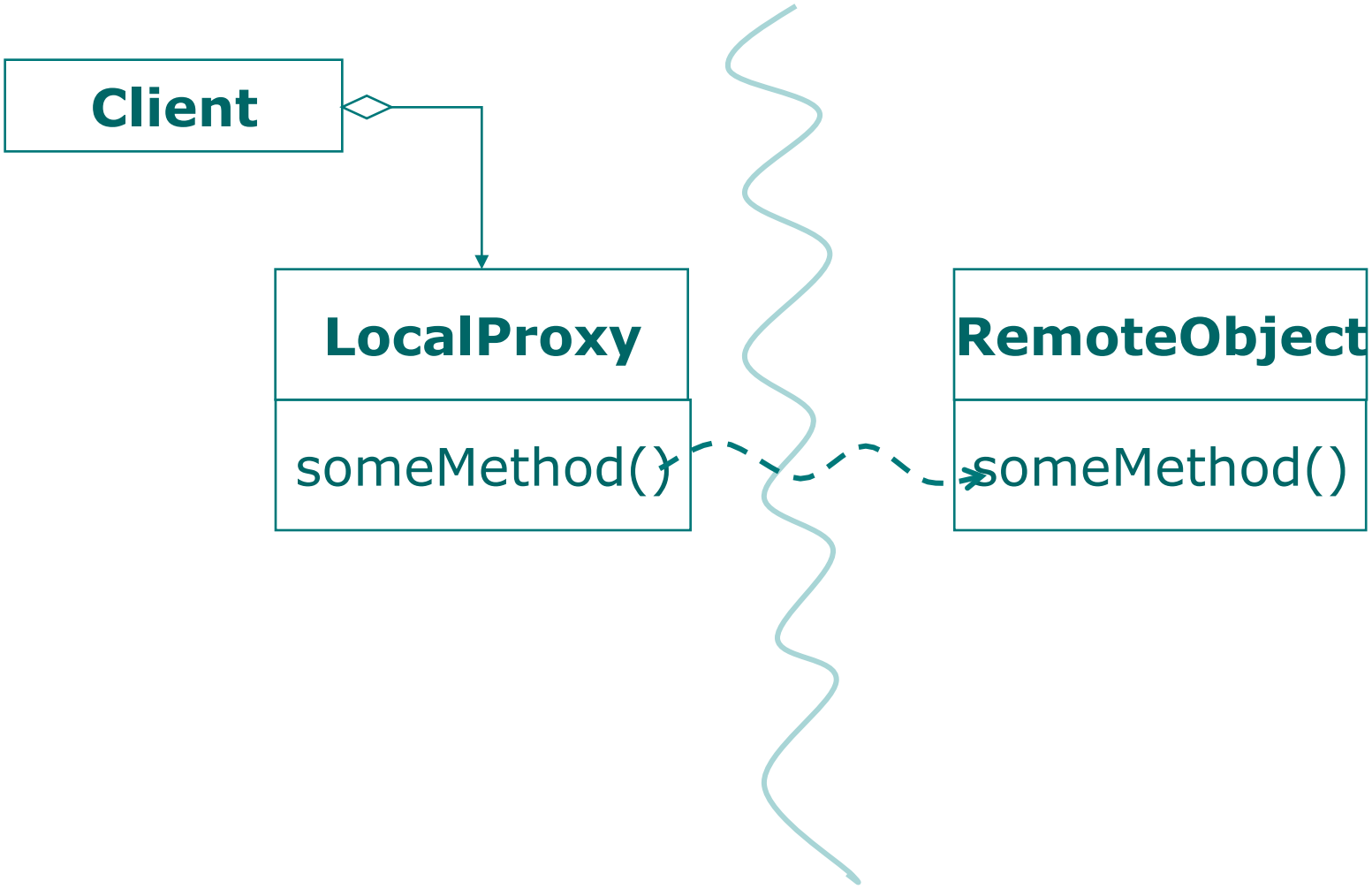
Proxy pattern Defined

- Proxy patterns provides a surrogate or placeholder for another object to control access to it
  - Remote proxy controls access to a remote object
  - Virtual proxy controls access to a resource that is expensive to create
  - Protection proxy controls access to a resource based on access rights

