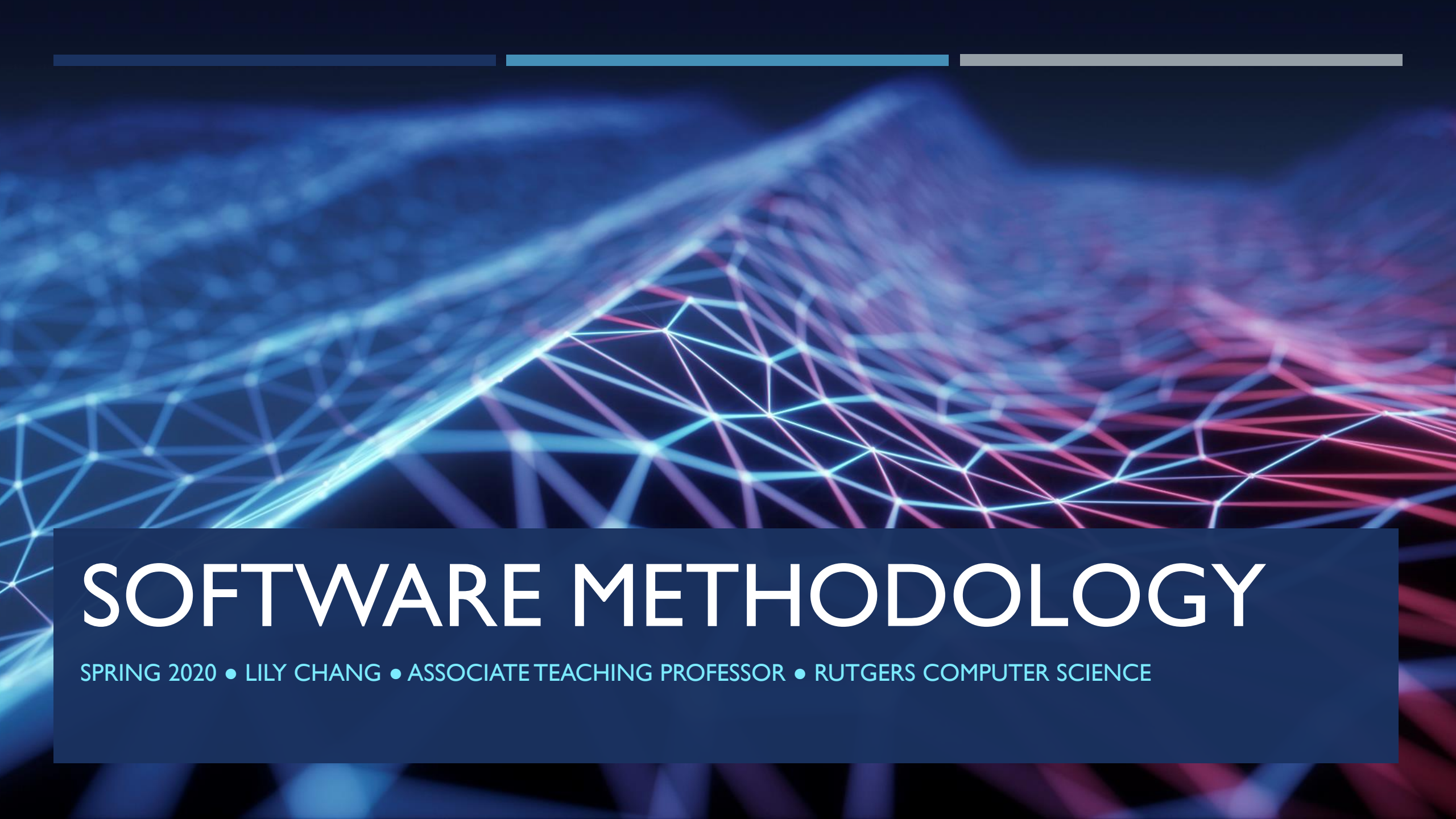


SOFTWARE METHODOLOGY

SPRING 2020 • LILY CHANG • ASSOCIATE TEACHING PROFESSOR • RUTGERS COMPUTER SCIENCE

DESIGN PATTERNS

WHAT IS DESIGN PATTERN?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

--Christopher Alexander

WHAT IS DESIGN PATTERN?

- Designing object-oriented software is hard, and designing reusable object-oriented software is even harder
  - You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them
  - Your design should be specific to the problem at hand but also general enough to address future problems and requirements
  - You'll find recurring patterns of classes and communicating objects in many object-oriented systems, and these patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable

ESSENTIAL COMPONENTS OF A PATTERN

- The pattern name
  - Use to describe a design problem, its solutions, and consequences in a word or two
  - A vocabulary for communication
- The problem
  - Describes when to apply the pattern
  - Explains the problem and its context; such as specific design problems, how to represent algorithms as objects, or symptomatic of an inflexible design
- The solution
  - Describes the elements that make up the design, their relationships, responsibilities, and collaborations.
  - Doesn't describe a particular concrete design or implementation,
  - A pattern is like a template that can be applied in many different situations, thus, provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
- The consequences
  - The results and trade-offs of applying the pattern.
  - Are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

OBJECT ORIENTED DESIGN PATTERNS

- Reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability
- The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context
- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design
- A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.
- Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

CLASSIFICATION OF DESIGN PATTERNS

- Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them.
- The classification helps you learn the patterns faster, and it can direct efforts to find new patterns as well
- The design patterns is classified by two criteria .
  - The first criterion, called **purpose**, reflects what a pattern does.
  - The second criterion, called **scope**, specifies whether the pattern applies primarily to classes or to objects.

CLASSIFICATION OF DESIGN PATTERNS

- Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

CLASSIFICATION OF DESIGN PATTERNS

- Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.
- Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object.
- The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects.
- The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

DESIGN FOR CHANGE

- The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.
- To design the system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting.
- Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.
- Design patterns help you avoid this by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

COMMON CAUSES OF REDESIGN

- Creating an object by specifying a class explicitly.
  - Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes.
  - To avoid it, create objects indirectly.
    - The Design patterns Abstract Factory, Factory Method, Prototype can be used to address the problems
- Dependence on specific operations.
  - When you specify a particular operation, you commit to one way of satisfying a request.
  - By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.
  - Design patterns Chain of Responsibility, Command can be used to address the problems

COMMON CAUSES OF REDESIGN

- Dependence on hardware and software platform.
  - External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms.
  - Software that depends on a particular platform will be harder to port to other platforms.
  - It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies.
  - Design patterns Abstract Factory, Bridge can be used to address the problems

COMMON CAUSES OF REDESIGN

- Dependence on object representations or implementations.
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes.
  - Hiding this information from clients keeps changes from cascading.
  - Design patterns Abstract Factory, Bridge, Memento, Proxy can be used to address the problems
- Algorithmic dependencies.
  - Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes.
  - Therefore algorithms that are likely to change should be isolated. Design patterns Builder, Iterator, Strategy, Template Method, Visitor can be used to address the problems

COMMON CAUSES OF REDESIGN

- Tight coupling.
  - Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other.
  - Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain.
  - Loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily.
  - Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems; such as Abstract Factory, Bridge, Chain of Responsibility

COMMON CAUSES OF REDESIGN

- Extending functionality by subclassing.
  - Customizing an object by subclassing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.).
  - Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And subclassing can lead to an explosion of classes, because you might have to introduce many new subclasses for even a simple extension.
  - Object composition in general and delegation in particular provide flexible alternatives to inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand.
  - Many design patterns produce designs in which you can introduce customized functionality just by defining one subclass and composing its instances with existing ones.
    - Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

COMMON CAUSES OF REDESIGN

- Inability to alter classes conveniently.
  - Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library). Or maybe any change would require modifying lots of existing subclasses.
  - Design patterns offer ways to modify classes in such circumstances.
    - Adapter, Decorator, Visitor.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

HOW TO SELECT A DESIGN PATTERN

DESIGN PATTERNS TO BE DISCUSSED

- Creational
  - Abstract Factory
  - Singleton
- Model-View-Controller (MVC)
  - Observer (Behavioral)
  - Composite (Structural)
  - Strategy (Behavioral)

CREATIONAL

- Creational design patterns abstract the instantiation process.
- They help make a system independent of how its objects are created, composed, and represented.
- A class creational pattern uses inheritance to vary the class that's instantiated
- An object creational pattern will delegate instantiation to another object.

