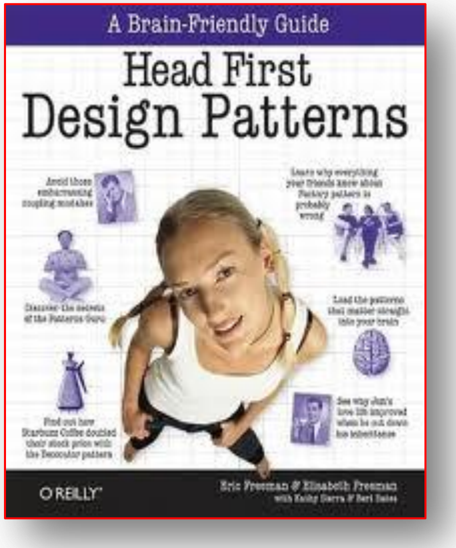
Design Patterns Tutorials

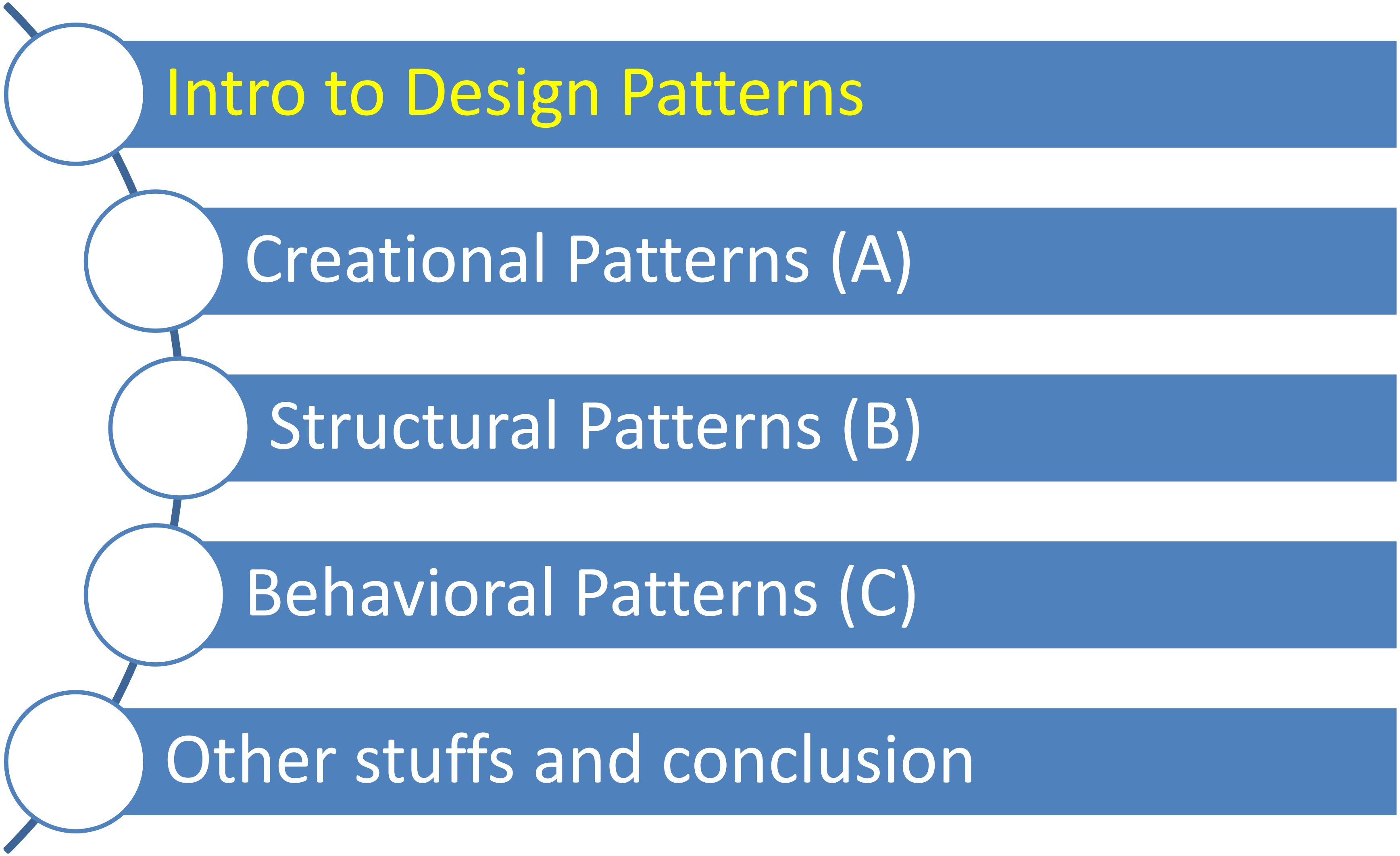

A Brain-Friendly Guide
Head First
Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

Editor: Nguyễn Đức Minh Khôi

@Feb 2012, HCMC University of Technology

Email: nguyenducminhkhoi@gmail.com

Content



- Intro to Design Patterns
- Creational Patterns (A)
- Structural Patterns (B)
- Behavioral Patterns (C)
- Other stuffs and conclusion

Intro to Design Patterns

- Object Oriented (OO) basics
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Intro to Design Patterns

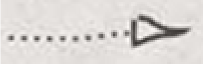
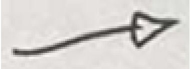
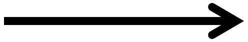
- OO Principles
  - Encapsulate what varies
  - Favor composition over inheritance
  - Program to interfaces, not implementations
  - Strive for loosely coupled design between objects and interact
  - Only talk to your friends

Intro to Design Patterns

- OO Principles (cont.)
  - Classes should be open for extension but closed for modification
  - Depend on abstraction. Do not depend on concrete classes
  - Don’t call us, we’ll call you
  - A class should have one reason to change

Intro to Design Patterns

- OO rules:
  - A class A extends (inherits) a class B

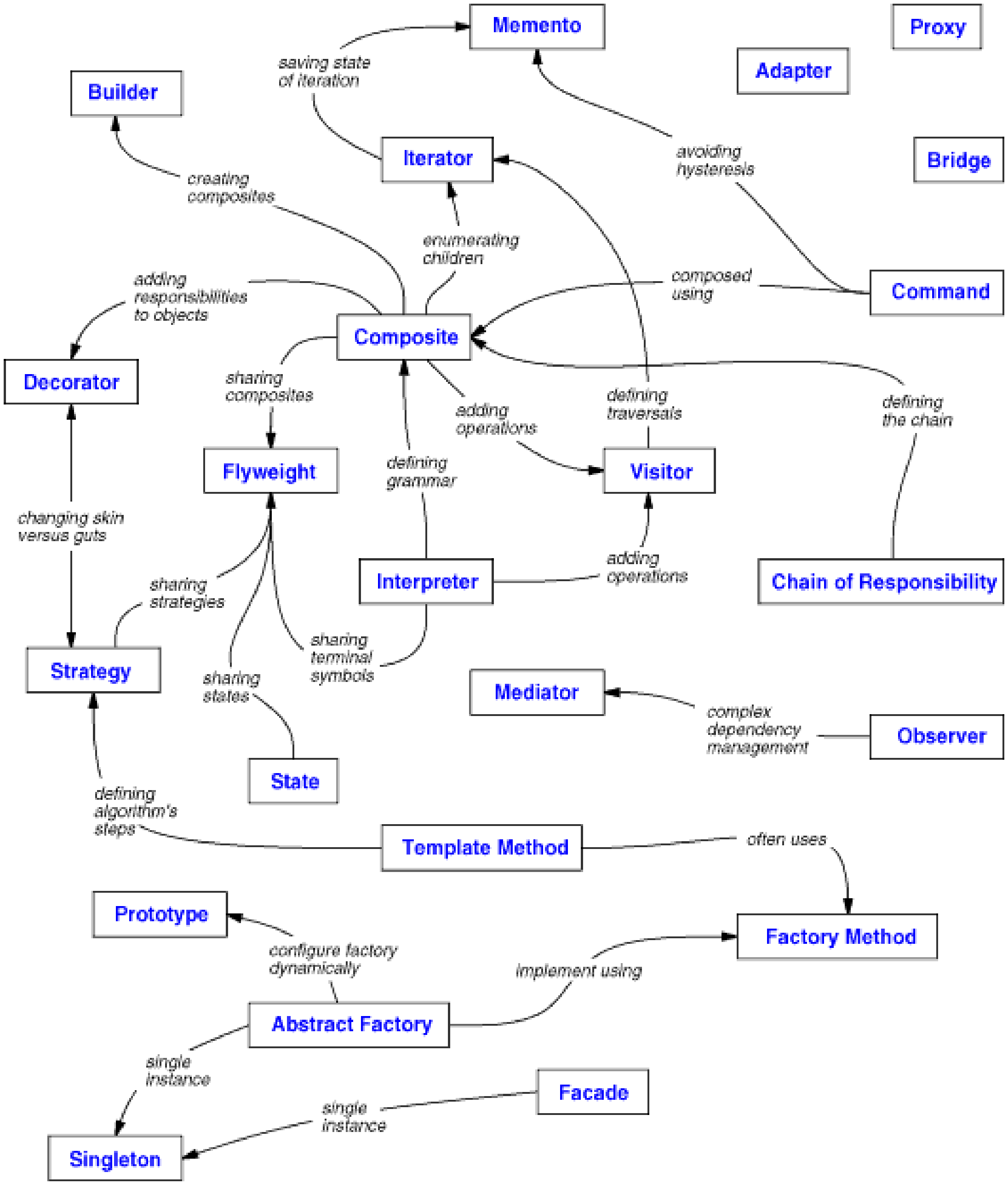
    (A: subclass, B: superclass)
  - An interface extends an interface
  - A class implements an interface 
  - Inheritance -> IS relationship 
  - Composition -> HAS relationship 

Intro to Design Patterns

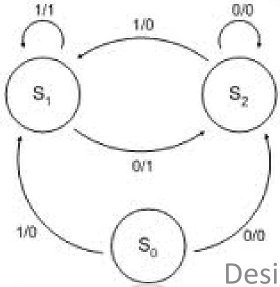
- What is Pattern?
  - **Pattern** is a **solution** to a **problem** in a **context**
  - **Context:** the situation in which the pattern apply
  - **Problem:** goal you are trying to achieve in this context (any constraints that occur in that context)
  - **Solution**: what you are after – general design that anyone can apply which resolves the goal and set of constraints

Intro to Design Patterns

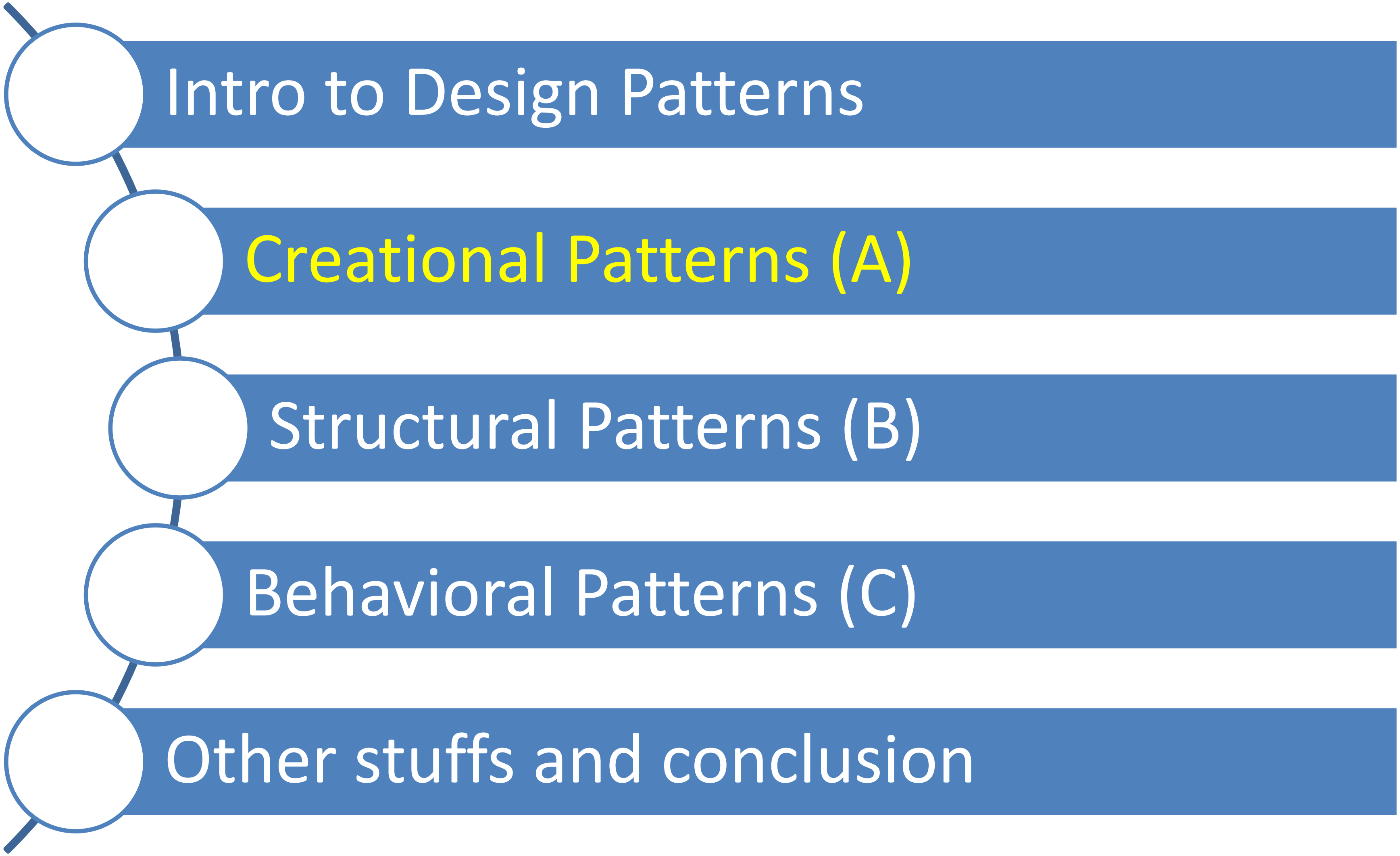
List of Design Patterns and their relationship