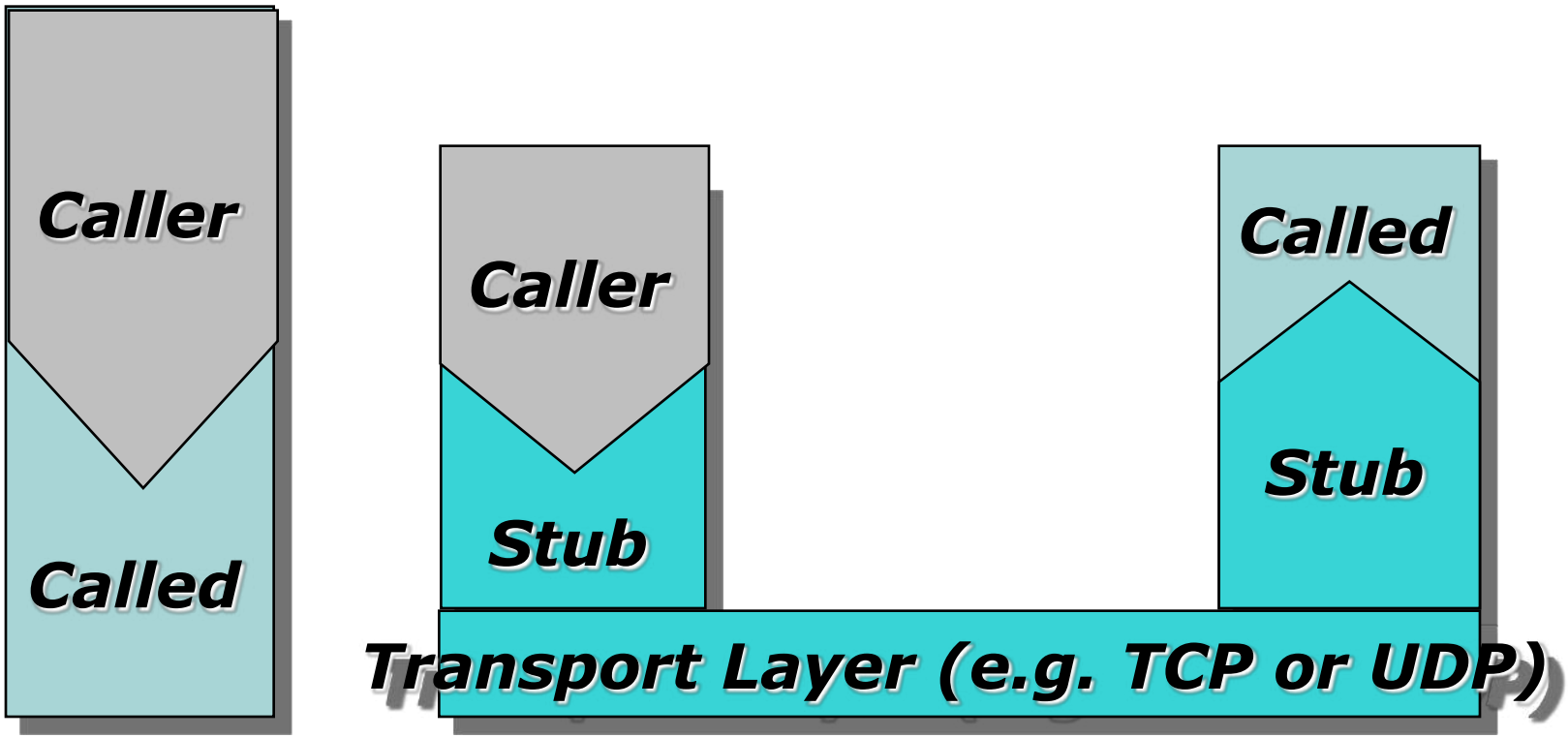
Proxy pattern structure



Example: Accessing Remote Object



Java RMI, the big picture



Categories of Proxies

- **Remote proxy** - as above
  - Local representative for something in a different address space
  - Java RMI tools help set these up automatically
  - Object brokers handle remote objects (CORBA or DCOM)
- **Virtual proxy**
  - Stand-in for an object that is expensive to implement or completely access
  - Example – image over the net
  - May be able to access some state (e.g., geometry) at low cost
  - Defer other high costs until it must be incurred
- **Protection proxy**
  - Control access to the "real" object
  - Different proxies provide different rights to different clients
  - For simple tasks, can do via multiple interfaces available to clients
  - For more dynamic checking, need a front-end such as a proxy

Variants of Proxies

- Firewall proxy – controls access to a set of network resources protecting the subject from "bad" clients
- Smart Reference Proxy – provides additional actions whenever a subject is referenced, such as counting the number of references to an object
- Caching Proxy – provides temporary storage for results of operations that are expensive. It can allow multiple clients to share the results to reduce computation or network latency
- Synchronization Proxy – provides safe access to a subject from multiple threads
- Complexity hiding Proxy – hides the complexity of and controls access to a complex set of classes. Sometimes called as Façade proxy for obvious reasons
- Copy-on-Write Proxy – controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of the Virtual proxy.

Consequences

- Advantages traded-off against cost of extra level of indirection
- Can hide information about the real object
  - Where it is (remote proxy)
  - When it is loaded (virtual proxy)
  - How it is accessed (protection proxy)
- The proxy interface is either:
  - The same as the real object
  - A subset of the real object