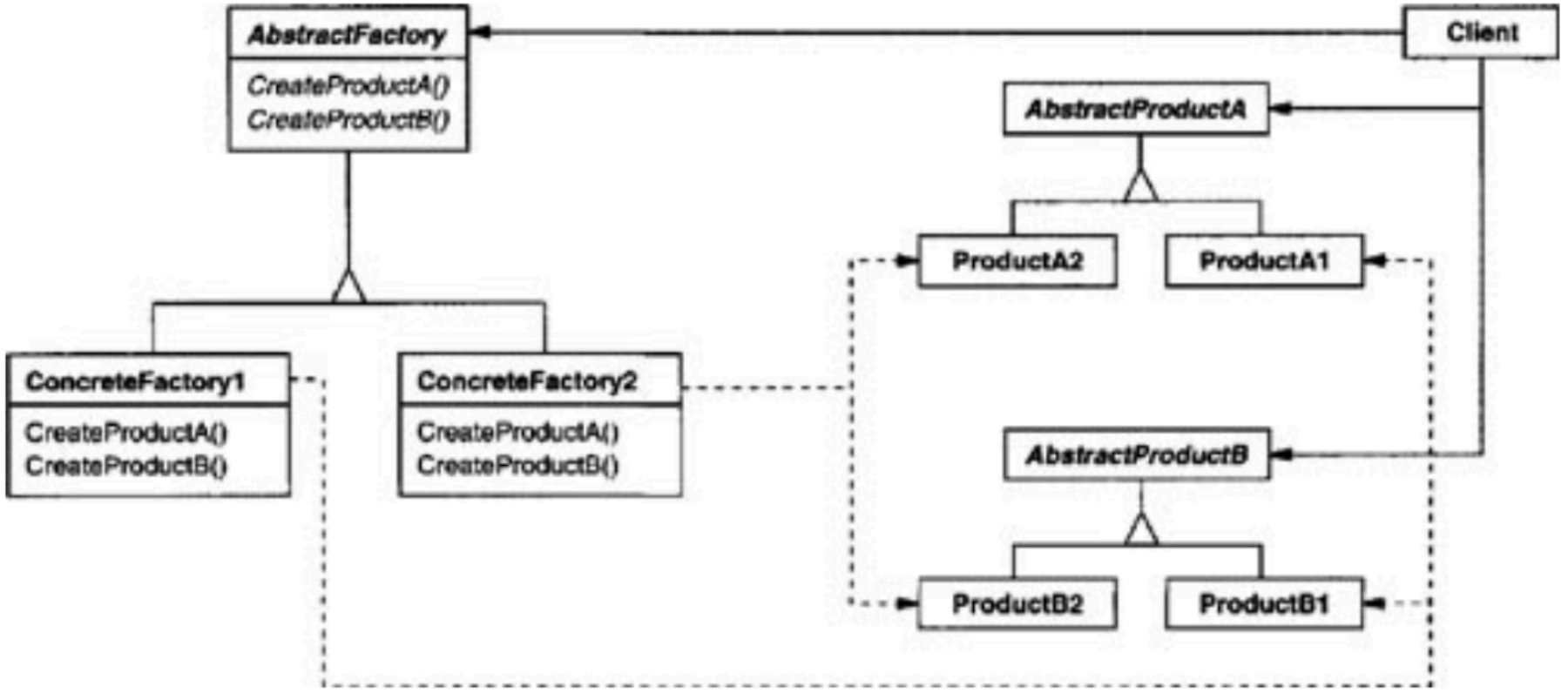
ABSTRACT FACTORY

- Consider a user interface toolkit that supports multiple look-and-feel standards
- Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.
- To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel.
- Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.
- We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget.