Overview of Patterns



Content



- Intro to Design Patterns
- Creational Patterns (A)
- Structural Patterns (B)
- Behavioral Patterns (C)
- Other stuffs and conclusion

Creational Patterns

Singleton

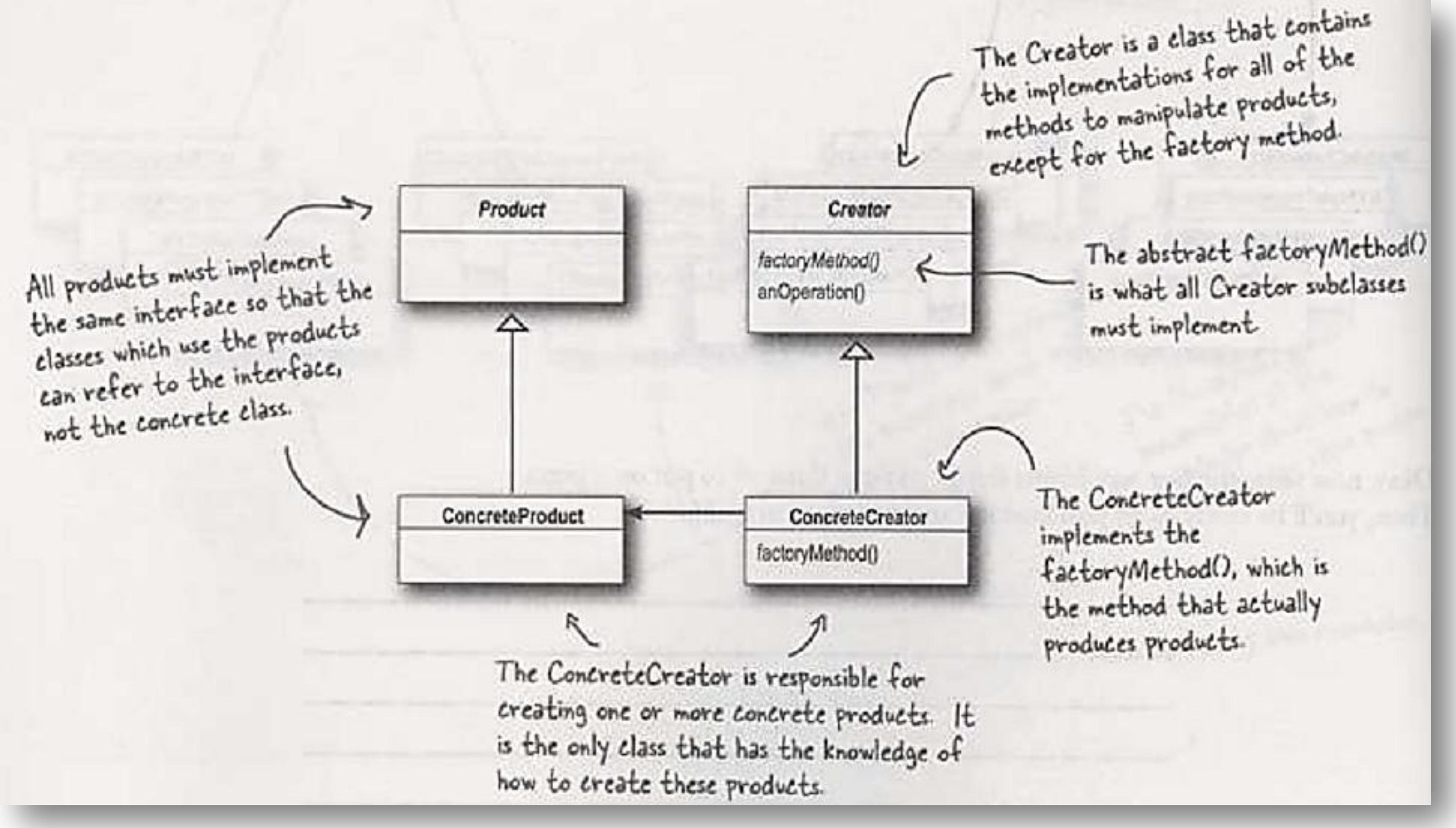
Abstract Factory

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate

Factory Method Pattern

- **Intent:** defines an interface for creating an object, but let subclasses decide which class to **instantiate**. Factory Method let a class **defer** instantiation to **subclasses**.
- Use the Factory Method pattern when:
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method Pattern (cont.)



The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.

All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.

The abstract factoryMethod() is what all Creator subclasses must implement.

The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.

The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

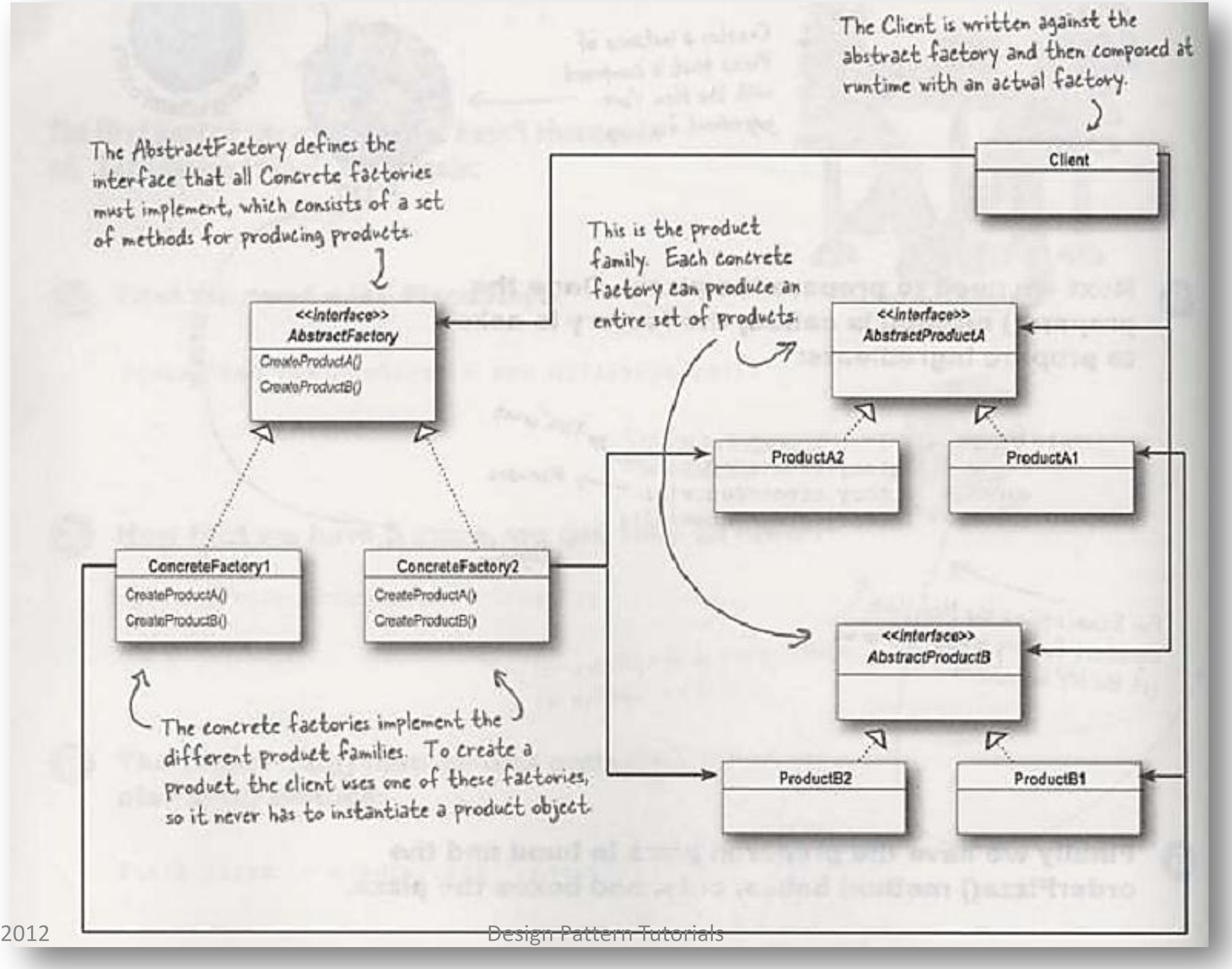
Factory Method Pattern (cont.)

- Notes:
  - Also known as **Virtual Constructor**
  - Encapsulate objects creation
  - Rely on inheritance, object creation is delegated to subclasses with implement the factory method to create objects.
  - Promote loose coupling by reducing the dependency of your application to concrete classes
  - Are powerful technique for coding abstraction, not concrete classes

Abstract Factory Pattern

- **Intent**: Provides an **interface** for creating **families** of related or dependent objects without specifying their concrete classes
- Use the Abstract Factory pattern when:
  - a system should be independent of how its products are created, composed, and represented.
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory Pattern (cont.)



The Client is written against the abstract factory and then composed at runtime with an actual factory.

The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.

This is the product family. Each concrete factory can produce an entire set of products.

The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

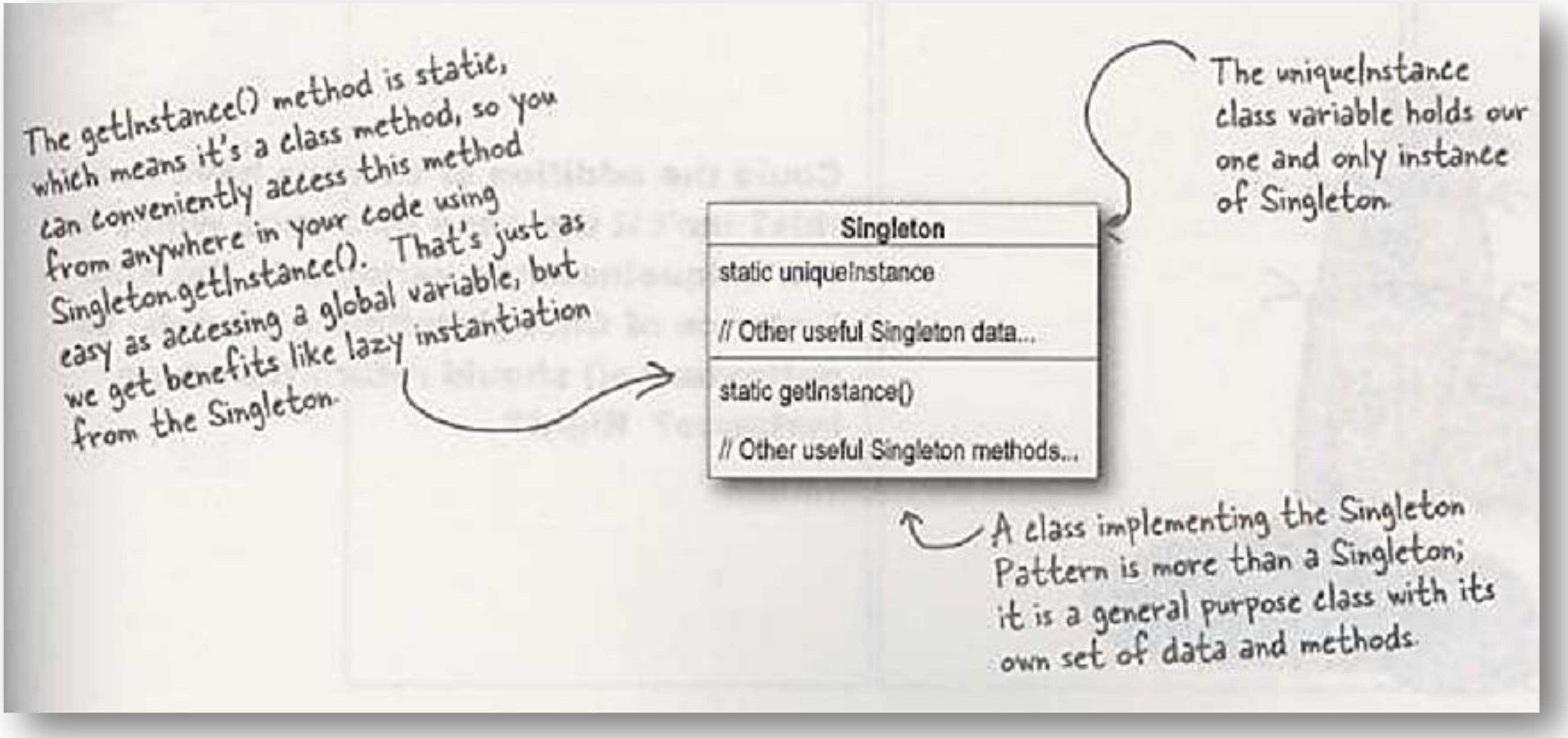
Abstract Factory Pattern (cont.)

- Notes:
  - Also known as KIT
  - Rely on composition, object creation is implemented in methods exposed in the factory interface
  - Others like Factory Method!

Singleton Pattern

- **Intents:** Ensure a class only has **one instance,** and provide a global point of access to it.
- 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.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
  - Be careful with multithread!

Singleton Pattern (cont.)