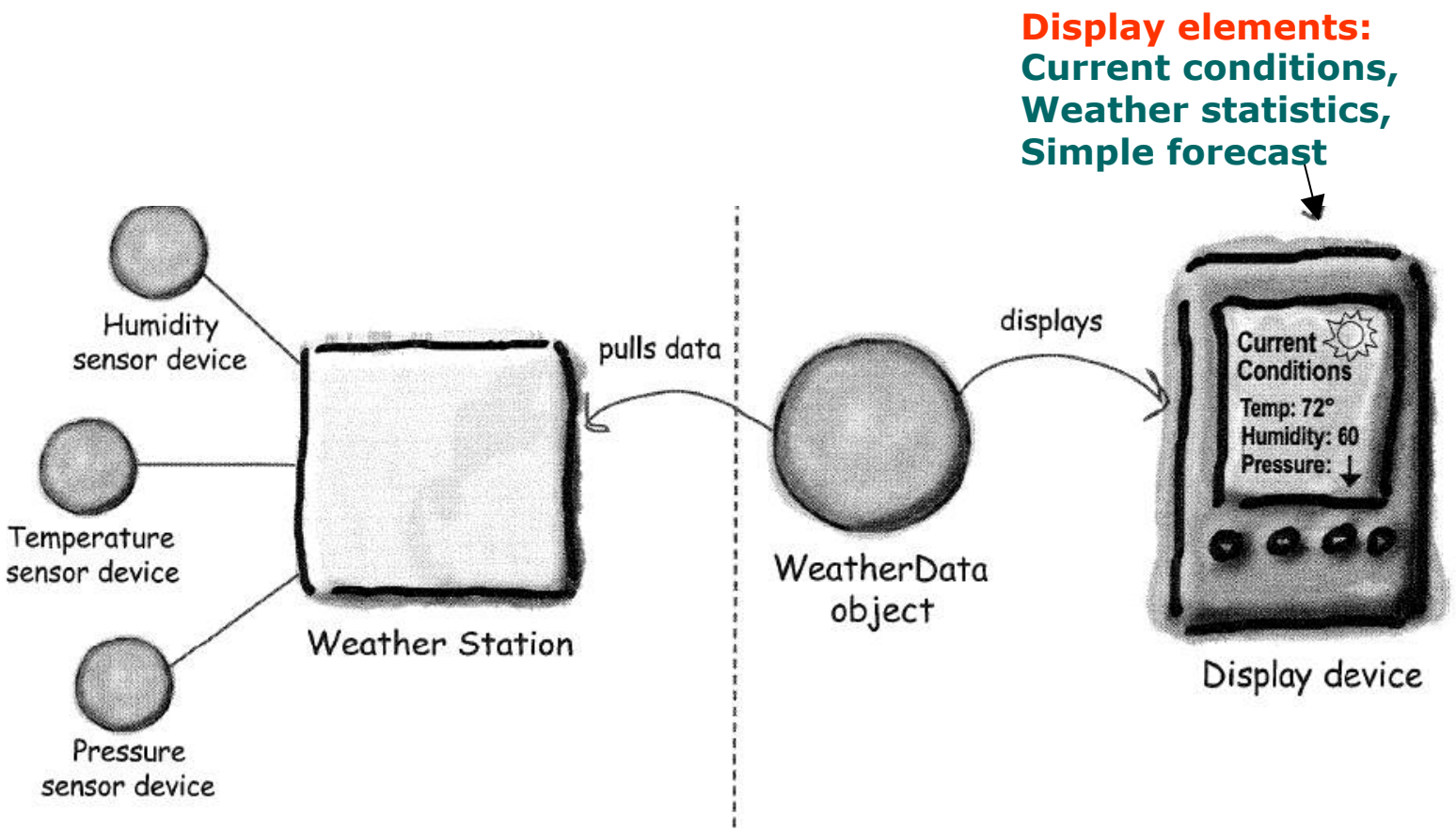
Proxies/Adapters/Facades

- Proxies and Adapters both place a stand-in object between the client and the real object
- Adapters do so to change the real object's interface
- Proxies do so to optimize access to the object via the same interface.
- Facades ease the use of sub-systems of objects

Observer pattern

Acknowledgement: Freeman & Freeman

Weather Monitoring application



The Problem

- Given
  - Clusters of related classes
  - Tight connections within each cluster of classes
  - Loose state dependency between clusters
- Desired
  - Keep each cluster state consistent when state changes in cluster it depends on
  - Provide isolation such that changed cluster has no knowledge of specifics of dependent clusters

Example: UI & Application

- Application classes represent information being manipulated.
- UI provides way to view and alter application state.
- May have several views of state (charts, graphs, numeric tables).
- Views may be added at any time.
- How to tell views when application state has changed?