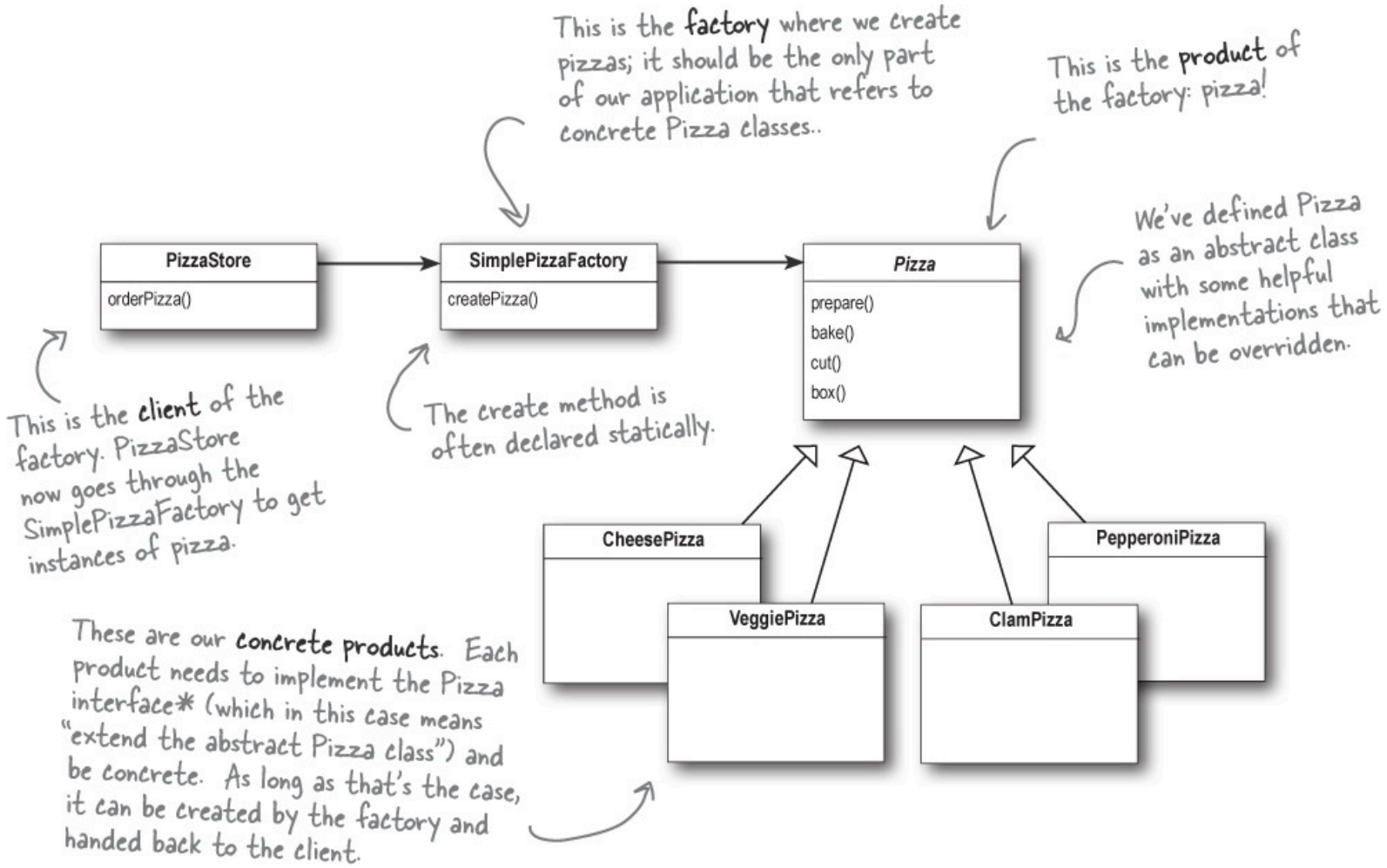
ABSTRACT FACTORY – APPLICABILITY

- When a system should be independent of how its products are created, composed, and represented; for example, the pizza store.
- When a system should be configured with one of multiple families of products; for example, different types of pizzas
- When a family of related product objects is designed to be used together, and you need to enforce this constraint.