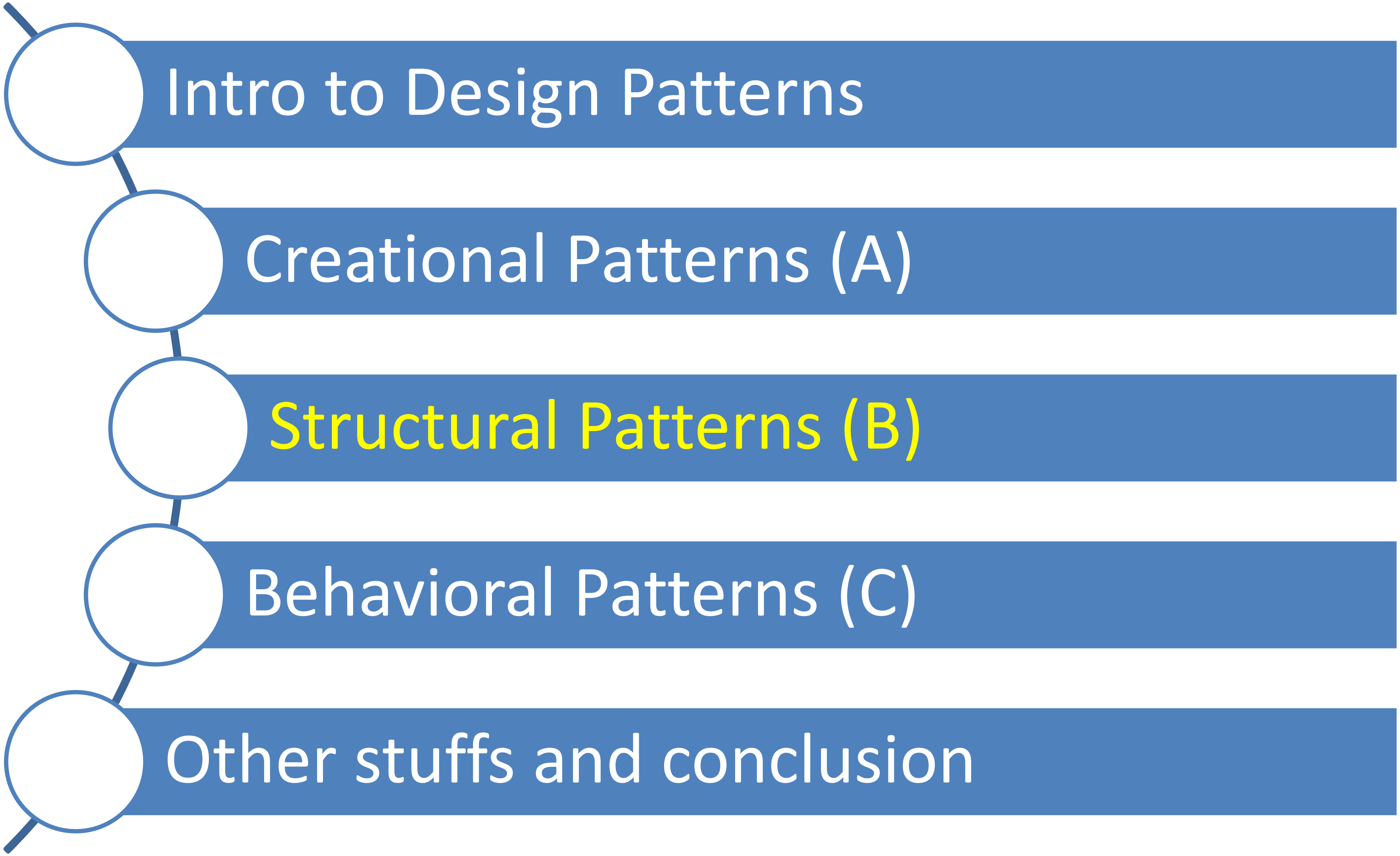

Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Content



Intro to Design Patterns

Creational Patterns (A)

Structural Patterns (B)

Behavioral Patterns (C)

Other stuffs and conclusion

Structural Patterns

Decorator

Adapter

Façade

Composite

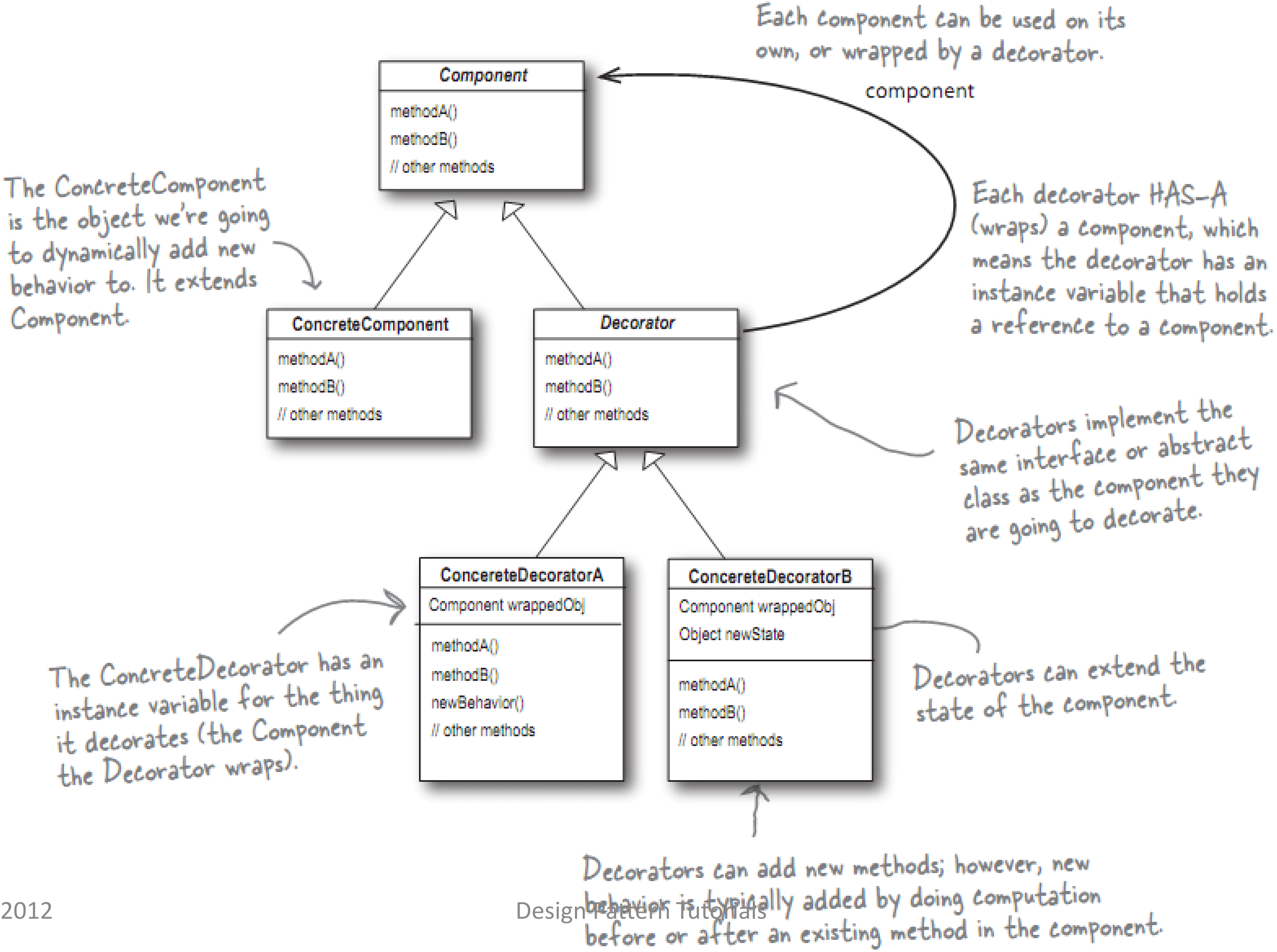
Proxy

Structural Patterns let you compose classes or objects into larger structures

Decorator Pattern

- **Intent:** Attach **additional** responsibilities to an object dynamically. Decorators provide a **flexible** alternative to subclassing for extending **functionality.**
- Use Decorator:
  - to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Decorator Pattern (cont.)



Each component can be used on its own, or wrapped by a decorator.

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

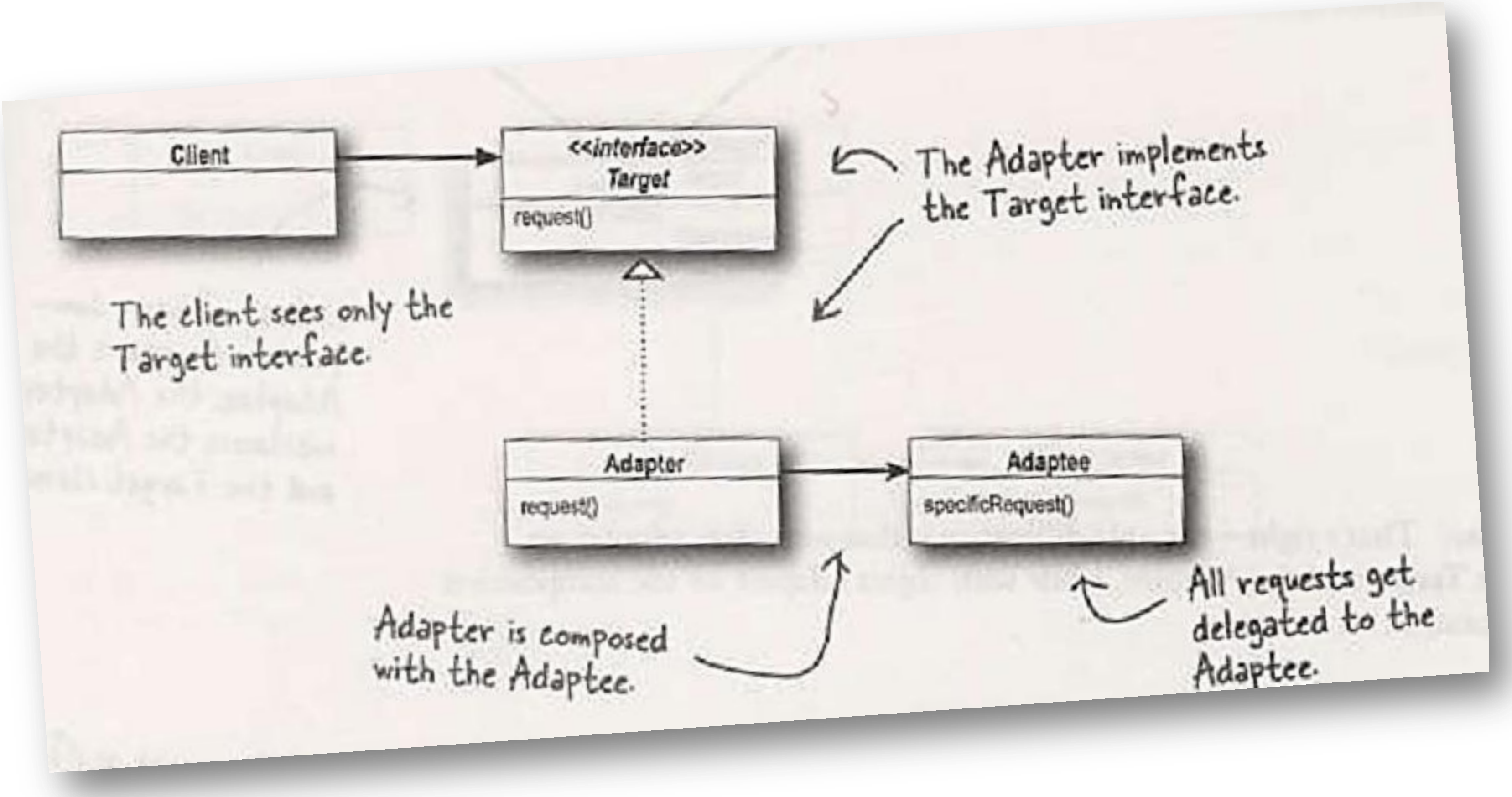
Decorator Pattern (cont.)

- Notes:
  - Also known as **Wrapper**
  - Decorator classes are the same type as the components they decorate, either through inheritance or interface implementation.
  - You can wrap a component with any number of decorators.
  - Decorators can result in many small objects in our design, and overuse can be complex.
  - Decorators are typically transparent to the client of the component

Adapter Pattern

- **Intent: convert** the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of **incompatible** interfaces.
- Use the Adapter pattern when:
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter Pattern (cont.)


Client
<<interface>>
Target
request()
Adapter
request()
Adaptee
specificRequest()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

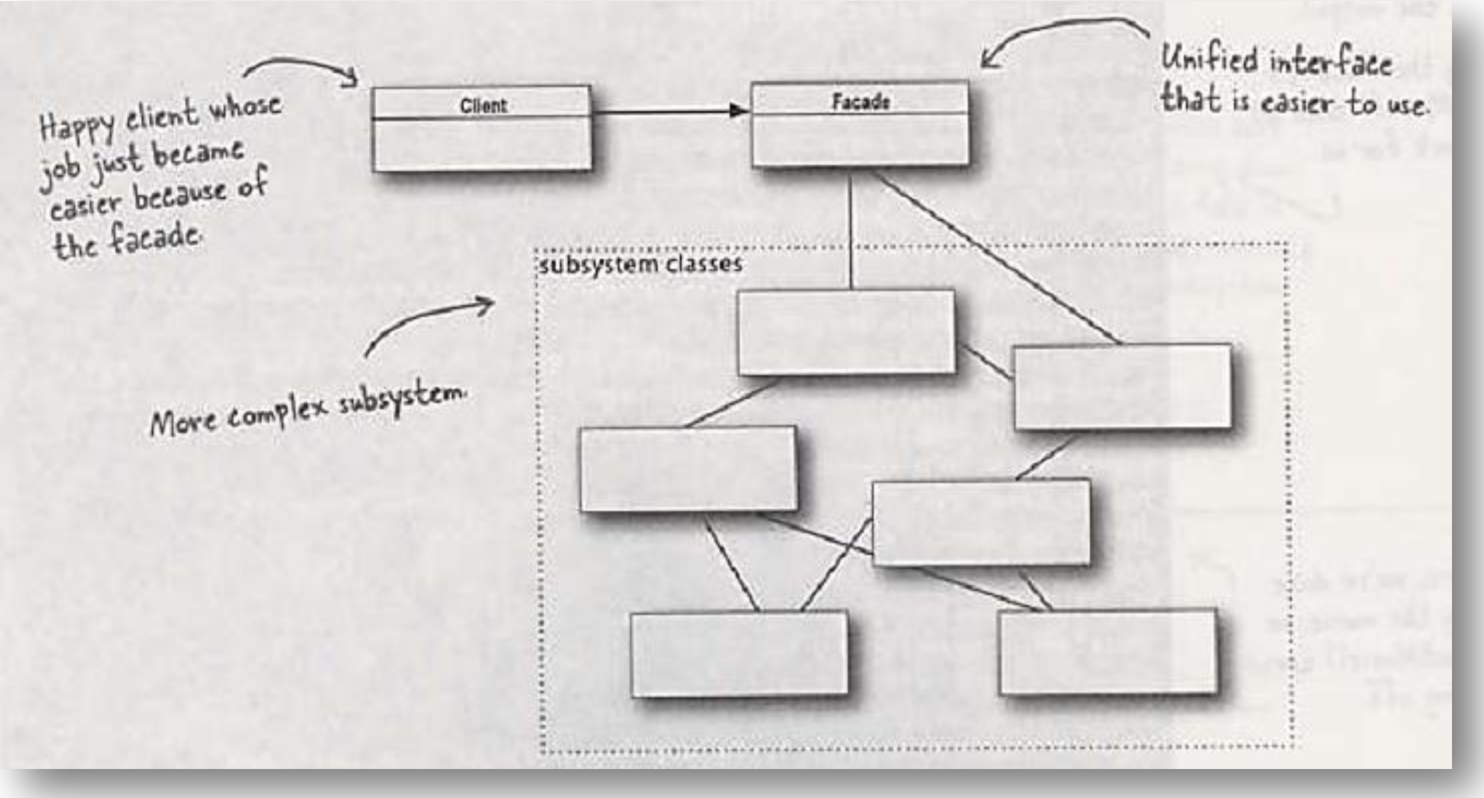
Adapter Pattern (cont.)