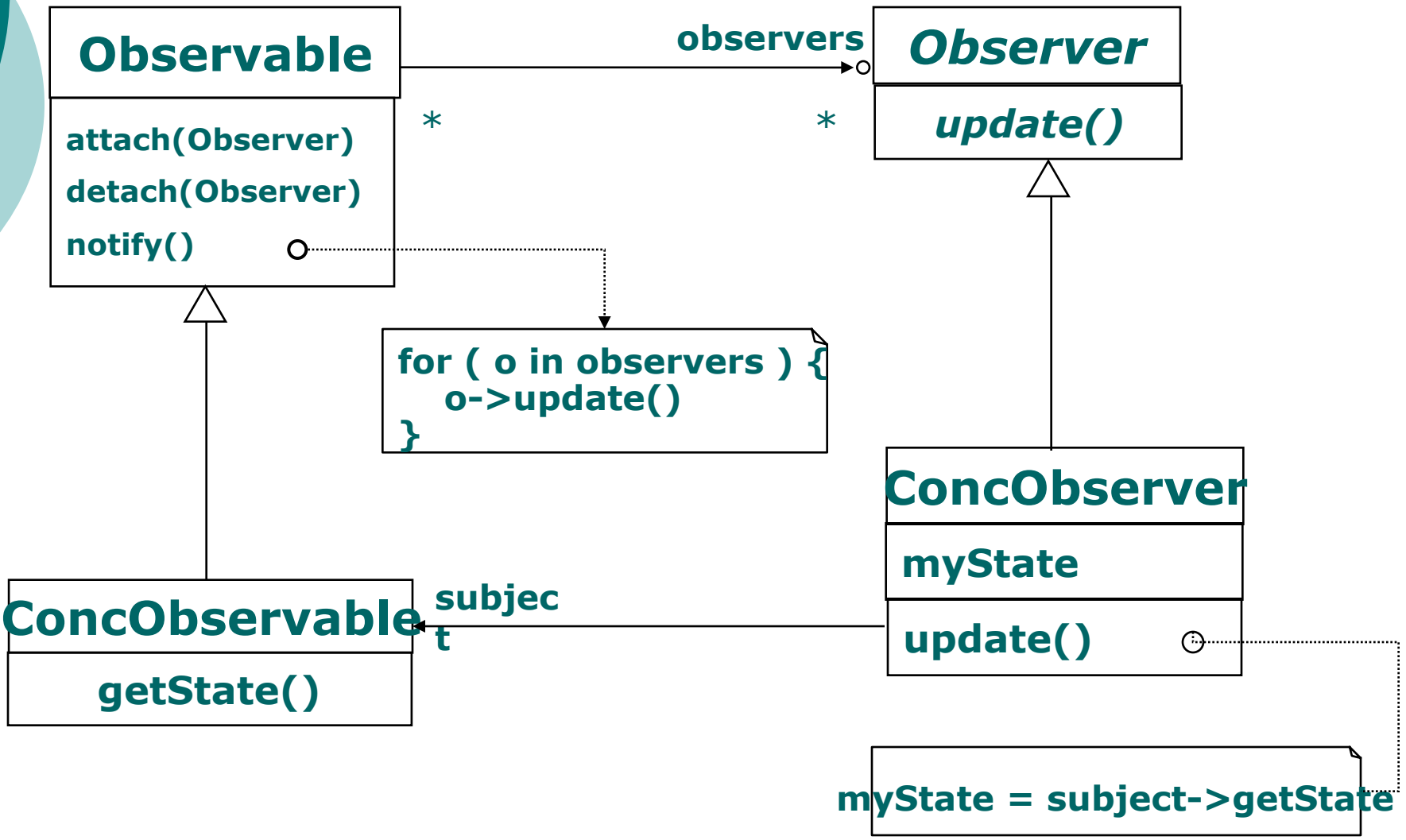
Observer Pattern

- The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

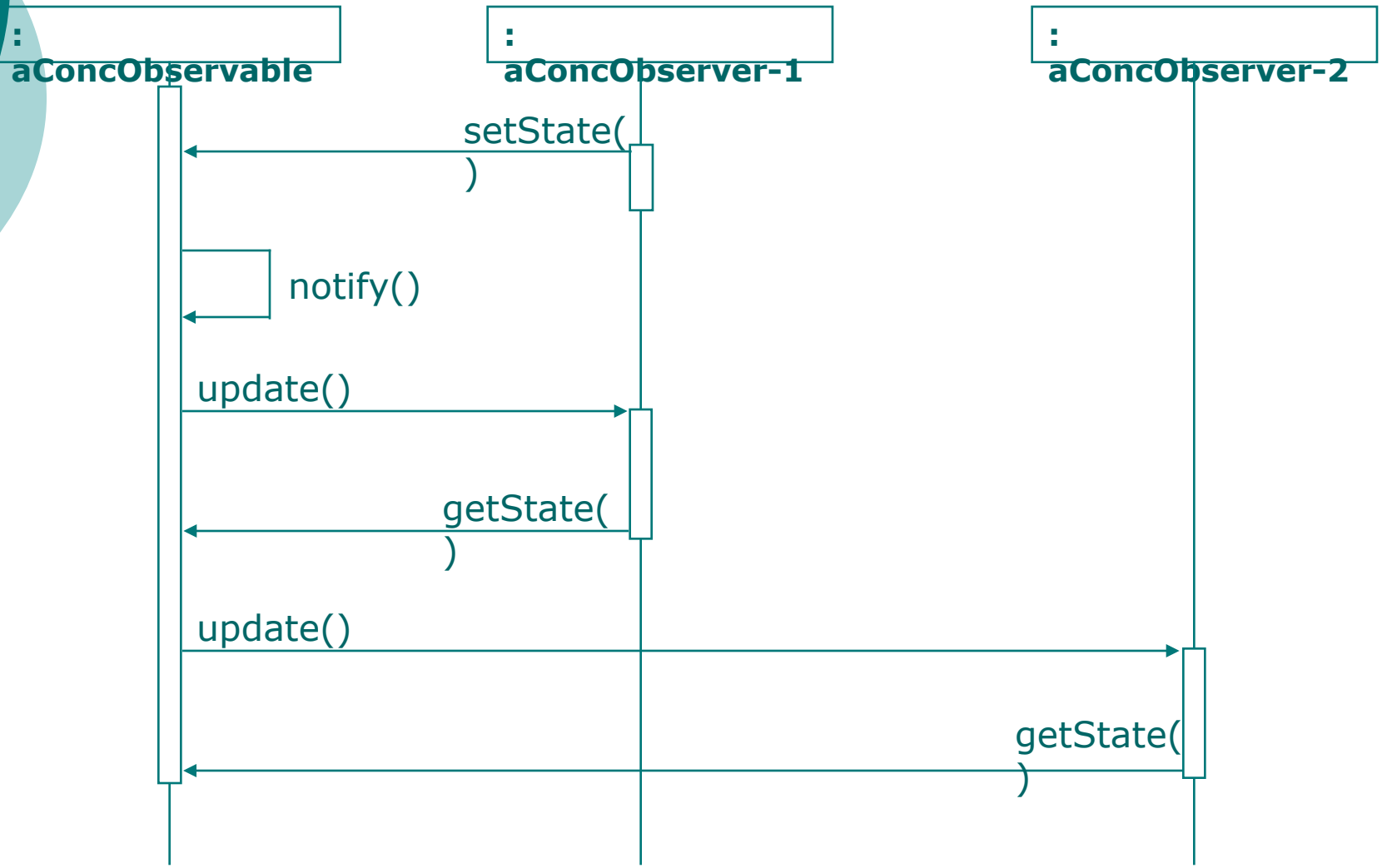
Observer/Subject
(AKA: publish/subscribe)