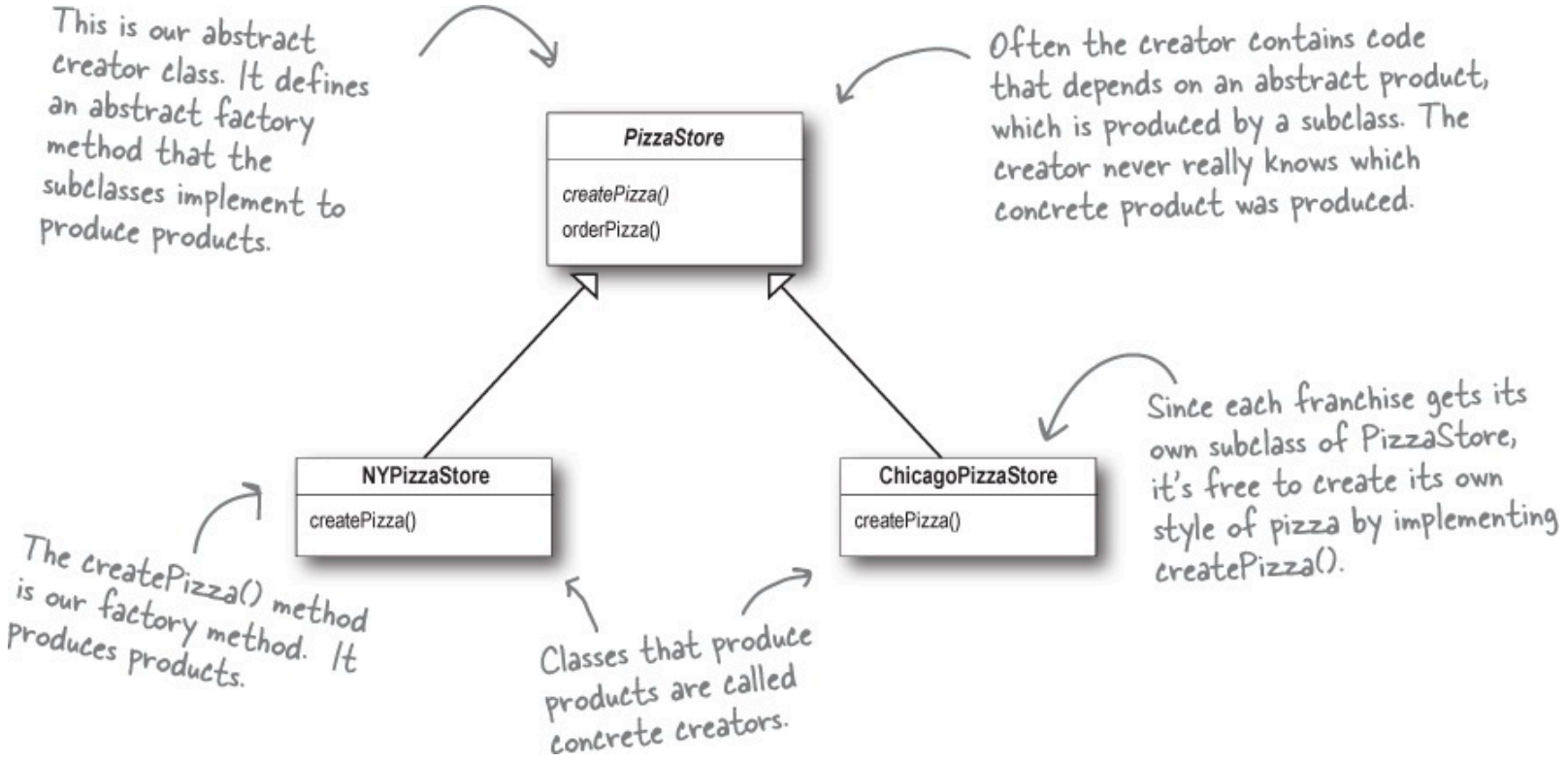
ABSTRACT FACTORY – STRUCTURE



ABSTRACT FACTORY – PIZZA STORE



ABSTRACT FACTORY – PIZZA STORE



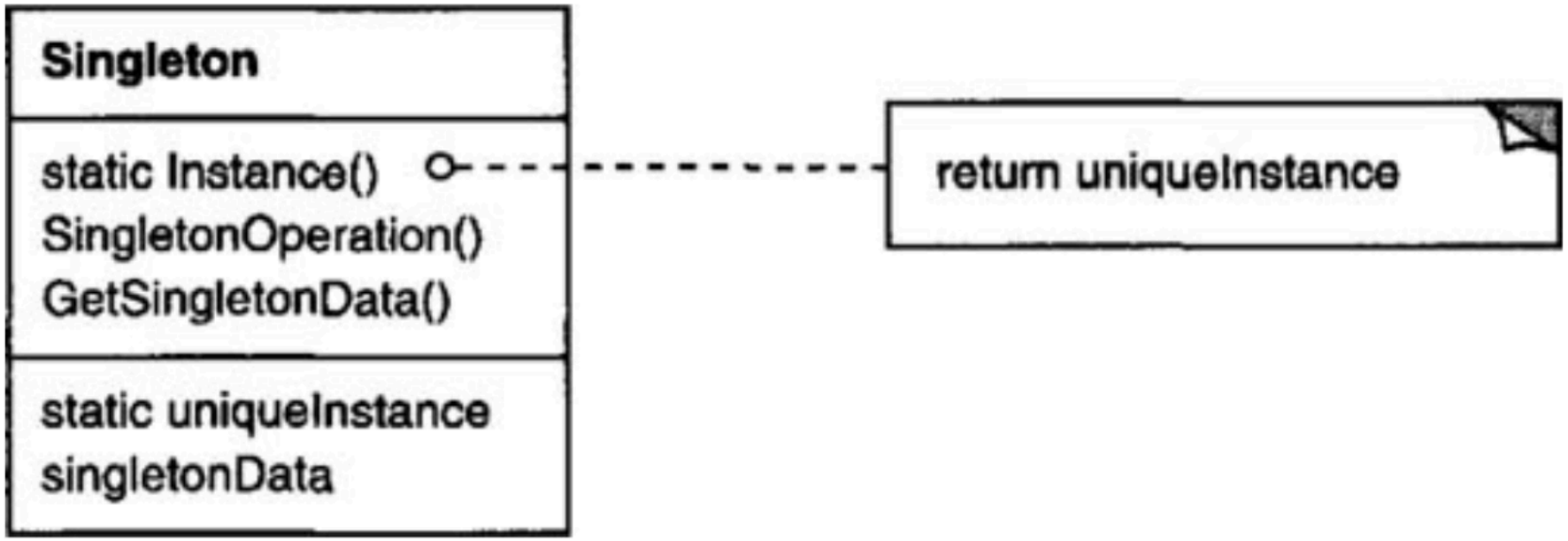
SINGLETON DESIGN PATTERN

- It's important for some classes to have exactly one instance; for example,
  - Although there can be many printers in a system, there should be only one printer spooler; there should be only one file system and one window manager; a digital filter will have one A/D converter; an accounting system will be dedicated to serving one company.
- How do we ensure that a class has only one instance and that the instance is easily accessible?
  - A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

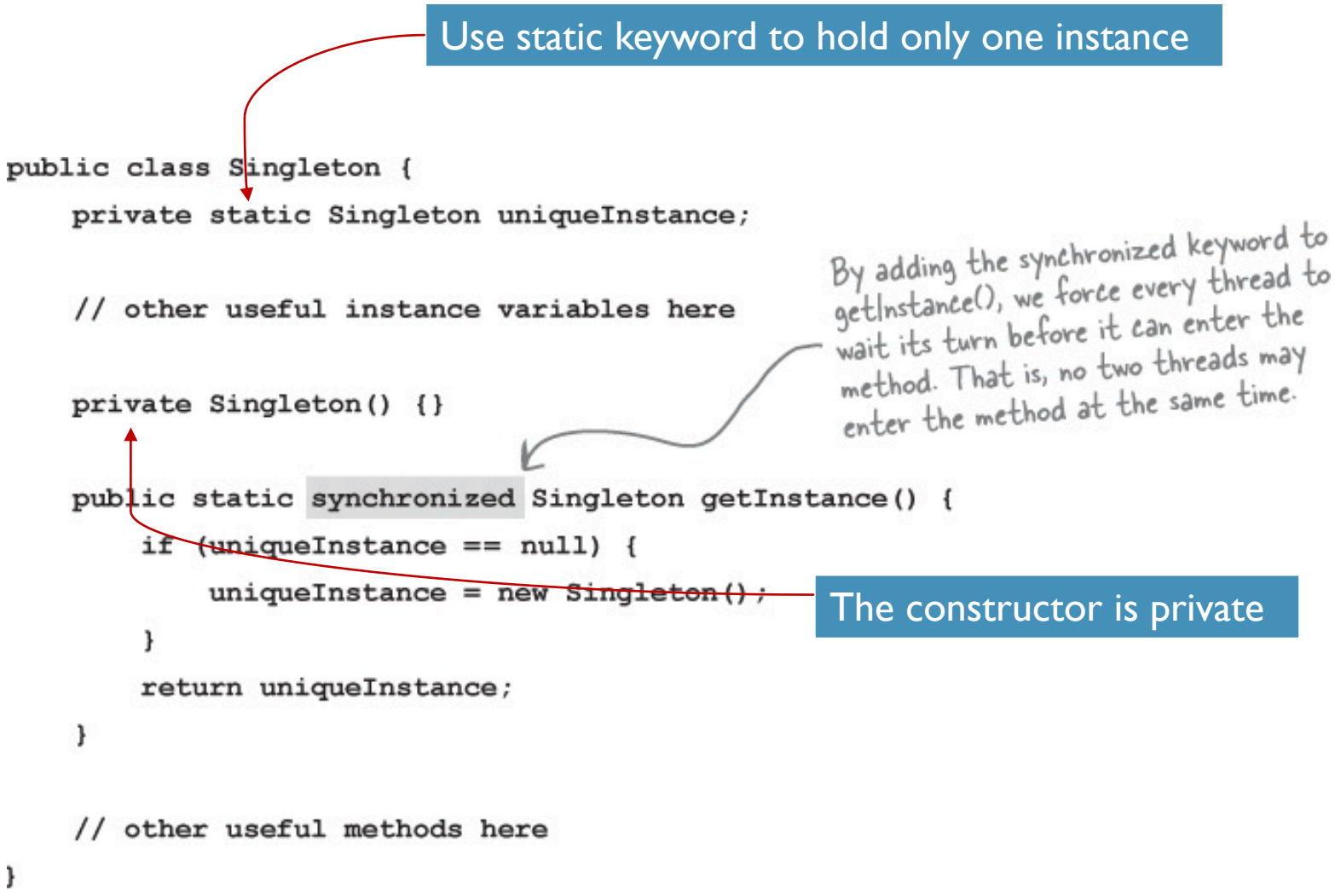
SINGLETON PATTERN – APPLICABILITY

- Use the Singleton pattern when
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
    - Ensure a class has only one instance, and provides a global point of access to it.
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