- Notes:
  - Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface
  - Adapter Pattern has 2 forms: object and class adapters. Class adapters require multiple inheritance
  - **Adapter** wraps an object to change its interface, a **decorator** wraps an object to add new behaviors and responsibilities

Facade 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.
- Use the Facade pattern when:
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes of an abstraction. you want to layer your subsystems.
  - Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Facade Pattern (cont.)


Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

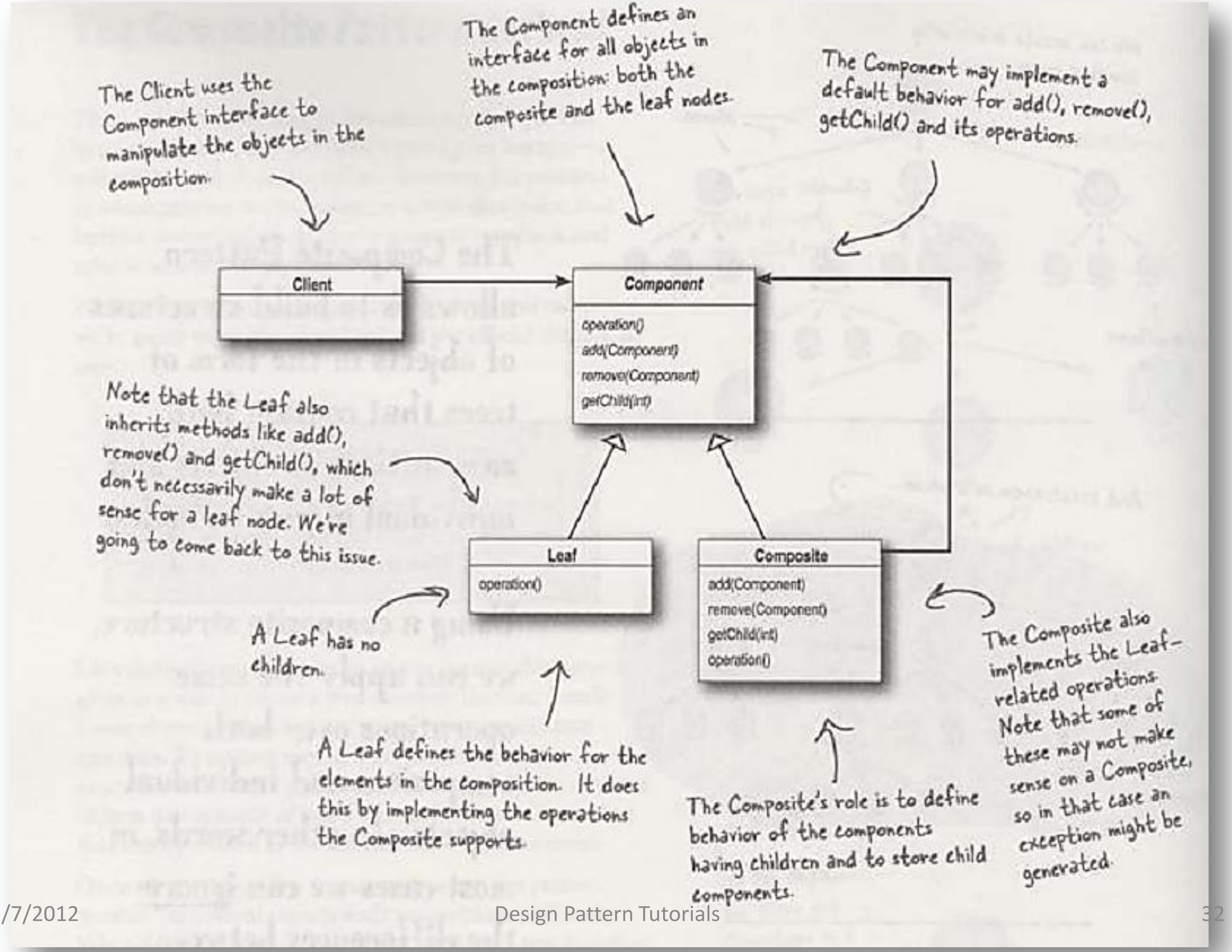
Facade Pattern (cont.)

- Notes:
  - A façade decouple a client from complex system
  - Implementing a façade requires that we compose the façade with its subsystem and use delegation to perform the work of the façade.
  - You can implement more than one façade for a subsystem.
  - A façade wraps a set of objects to simplify

Composite Pattern

- **Intent**: Compose objects into **tree structures** to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects **uniformly.**
- 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.

Composite Pattern (cont.)



The Client uses the Component interface to manipulate the objects in the composition.

The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.

The Component may implement a default behavior for add(), remove(), getChild() and its operations.

Note that the Leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.

A Leaf has no children.

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.

The Composite's role is to define behavior of the components having children and to store child components.

The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

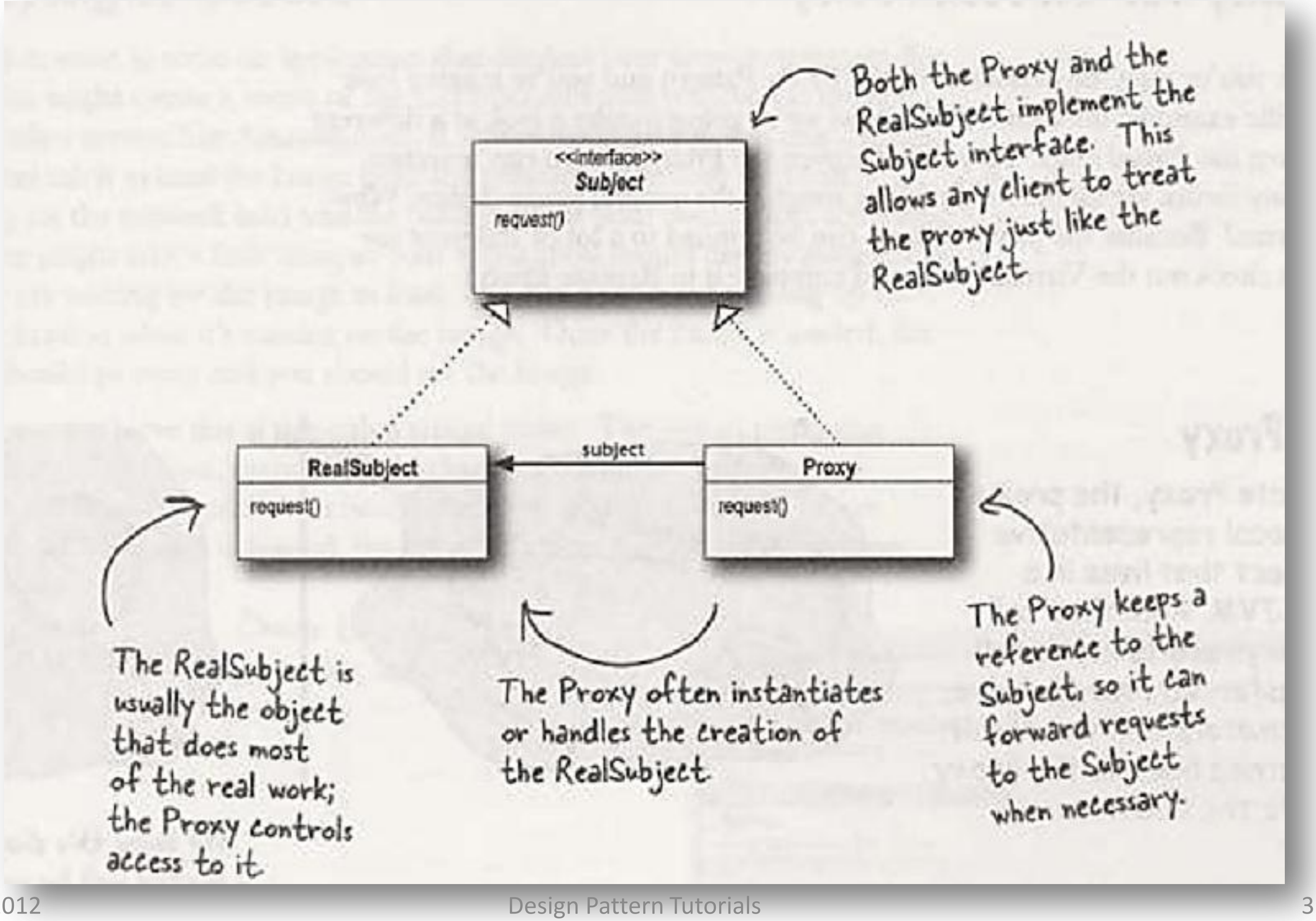
Composite Pattern (cont.)

- Notes:
  - A composition provides a structure to hold both individual (leaf nodes) objects and composites
  - There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs

Proxy Pattern

- **Intent:** Provide a surrogate or **placeholder** for another object to **control access** to it.
- 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.

Proxy Pattern (cont.)



Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.

The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.

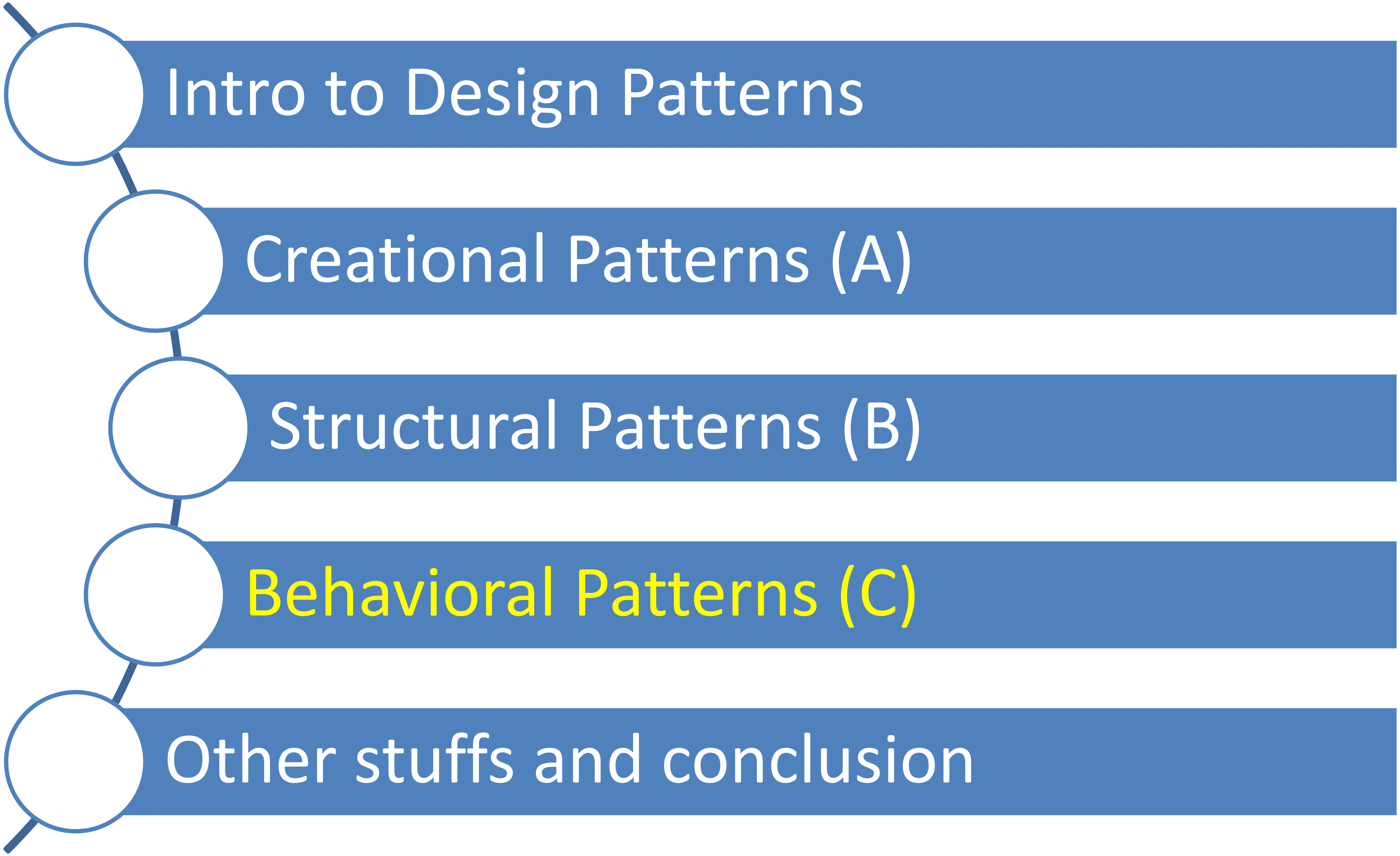
The Proxy often instantiates or handles the creation of the RealSubject.