- Observables: objects with interesting state changes
- Observers: objects interested in state changes
- Observers *register* interest with observable objects.
- Observables *notify* all registered observers when state changes.

Observer Pattern - Class diagram



Interaction Diagram



When to Use Observer

- Two subsystems evolve independently but must stay in synch.
- State change in an object requires changes in an unknown number of other objects (broadcast)
- Desire loose coupling between changeable object and those interested in the change.

Consequences

- Subject/observer coupling (loose coupling)
  - Subject only knows it has a list of observers
  - The only thing the Subject knows about an Observer is that it implements a certain interface
  - Observers can be added at any time
  - Does not know any Observer concrete class
  - Subjects don't need to be modified to add new types of Observers
  - Subjects and Observers can be reused independently
- Supports broadcast communication
  - Observables know little about notify receivers
  - Changing observers is trivial
- Unexpected & cascading updates
  - change/notify/update -> change/notify/update
  - May be hard to tell *what* changed

Observer Pattern – Key points

- Observer pattern defines a one-to-many relationship between objects
- Subjects/Observables update observers using a common interface
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement observer interface
- You can PUSH or PULL data from the Observable when using the pattern (pull is considered more "correct")
- Don't depend on specific order of notification for your observers

Composite pattern

Acknowledgement: Freeman & Freeman

The Problem

- Problem
  - Have simple primitive component classes that collect into larger *composite* components
- Desire
  - Treat composites like primitives
  - Support composite sub-assemblies
  - Operations (usually) recurse to subassemblies
- Solution
  - Build composites from primitive elements