SINGLETON PATTERN – STRUCTURE



SINGLETON PATTERN - EXAMPLE

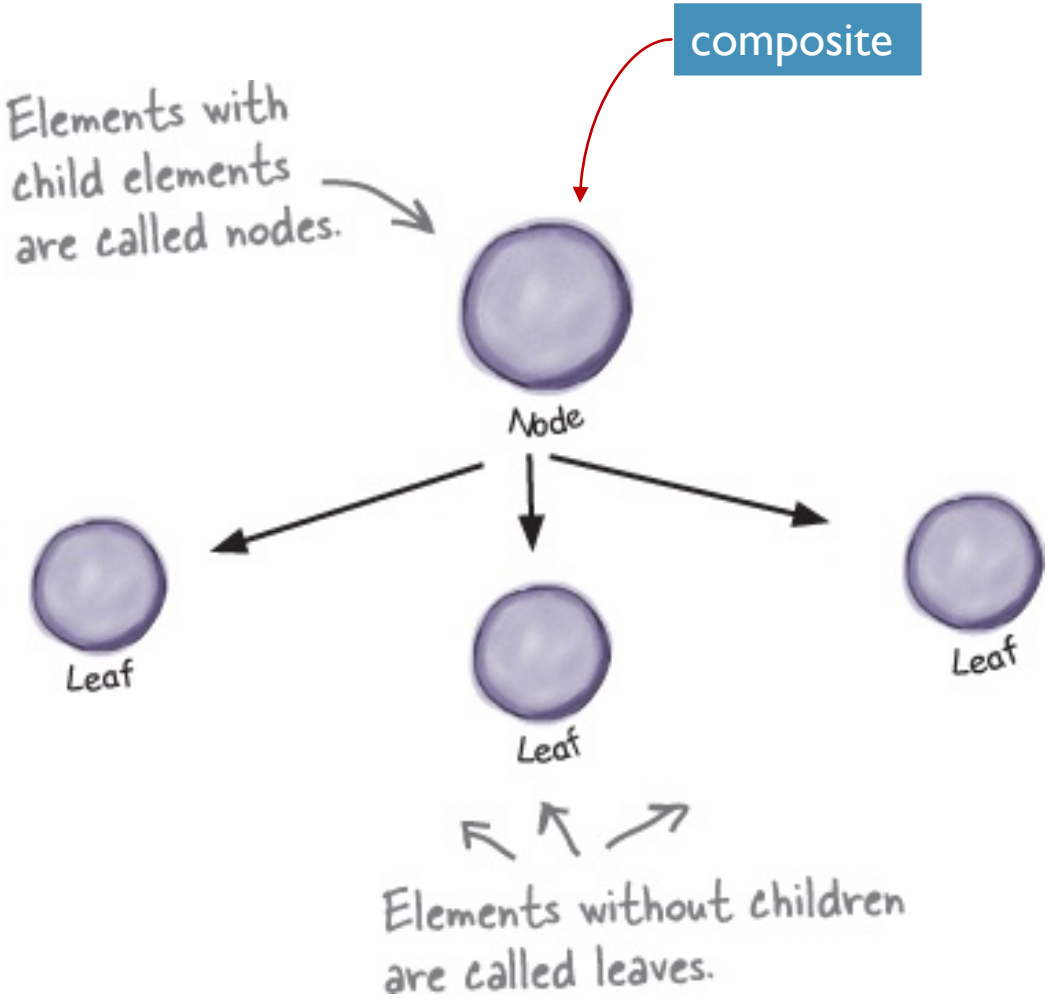


STRUCTURAL PATTERNS

- Structural patterns are concerned with how classes and objects are composed to form larger structures
- This pattern is particularly useful for making independently developed class libraries work together

COMPOSITE PATTERN

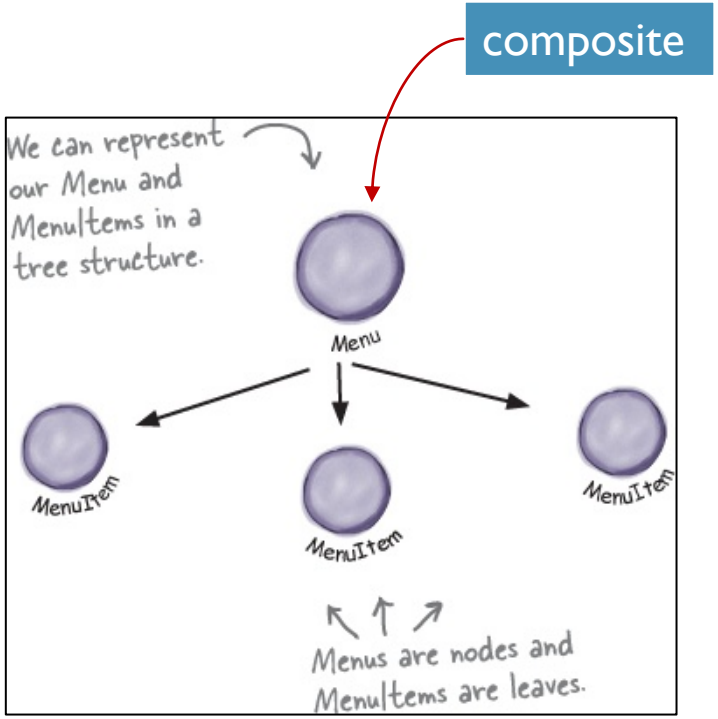
- The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

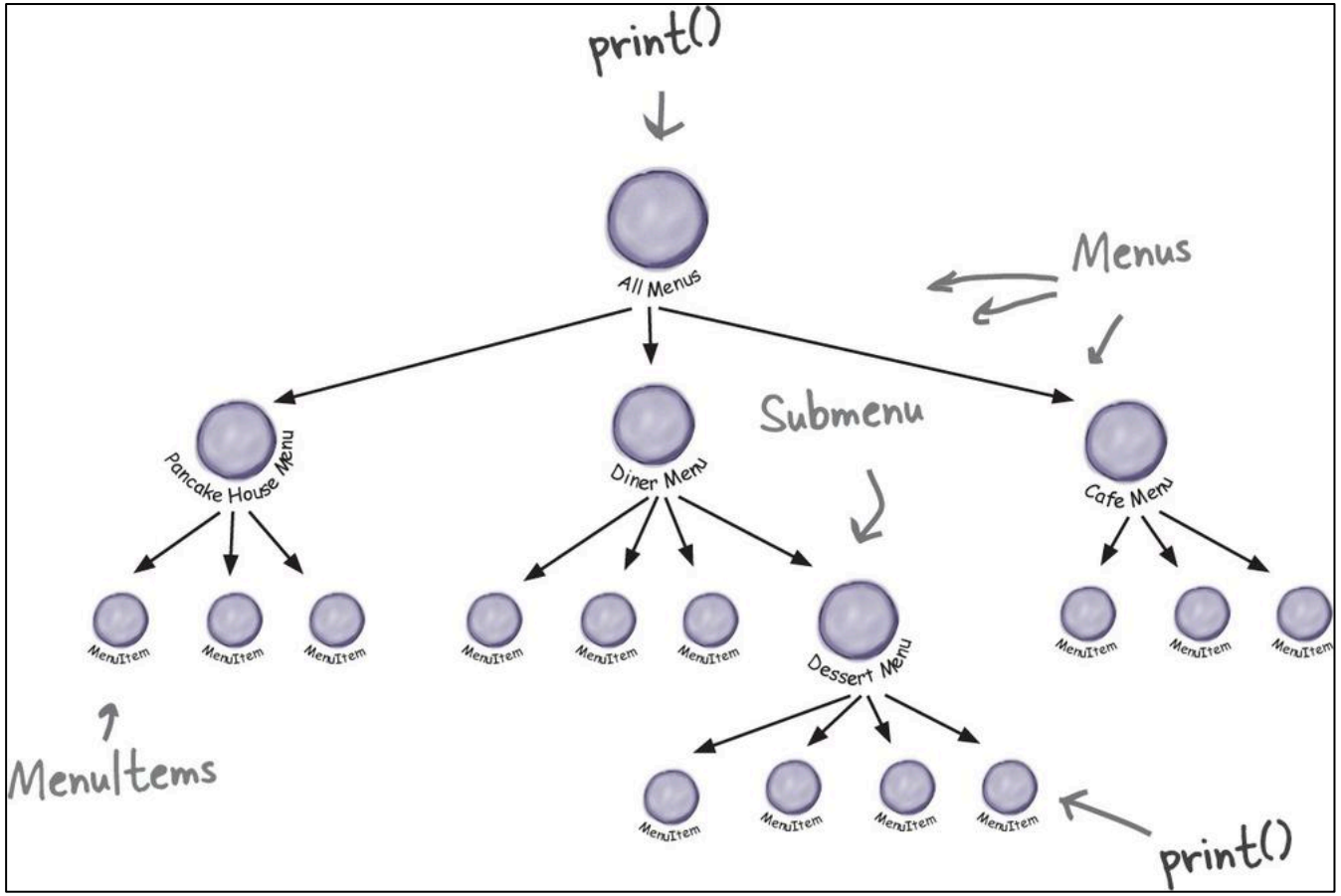


COMPOSITE PATTERN - APPLICABILITY

- Use the Composite pattern when
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- Imagine a graphical user interface; there you'll often find a top level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the components that contain other components, composite objects, and components that don't contain other components, leaf objects