The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

Proxy Pattern (cont.)

- Notes:
  - Also known as **Surrogate**
  - The Decorator Pattern adds behavior to an object, while a Proxy controls access
  - Like any wrapper, proxies will increase the number of classes and objects in your designs
  - Types of proxy: Remote proxy, virtual proxy, protection proxy

Content



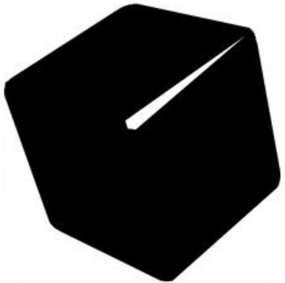
- Intro to Design Patterns
- Creational Patterns (A)
- Structural Patterns (B)
- Behavioral Patterns (C)
- Other stuffs and conclusion

Behavioral Patterns

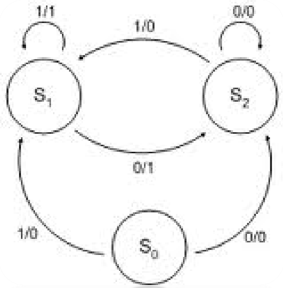
Strategy

Command

Observer



Iterator



State

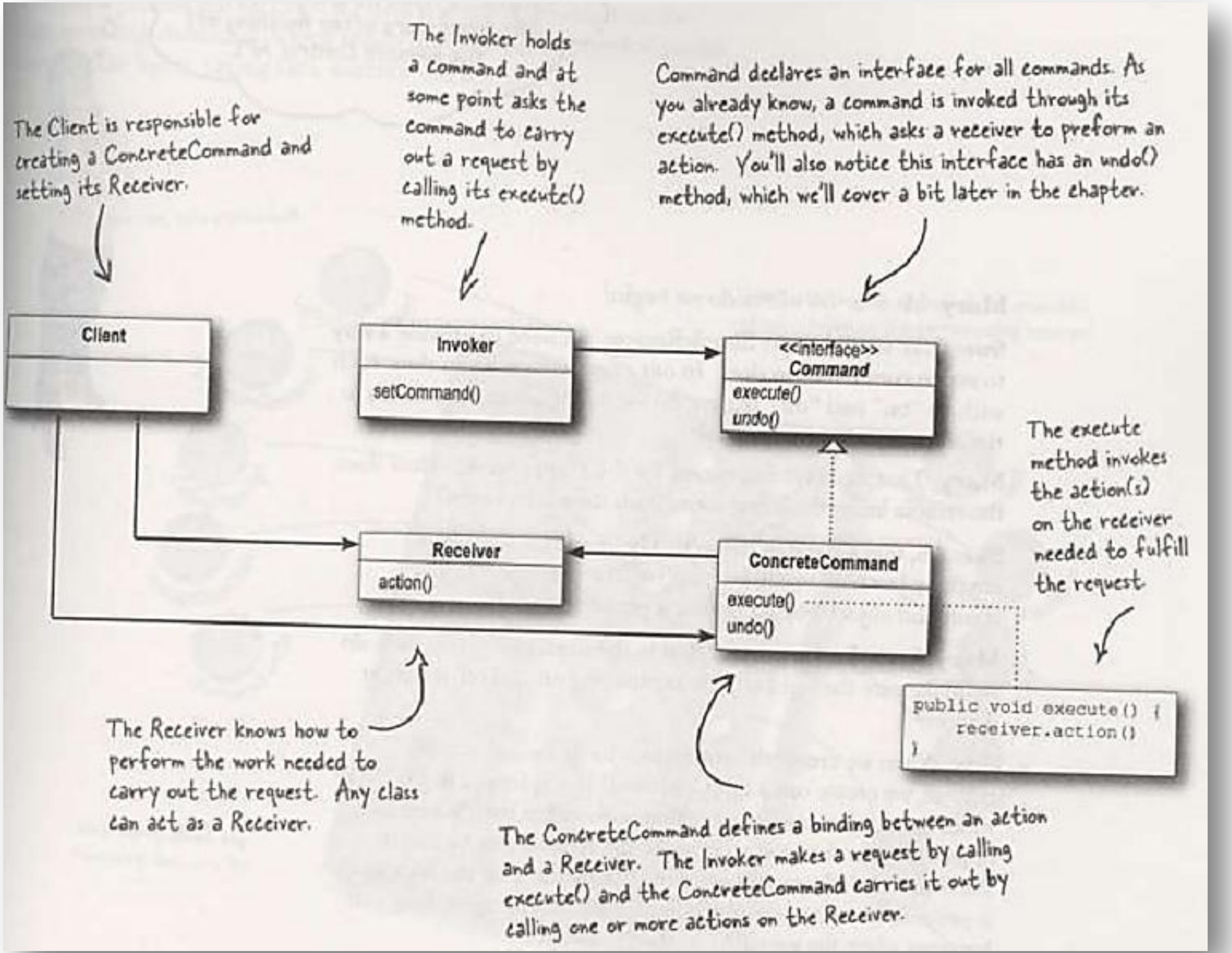
Template

Behavioral Patterns: concerned with how classes and objects interact and distribute responsibility

Command Pattern

- **Intent:** Encapsulate a request as an object, thereby letting you **parameterize** clients with different requests, queue or log requests, and support **undoable** operations.
- Use the Command pattern when you want to:
  - parameterize objects by an action to perform, as MenuItem objects did above.
  - specify, queue, and execute requests at different times.
  - support undo. The Command's Execute operation can store state for reversing its effects in the command itself.
  - support logging changes so that they can be reapplied in case of a system crash.
  - structure a system around high-level operations built on primitives operations.

Command Pattern (cont.)



The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.

The execute method invokes the action(s) on the receiver needed to fulfill the request.

```
public void execute() {
    receiver.action()
}
```

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

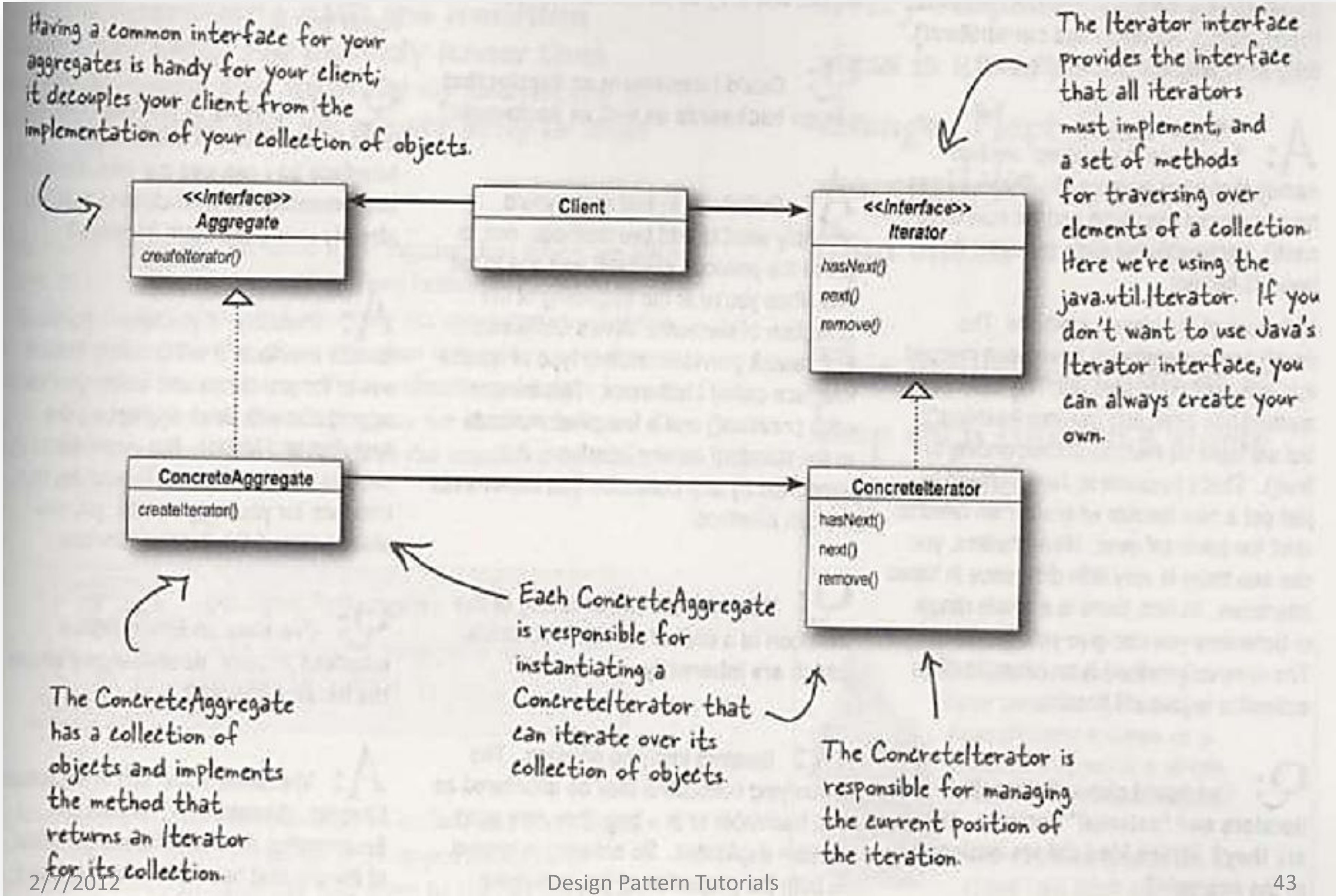
Command Pattern (cont.)

- Notes:
  - Also Known As **Action**, **Transaction**
  - In practice, it is not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receiver.
  - Commands may also be used to implement logging and transactional systems.

Iterator Pattern

- **Intent:** Provide a way to access the elements of an aggregate object **sequentially without exposing** its underlying representation.
- Use the Iterator pattern:
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Iterator Pattern (cont.)



Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.

The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.

The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteIterator is responsible for managing the current position of the iteration.

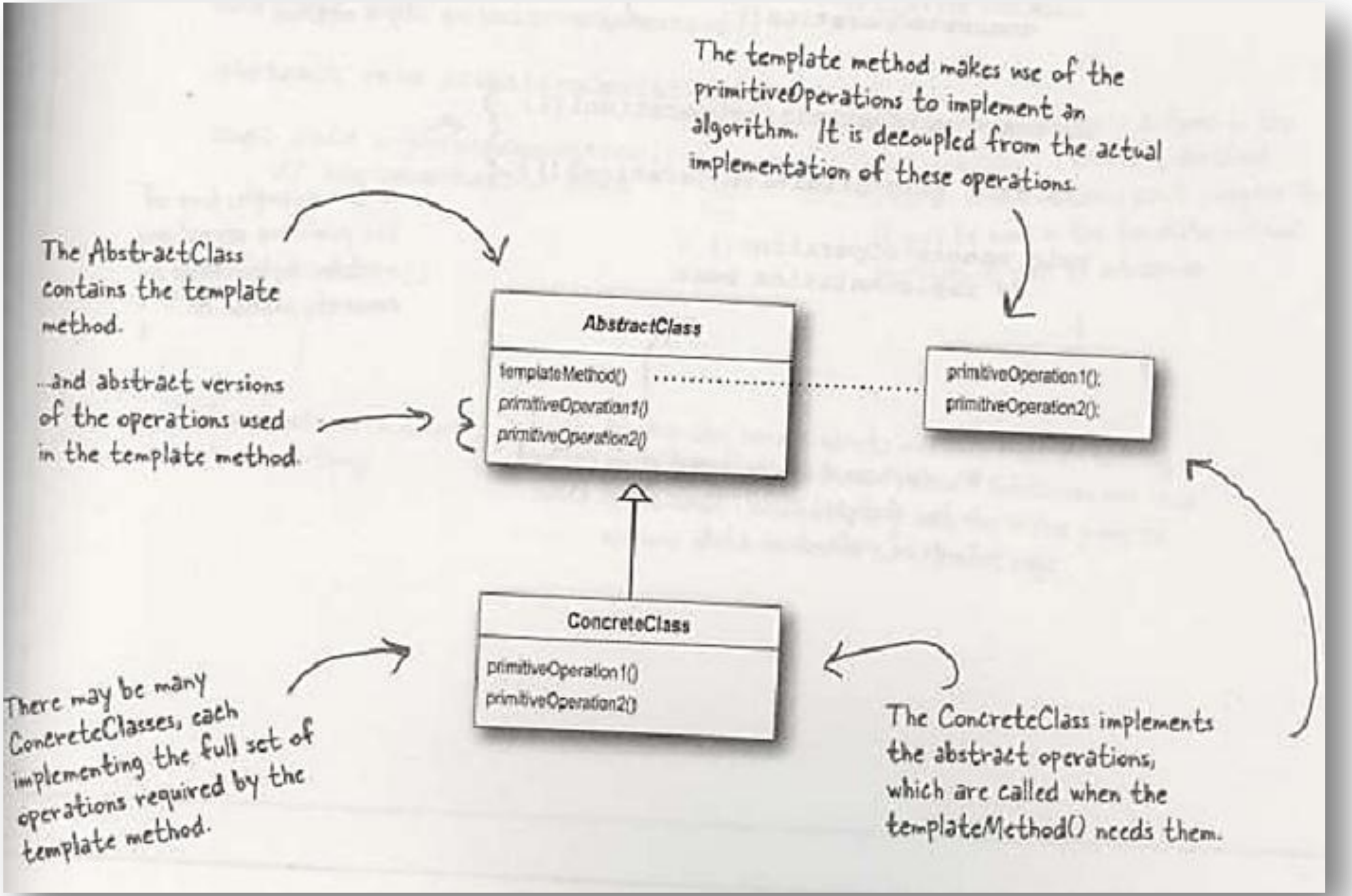
Iterator Pattern (cont.)