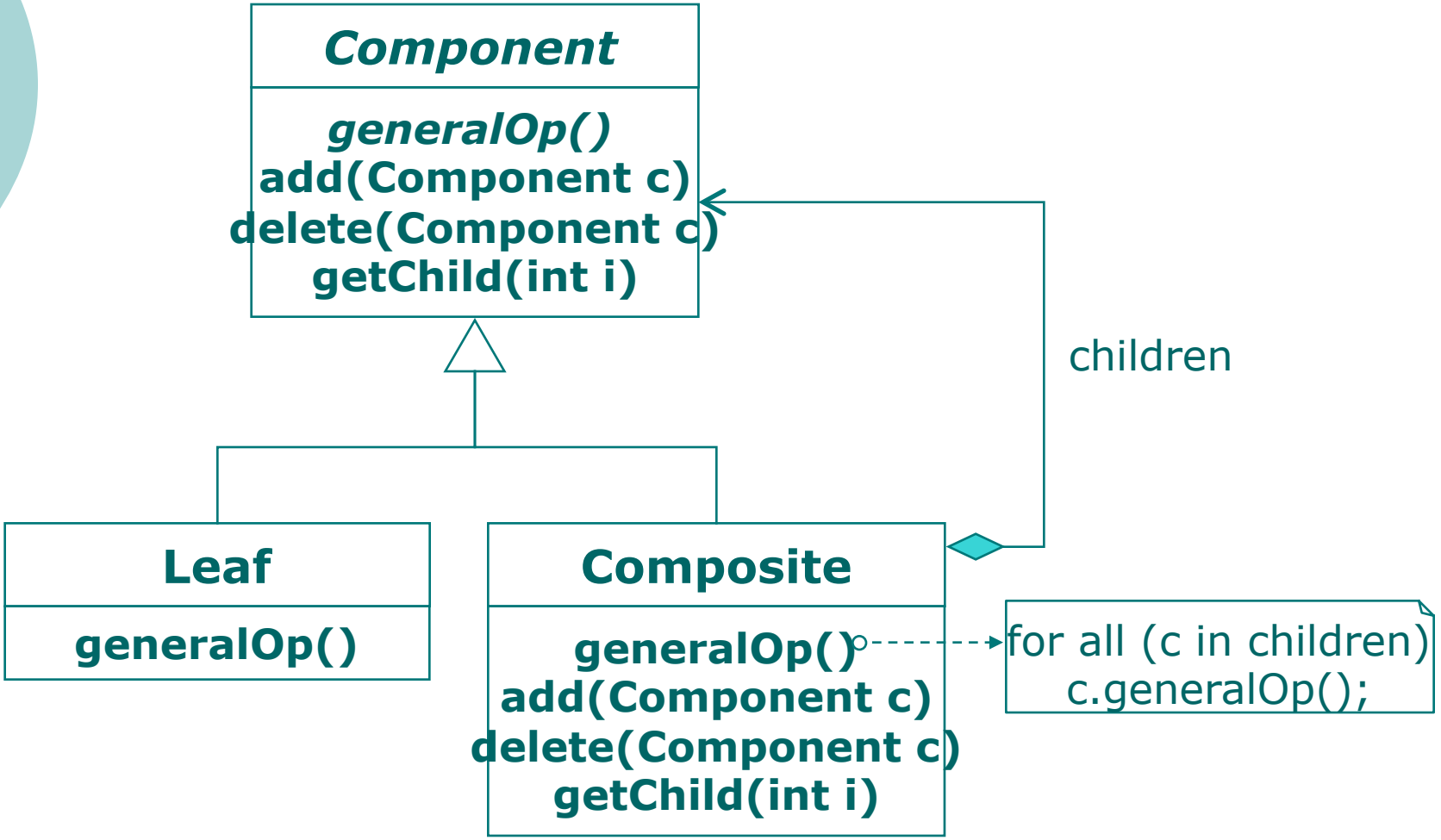
Examples - 1

- File systems
  - Primitives = text files, binary files, device files, etc.
  - Composites = directories (w/subdirectories)
- Make file dependencies
  - Primitives = leaf targets with no dependents
  - Composites = targets with dependents
- Menus
  - Primitives = menu entries
  - Composites = menus (w/submenus)