COMPOSITE PATTERN – EXAMPLE



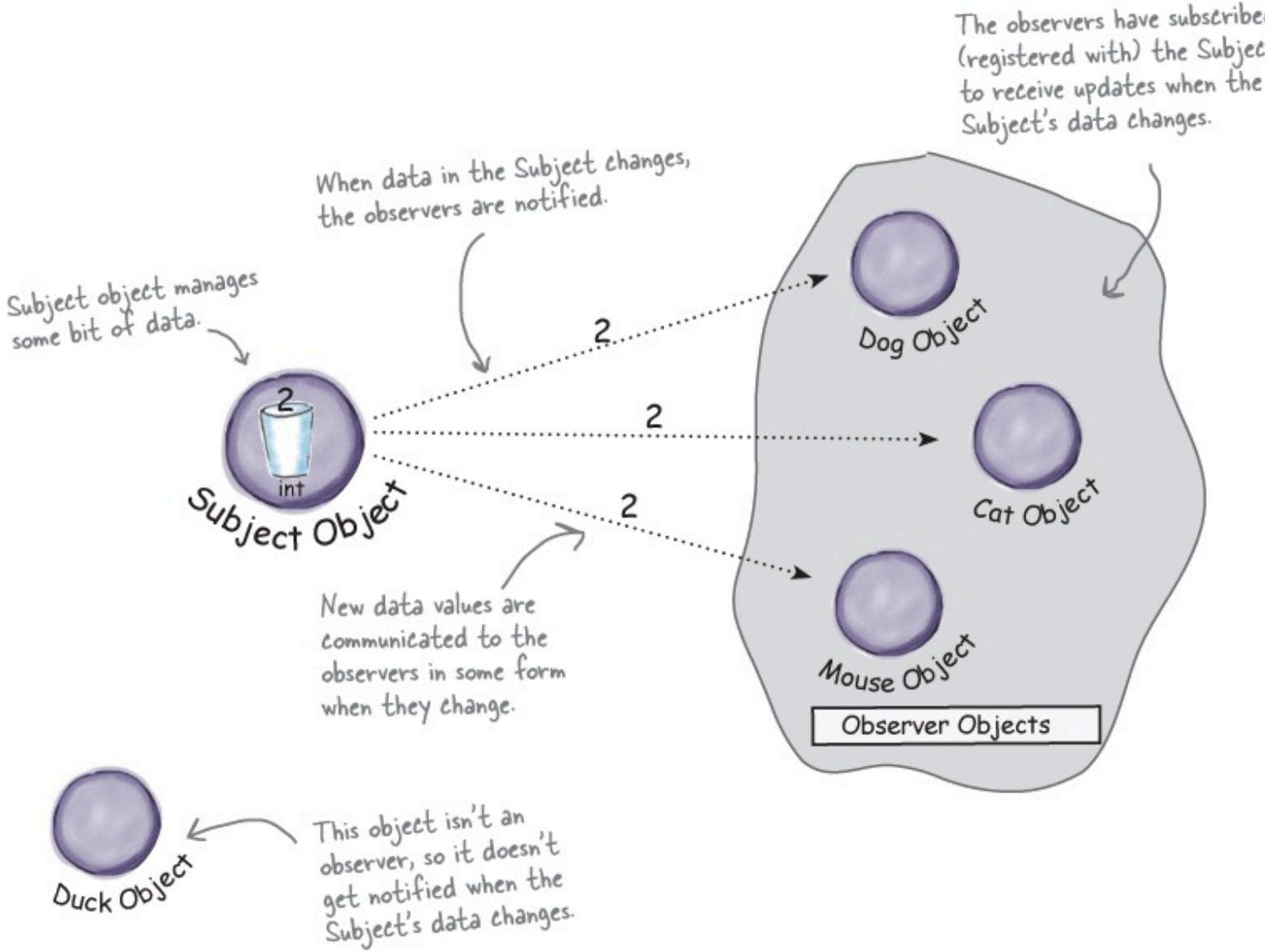


BEHAVIORAL PATTERNS

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
- These patterns characterize complex control flow that's difficult to follow at run-time.
- They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

OBSERVER – ALSO KNOWN AS DEPENDENTS, PUBLISH-SUBSCRIBE

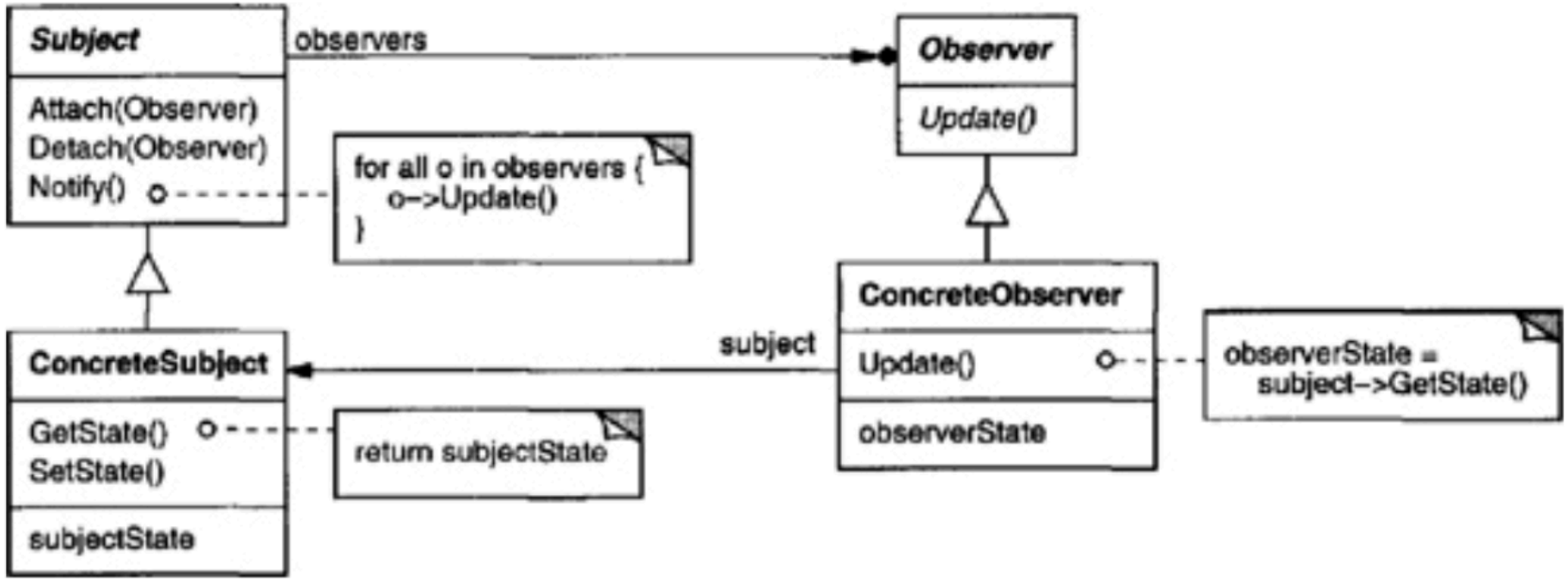
- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- Observer pattern intent to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The Observer pattern defines and maintains a dependency between objects.
- The classic example of Observer is Model/View/Controller, where all views of the model are notified whenever the model's state changes.