- Notes:
  - Also known as **Cursor**
  - When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
  - An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate

Template Pattern

- **Intent:** define the **skeleton** of an algorithm in an operation, **deferring** some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- The Template Method pattern should be used:
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
  - to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Template Pattern (cont.)



The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

The AbstractClass contains the template method.

...and abstract versions of the operations used in the template method.

There may be many ConcreteClasses, each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

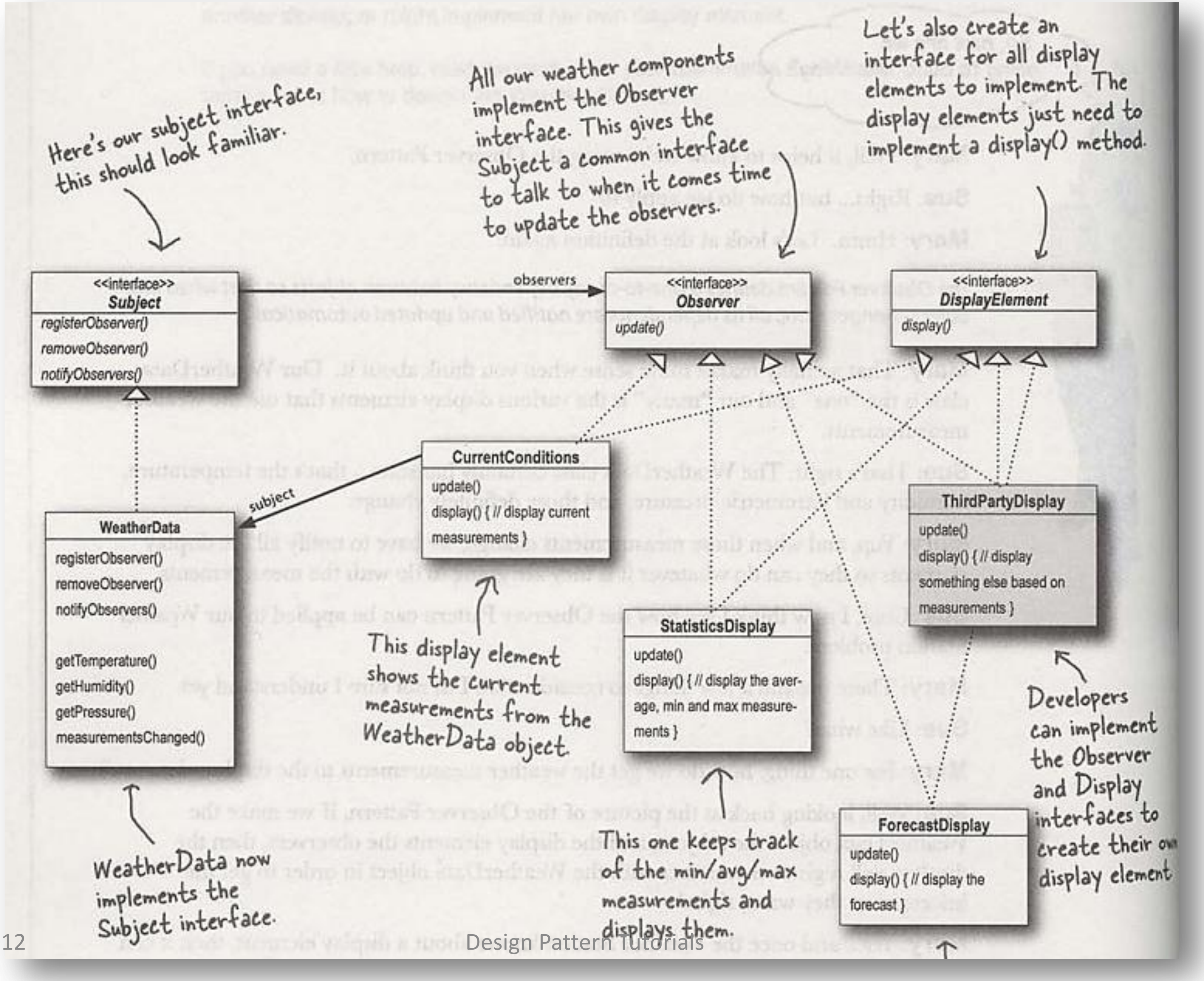
Template Pattern

- Notes:
  - Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass
  - To prevent subclasses from changing the algorithm in the template method, declare the template method as final
  - The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition
  - The Factory Method is a specialization of Template Method

Observer Pattern

- **Intent:** define a **one-to-many** dependency between objects so that when one object changes state, all its dependents are **notified and updated automatically**.
- 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 (cont.)



Here's our subject interface, this should look familiar.

All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.

This display element shows the current measurements from the WeatherData object.

Developers can implement the Observer and Display interfaces to create their own display element

WeatherData now implements the Subject interface.

This one keeps track of the min/avg/max measurements and displays them.

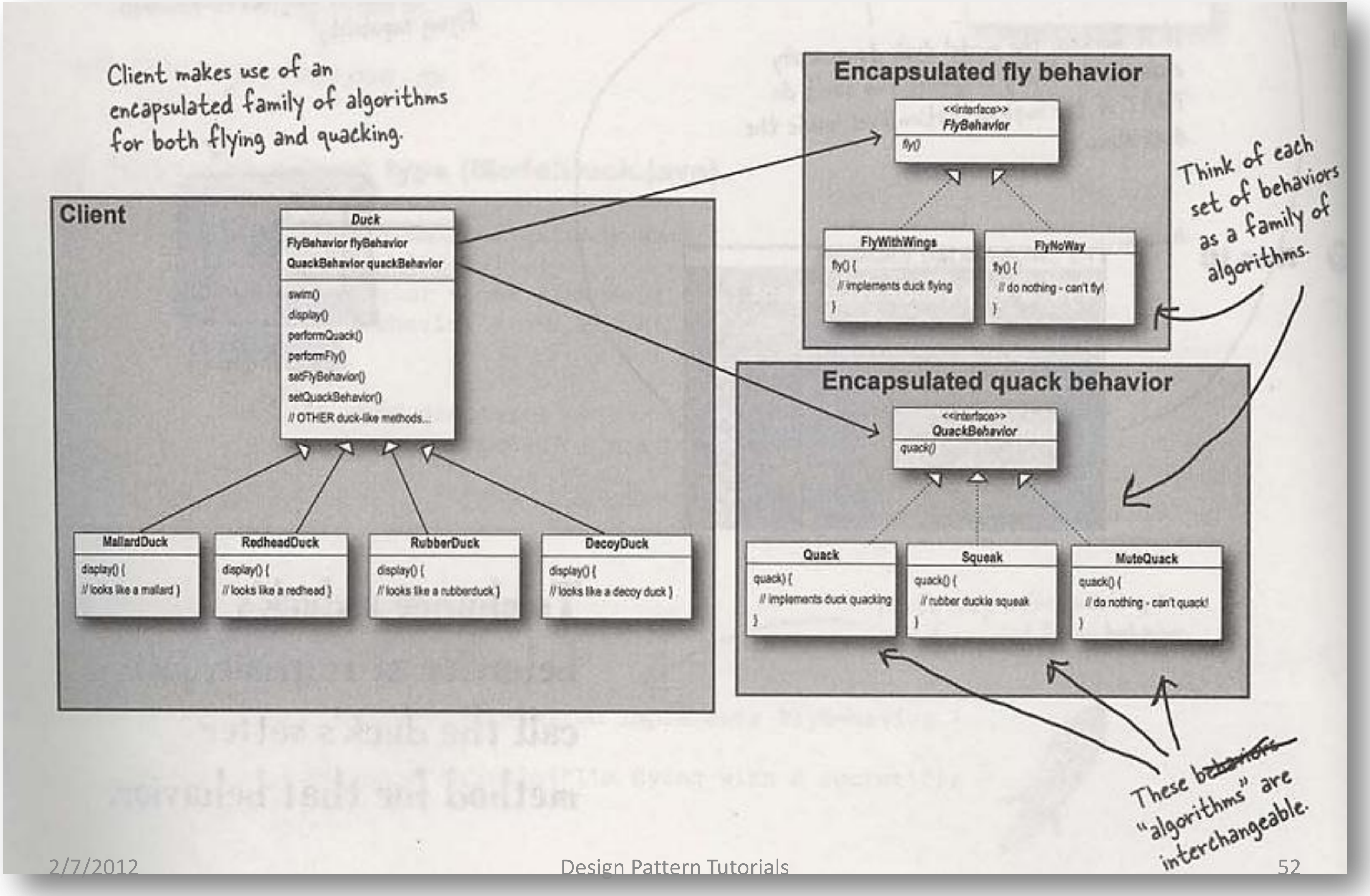
Observer Pattern (cont.)

- Notes:
  - Also known as **Dependents**, **Publish-Subscribe**
  - You can push or pull data from the Observable when using the pattern
  - Don’t depend on a specific order of notification for your Observers.
  - Java has several implementations of the Observer Pattern, including the general purpose java.util.Observable

Strategy Pattern

- **Intent:** define a **family of algorithms**, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known as **Policy**
- 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.
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy Pattern (cont.)

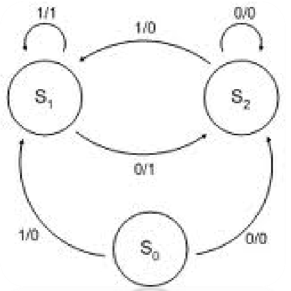


Client makes use of an encapsulated family of algorithms for both flying and quacking.

Think of each set of behaviors as a family of algorithms.

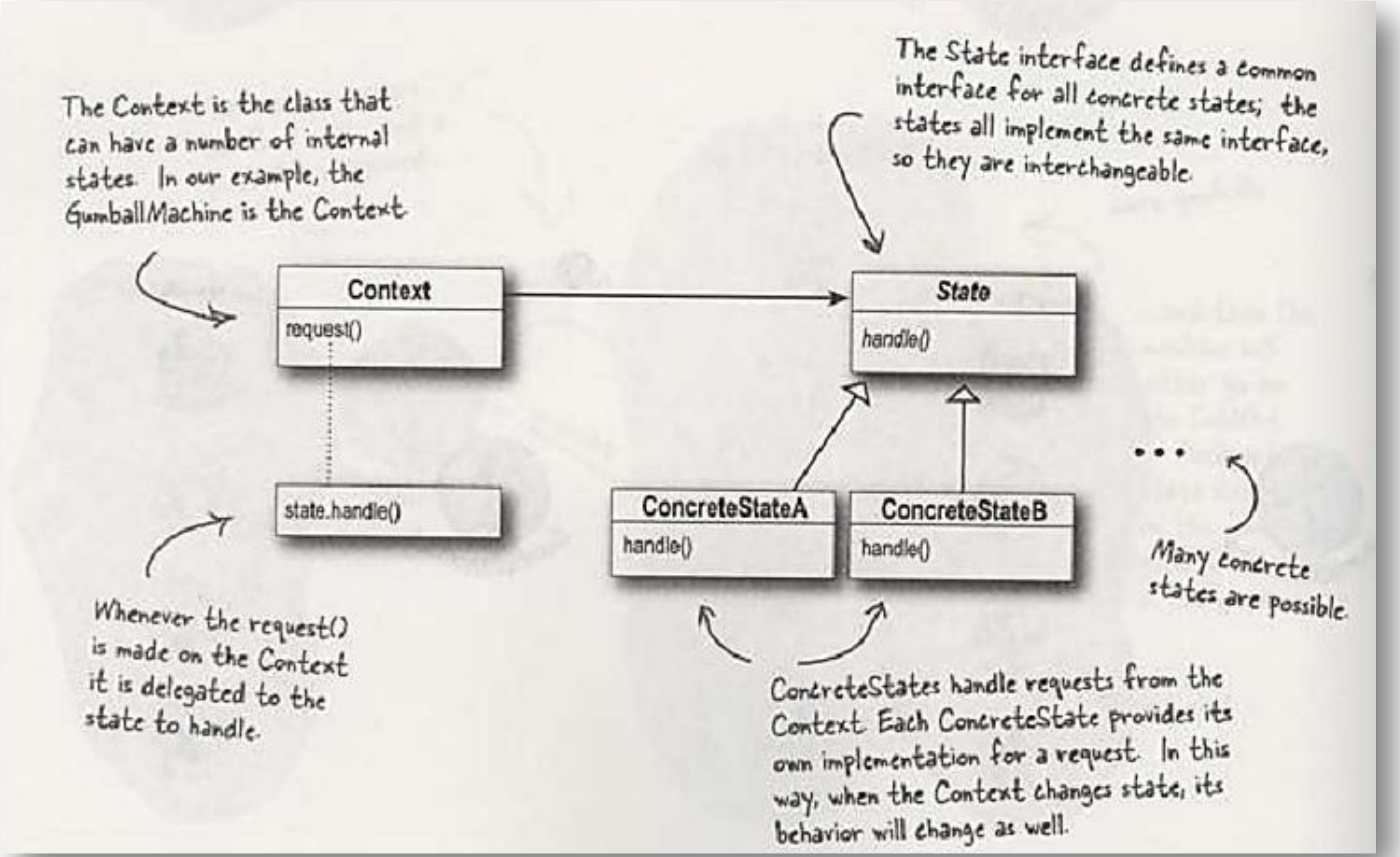
These ~~behaviors~~ "algorithms" are interchangeable.