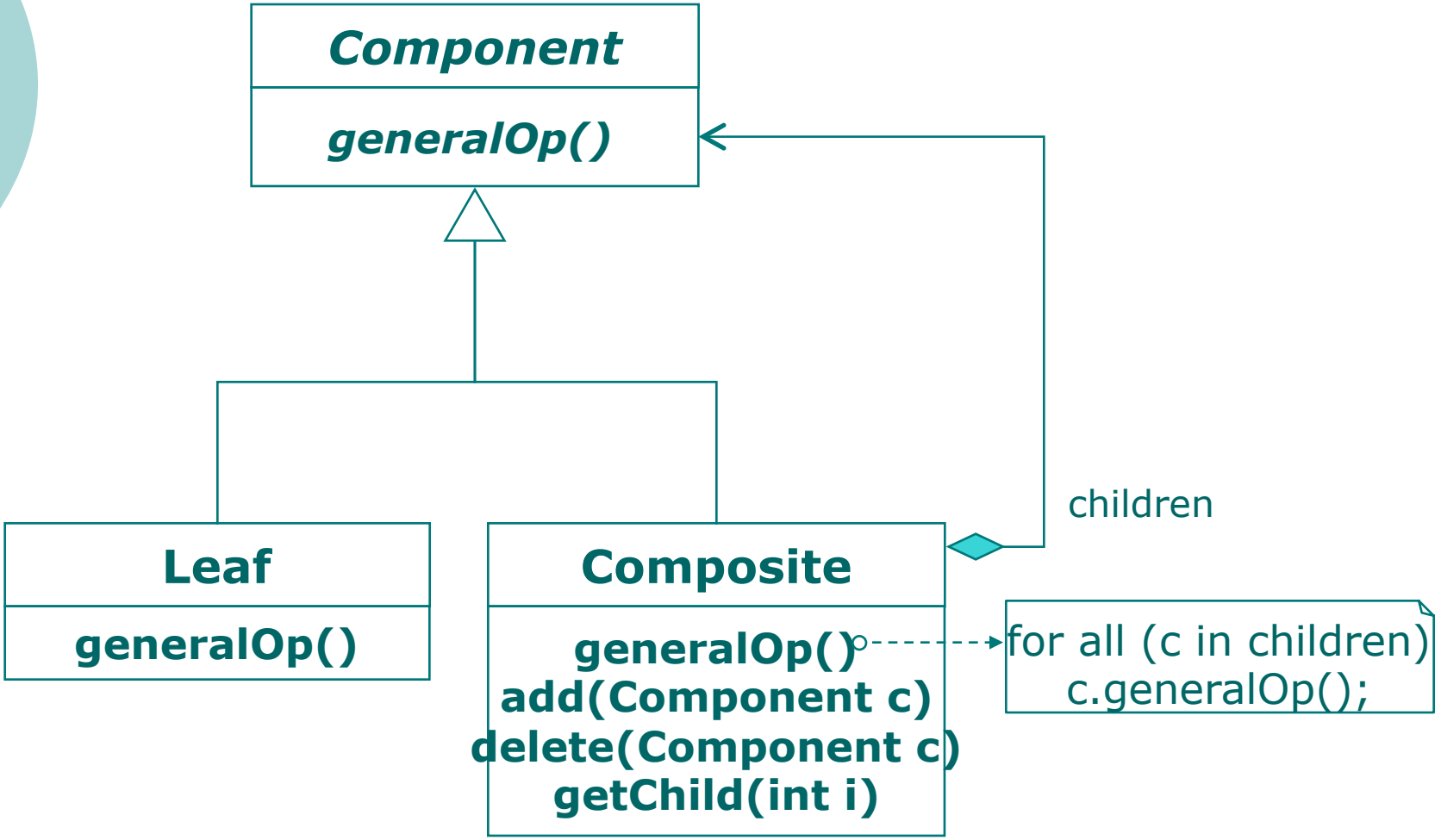
Examples - 2

- GUI Toolkits
  - Primitives = basic components (buttons, textareas, listboxes, etc).
  - Composites = frames, dialogs, panels.
- Drawing Applications
  - Primitives = lines, strings, polygons, etc.
  - Composites = groupings treated as unit.
- HTML/XML
  - Pages as composites of links (hypertext)
  - Pages as collections of paragraphs (with subparagraphs for lists, etc.)

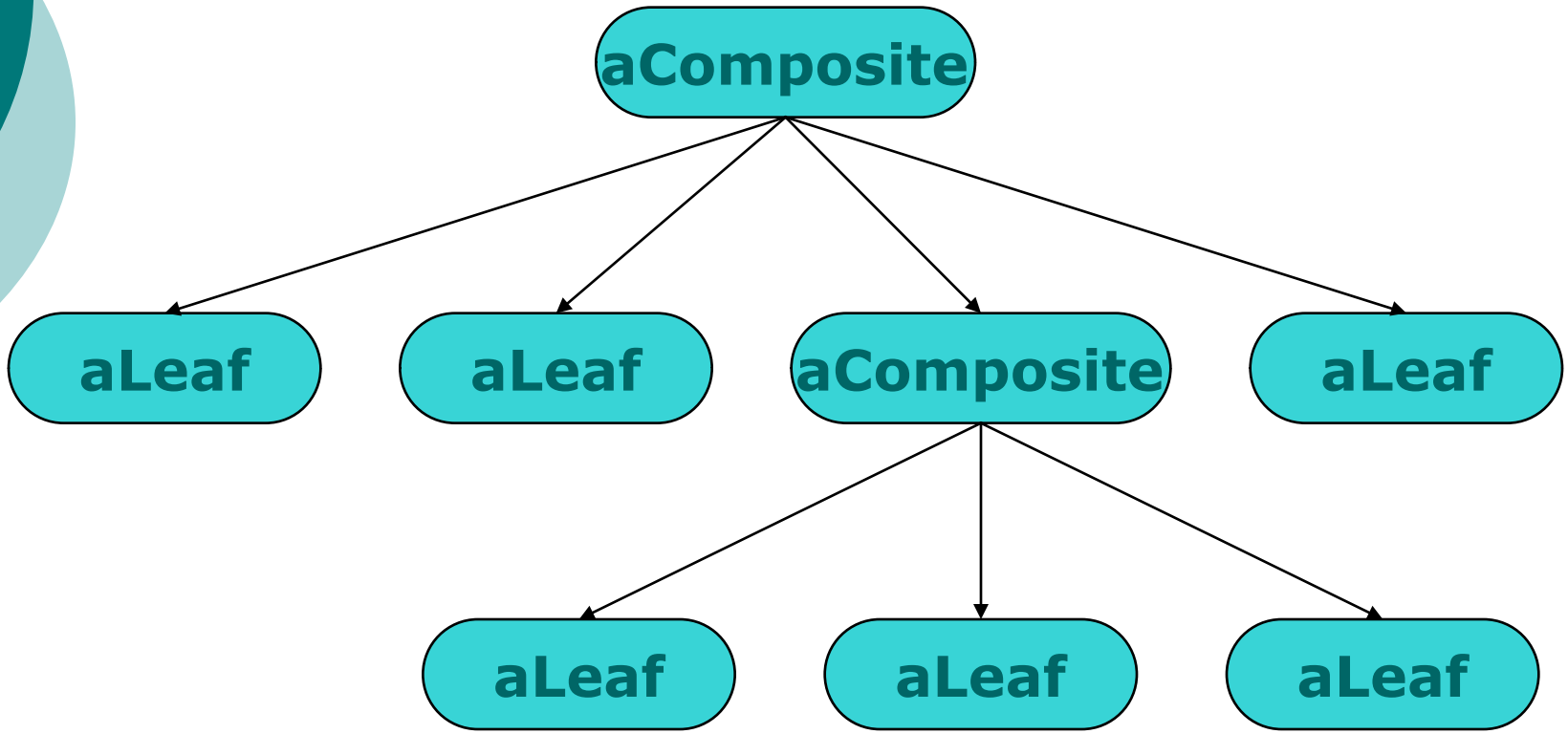
Class Diagram (Alternative 1)