The observers have subscribed (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
Subject object manages some bit of data.
2
int
Subject Object
2
2
2
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

OBSERVER PATTERN – APPLICABILITY

- Use the Observer pattern in any of the following situations
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

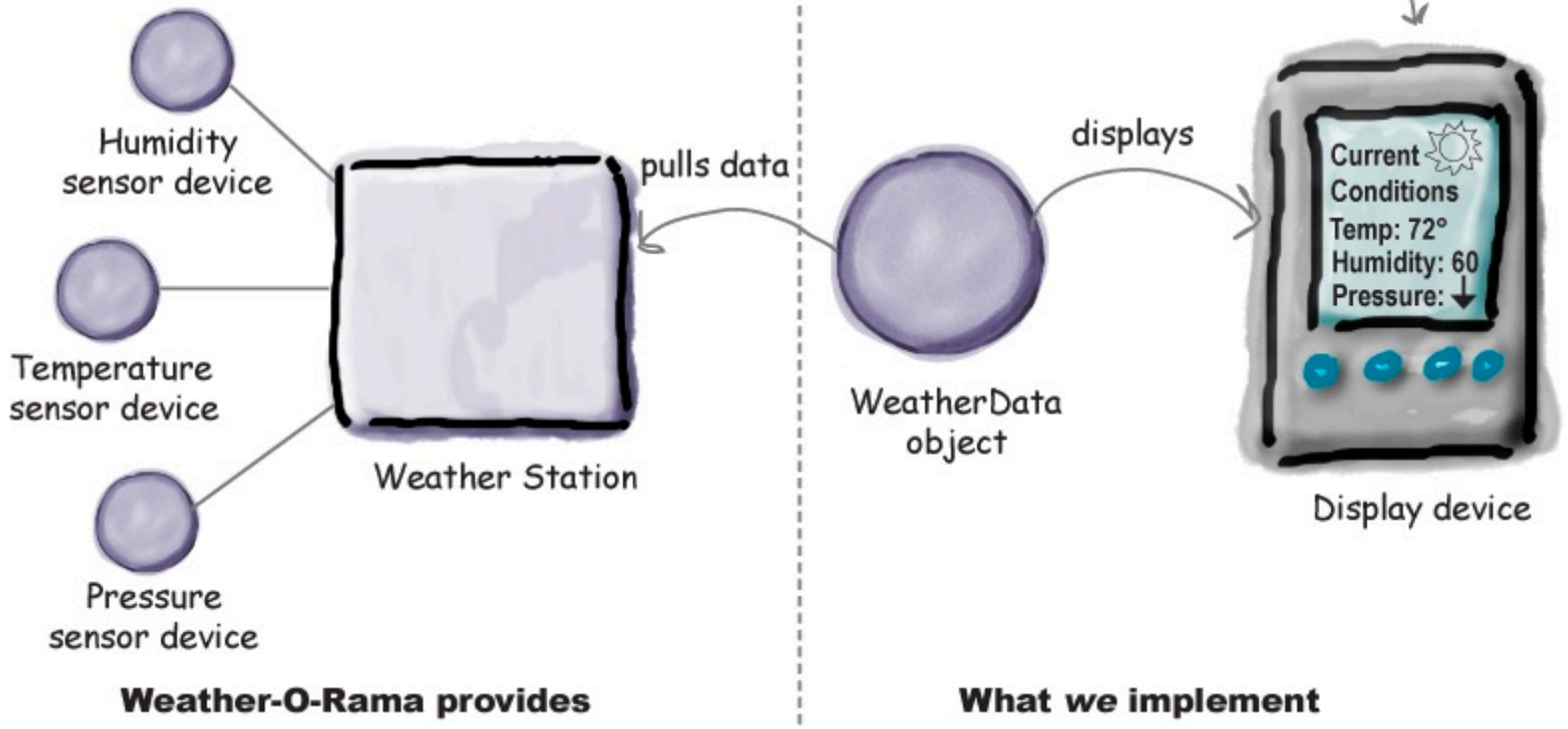
OBSERVER PATTERN – STRUCTURE



AN EXAMPLE – WEATHER MONITOR SYSTEM

```
currentConditionsDisplay.update(temp, humidity, pressure);
statisticsDisplay.update(temp, humidity, pressure);
forecastDisplay.update(temp, humidity, pressure);
```

Current Conditions is one of three different displays. The user can also get weather stats and a forecast.



AN EXAMPLE – WEATHER MONITOR SYSTEM

```
public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