State Pattern



- **Intent:** allow an object to **alter** its behavior when its internal state changes. The object will appear to change its class.
- Use the State pattern in either of the following cases:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

State Pattern (cont.)



The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.

The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.

Whenever the request() is made on the Context it is delegated to the state to handle.

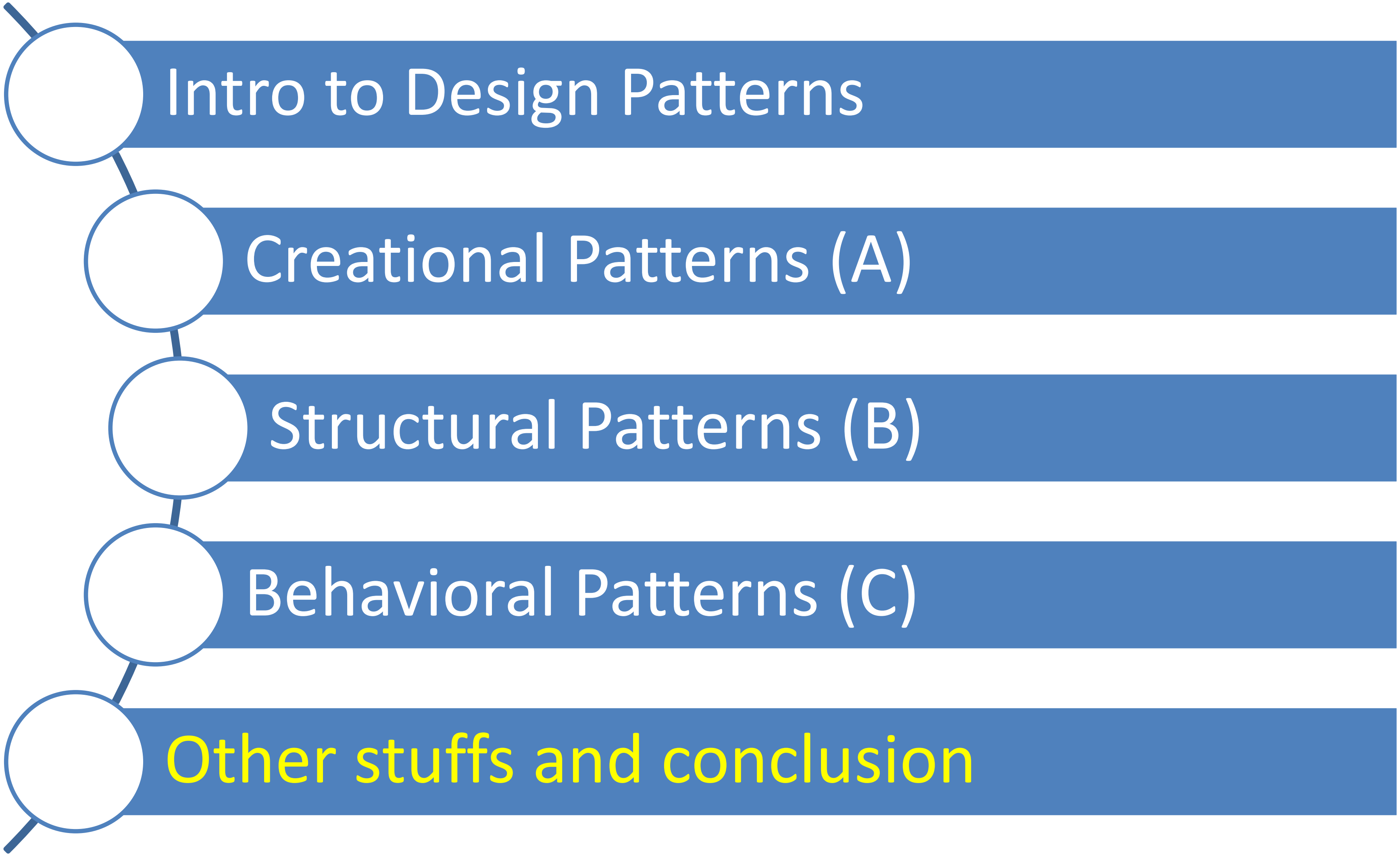
Many concrete states are possible.

ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

State Pattern

- Notes:
  - Also known as **Objects for States**
  - The Context gets its behavior by delegating to the current state object is composed with.
  - The State and Strategy Patterns have the same class diagram, but they differ in intent.
  - Using the State Pattern will typically result in a greater number of classes in your design
  - State classes may be shared among Context instances

Content



Intro to Design Patterns

Creational Patterns (A)

Structural Patterns (B)

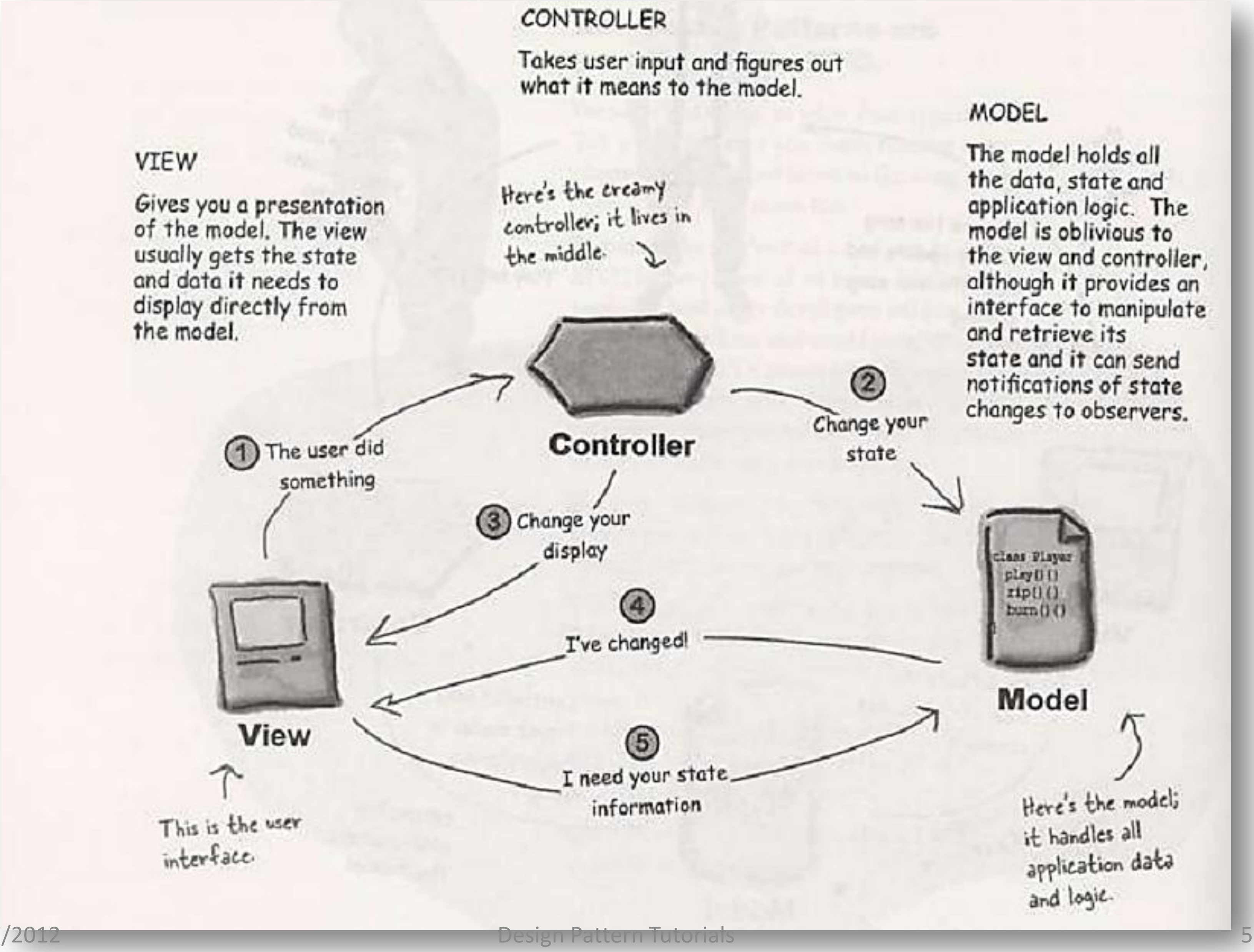
Behavioral Patterns (C)

Other stuffs and conclusion

Pattern of pattern

- Pattern are often used together and combined within the same design solution
- **Compound pattern** combines two or more patterns into a solution that solves a recurring or general problem
- The **Model View Controller Pattern** (MVC) is a compound pattern consisting of the Observer, Strategy and Composite patterns.

MVC Patterns



CONTROLLER

Takes user input and figures out what it means to the model.

MODEL

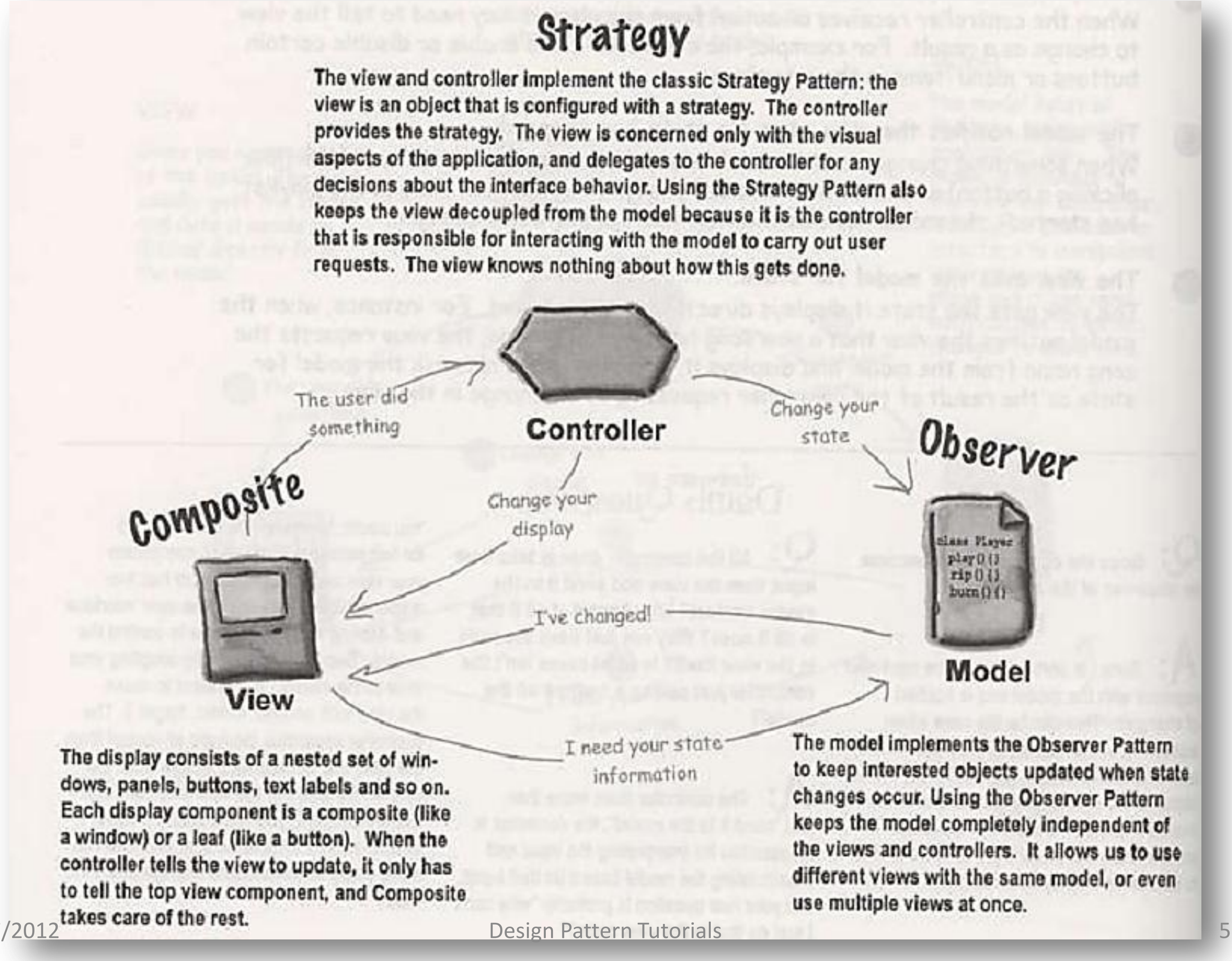
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

VIEW

Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

Here's the creamy controller; it lives in the middle.
Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
View
Model
class Player
play(){}
rip(){}
burn(){}
This is the user interface.
Here's the model; it handles all application data and logic.

MVC Patterns (cont.)