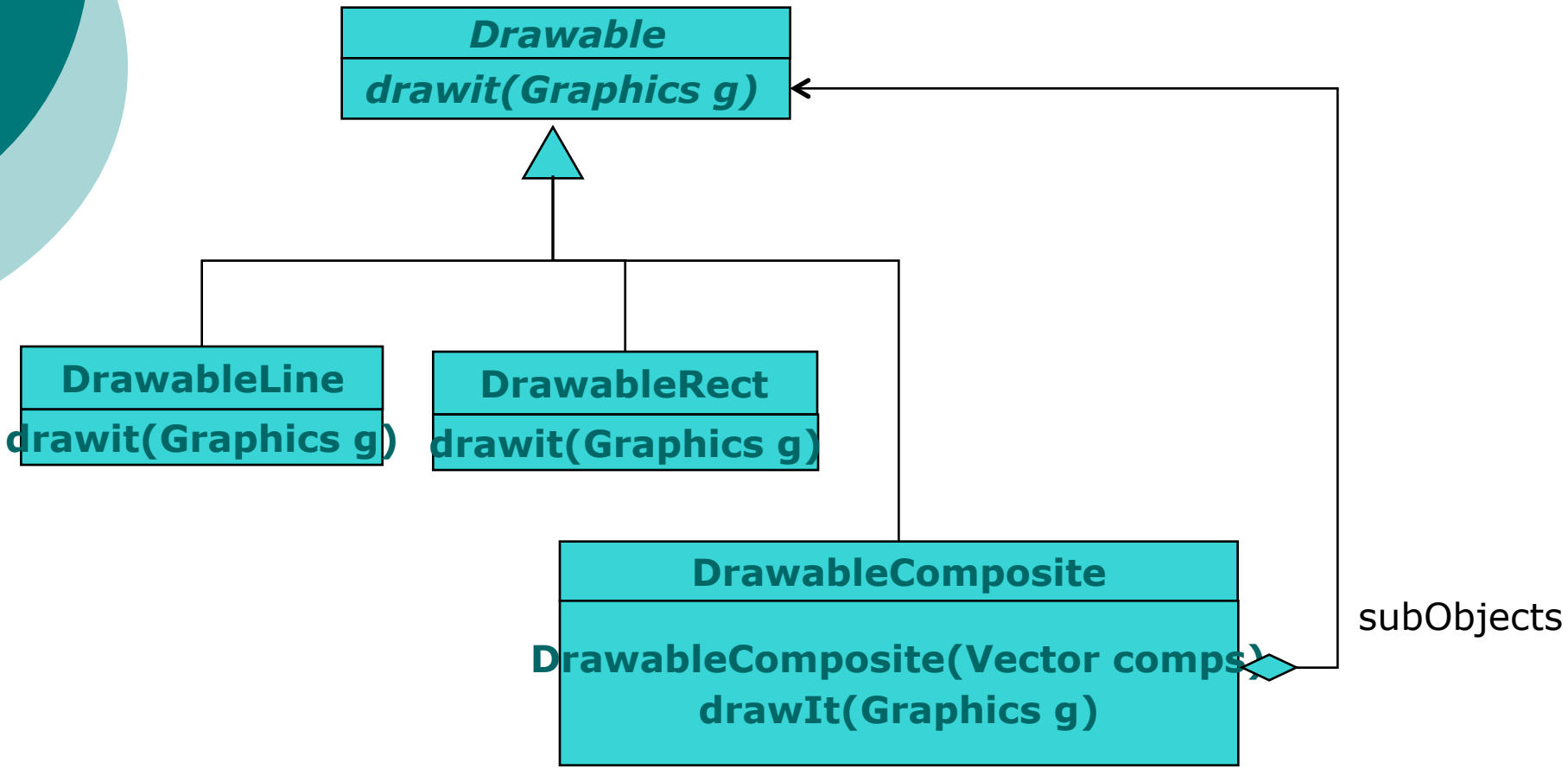
Class Diagram (Alternative 2)



Example Object Structure



Example: Drawable Figures



Discussion

- Use to model whole/part hierarchies
- Clients (usually) ignore differences between primitives & composites
- Clients access (most) components via the generic interfaces
  - Primitives implement request directly
  - Composites can handle directly or forward
- Arbitrary composition to indefinite depth
  - Tree structure – no sharing of nodes
  - General digraph – supports sharing, multiple parents – be careful!
- Eases addition of new components
  - Almost automatic
- Overly general design?