```
//all observers implement update()
public void notifyObservers() {
    for (Observer observer: observers)
        observer.update(temperature, humidity, pressure);
}
```

- CurrentConditionDisplay
- StatisticsDisplay
- ForecastDisplay
- OtherDisplay

THE POWER OF LOOSE COUPLING

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
- The Observer Pattern provides an object design where subjects and observers are loosely coupled – the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer
  - We can add new observers at any time.
  - We never need to modify the subject to add new types of observers.
  - We can reuse subjects or observers independently of each other.
  - Changes to either the subject or an observer will not affect the other.

EXAMPLE – JAVAFX OBSERVABLE INTERFACE

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
`void`	**`addListener`**`(`**`InvalidationListener`**` listener)`	Adds an `InvalidationListener` which will be notified whenever the `Observable` becomes invalid.
`void`	**`removeListener`**`(`**`InvalidationListener`**` listener)`	Removes the given listener from the list of listeners, that are notified whenever the value of the `Observable` becomes invalid.

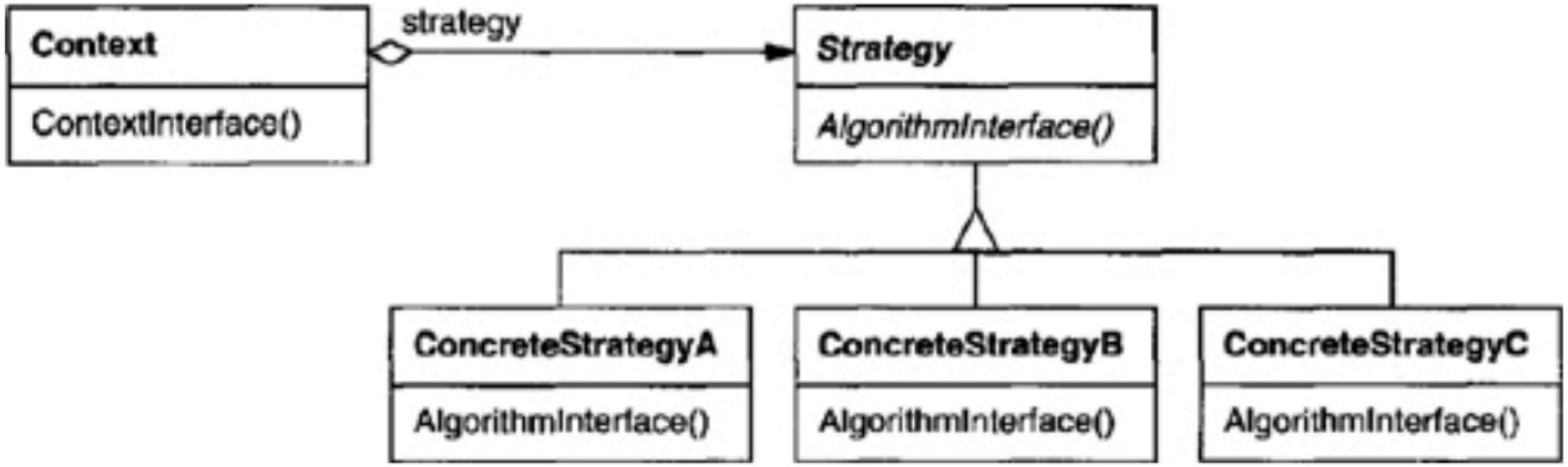
- ObservableList Interface is the subInterface of Observable Interface

STRATEGY PATTERN

- Use the Strategy pattern when
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

STRATEGY PATTERN - STRUCTURE

- The strategy pattern suggests that you implement these algorithms in separate classes. When you encapsulate an algorithm in a separate class, you call it a strategy. An object that uses the strategy object is often referred to as a context object. These "algorithms" are also called behaviors in some applications.

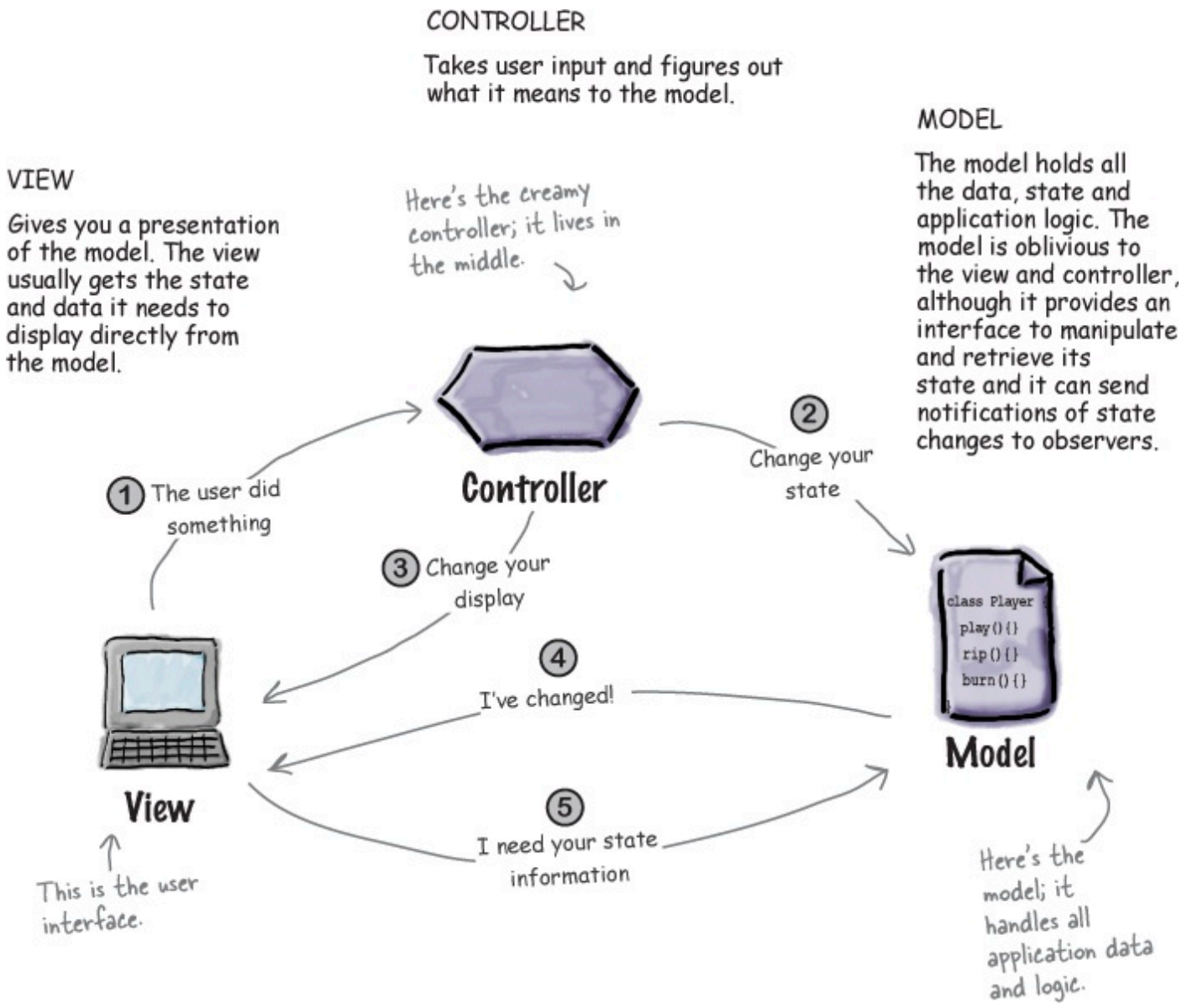


STRATEGY PATTERN – EXAMPLE

- Suppose that you have a list of integers and you want to sort them. You do this by using various algorithms; for example, Bubble Sort, Merge Sort, Quick Sort, Insertion Sort, and so forth. So, you can have a sorting algorithm with many different variations. Now you can implement each of these variations (algorithms) in separate classes and pass the objects of these classes in client code to sort your integer list.

MODEL VIEW CONTROLLER (MVC)

- A set of patterns working together
  - MVC uses Observer, Composite, and Strategy design patterns

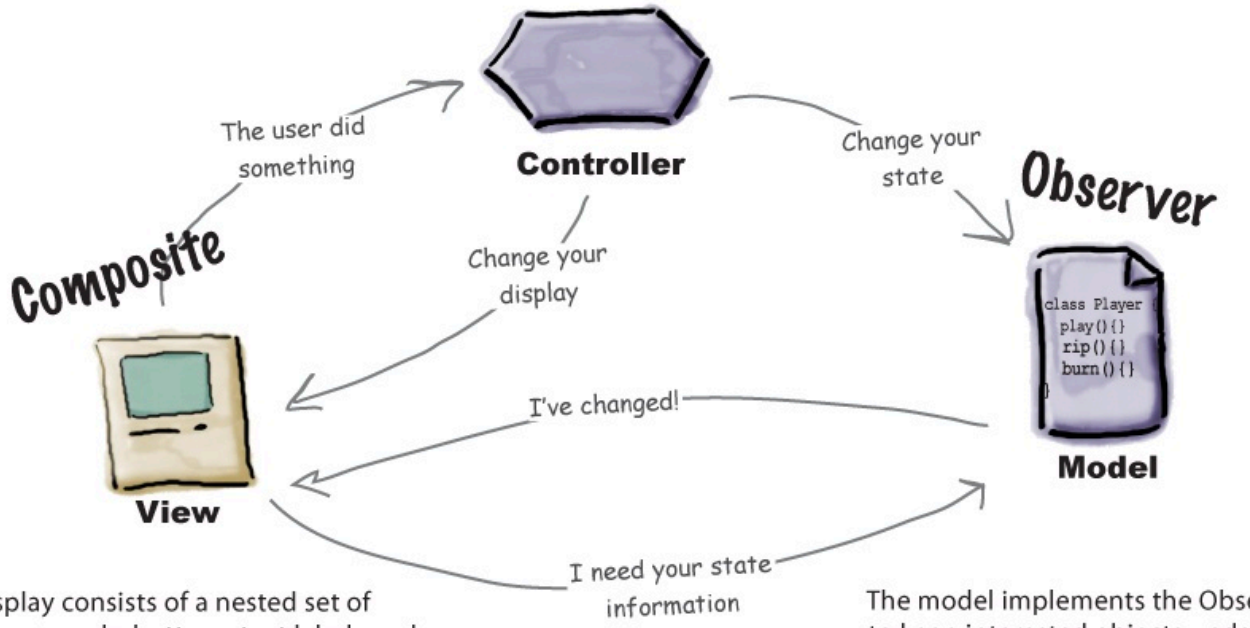


MVC PATTERN

- Observer, Composite, and Strategy design patterns work together

Strategy

The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

THANK YOU