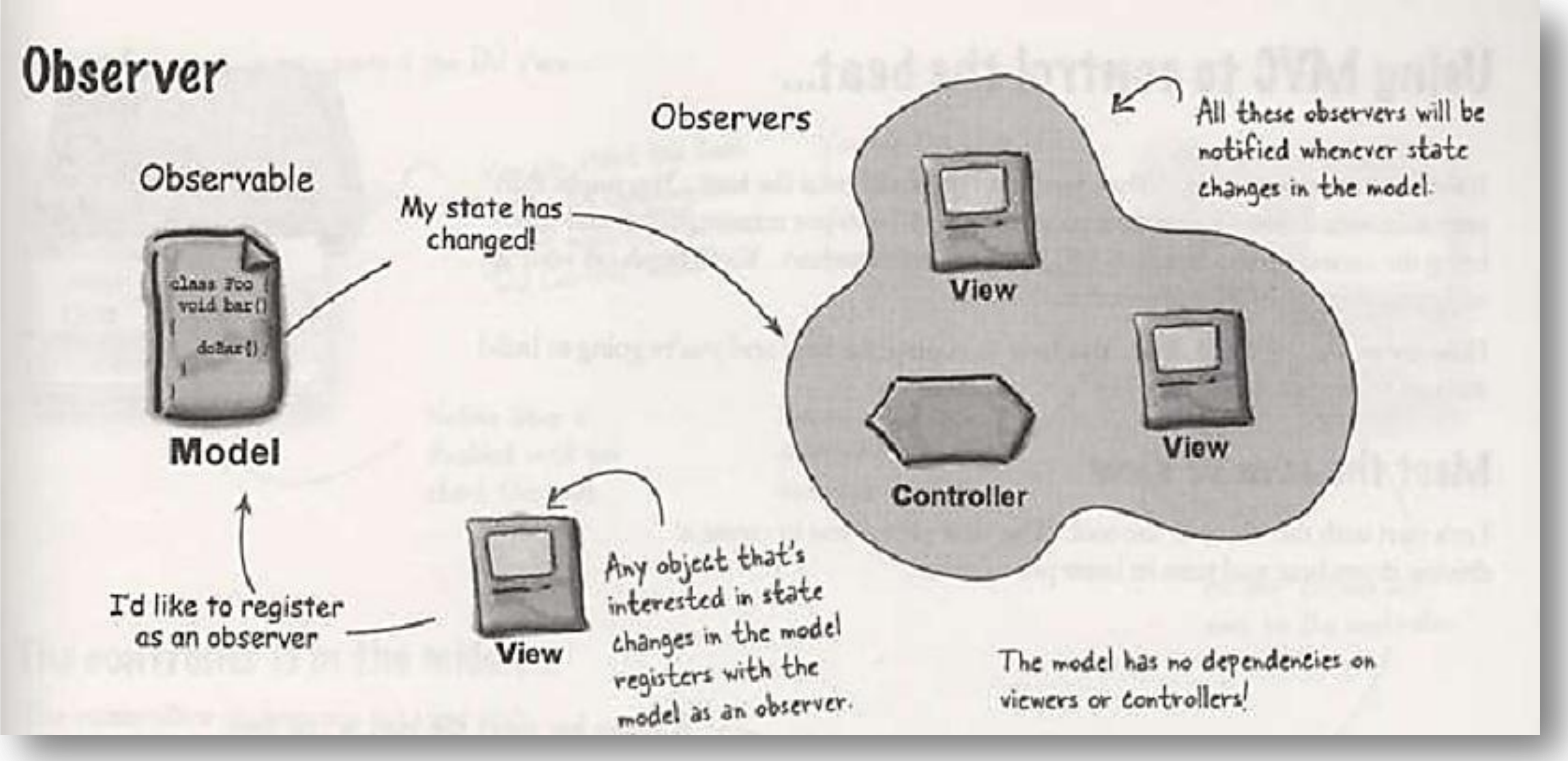
Strategy

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.

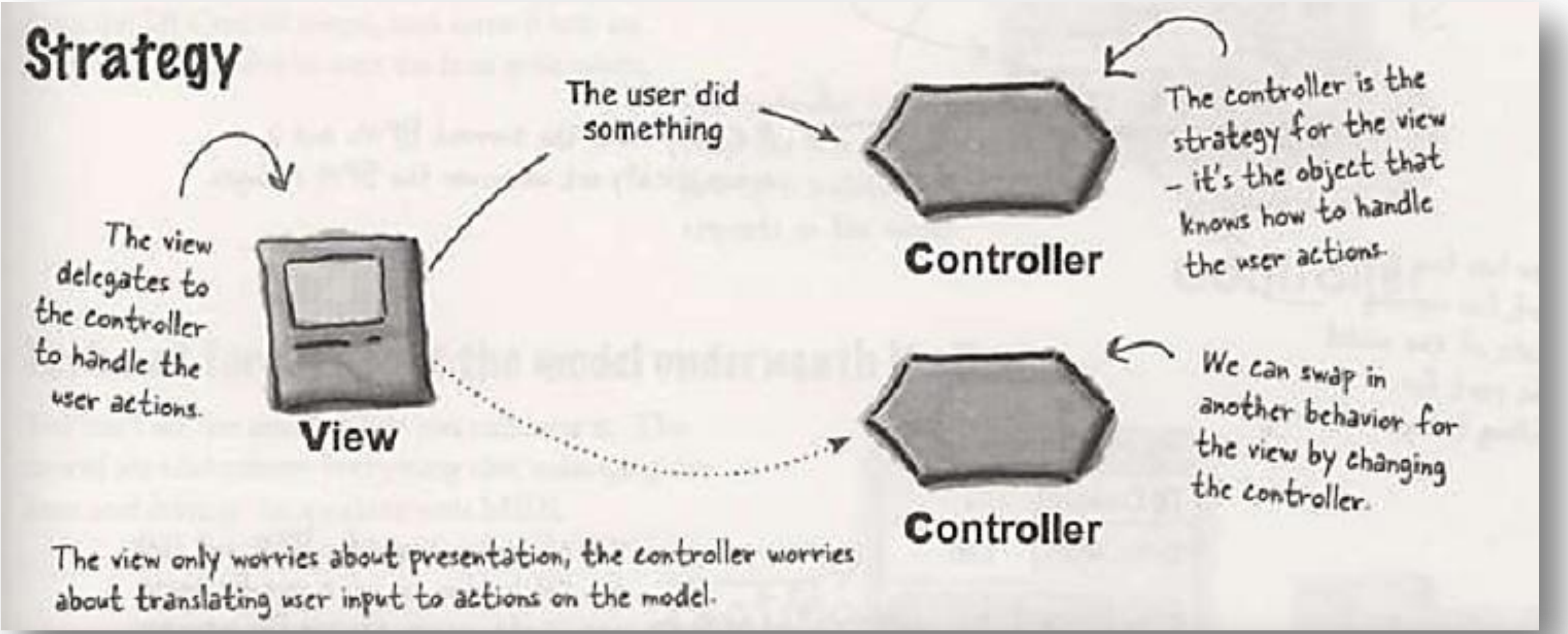
The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

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.

MVC Patterns (cont.)


Observer
Observers
Observable
My state has changed!
class Foo {
void bar()
{
doBar()
}
Model
View
View
Controller
All these observers will be notified whenever state changes in the model.
View
I'd like to register as an observer
Any object that's interested in state changes in the model registers with the model as an observer.
The model has no dependencies on viewers or controllers!

MVC Patterns (cont.)


Strategy

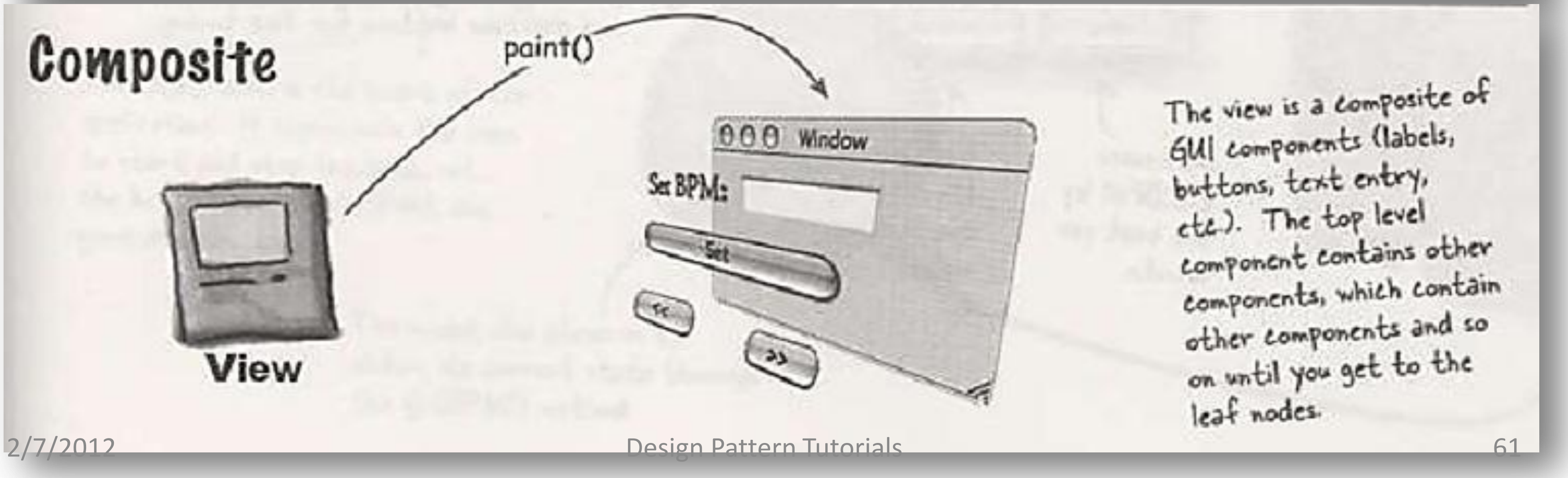
The view delegates to the controller to handle the user actions.

The user did something

The controller is the strategy for the view – it's the object that knows how to handle the user actions.

We can swap in another behavior for the view by changing the controller.

The view only worries about presentation, the controller worries about translating user input to actions on the model.


Composite

The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.

Other Design Patterns

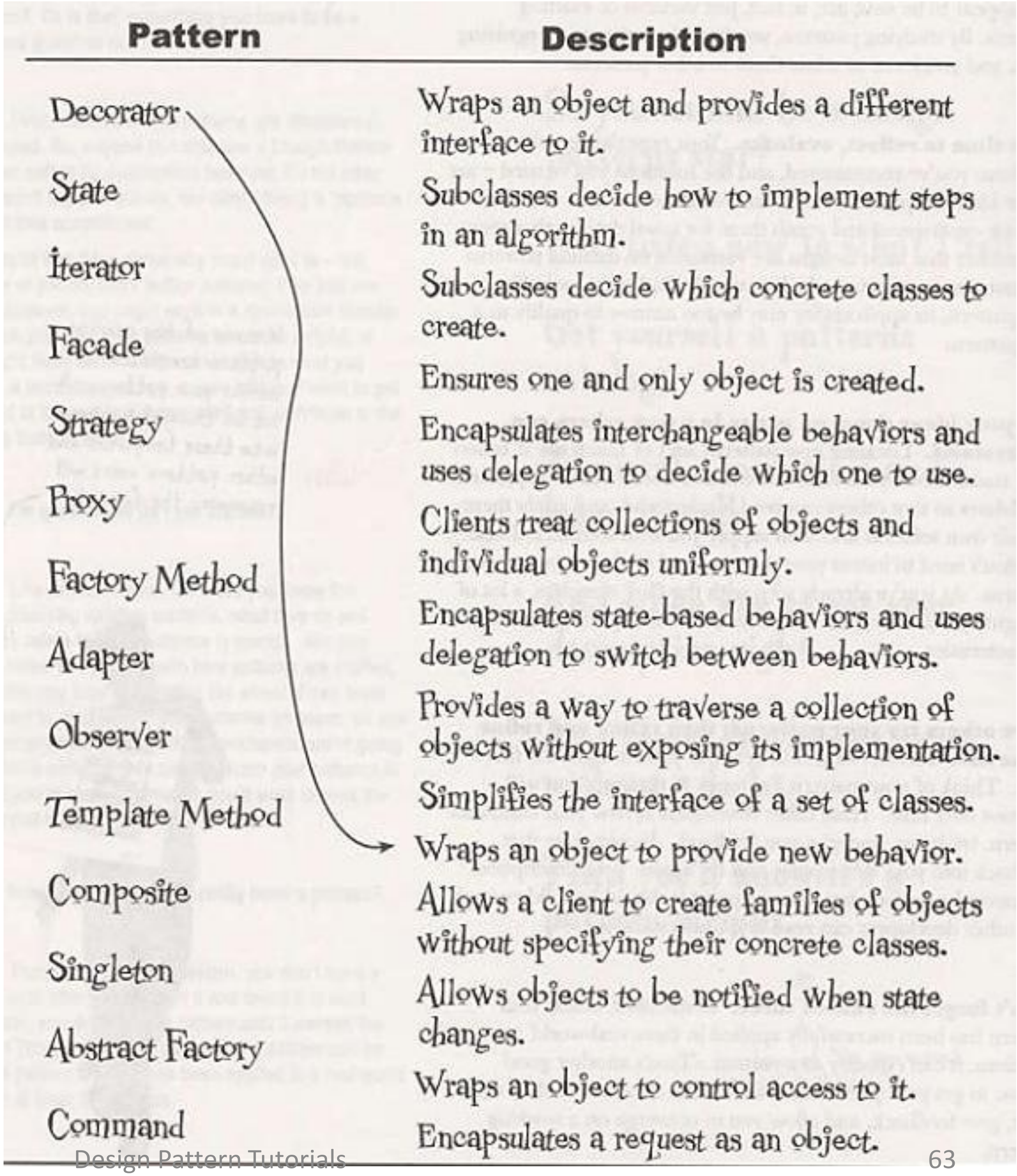
- Prototype (A)
- Builder (A)
- Bridge (B)
- Flyweight (B)

- Interpreter (C)
- Mediator (C)
- Memento (C)
- Visitor (C)
- Chain of responsibility (C)

Look up them more in the Head First Design Pattern book for more details!

Test your understanding!

Match each pattern with its description



Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

Conclusion

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	families of product objects
	Factory Method	subclass of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Proxy	how an object is accessed; its location

Conclusion (cont.)

Purpose	Design Pattern	Aspect(s) That Can Vary
Behavioral	Command	when and how a request is fulfilled
	Iterator	how an aggregate's elements are accessed, traversed
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm

Conclusion (cont.)

- Design Patterns (DP) aren’t set in stone, adapt and tweak them to meet your needs
- **Always use the simplest solution that meets your needs, even if it doesn’t include a pattern**
- Study DP catalogs to familiarize yourself with patterns and the relationship among them
- Most patterns you encounter will be adaptations of existing patterns, not new pattern

References

- **Head First Design Patterns** – by Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson - O'Reilly Media (2004)
- **Design Patterns: Elements of Reusable Object-Oriented Software** – by Gang of Four - Addison-Wesley Professional (1994)

The End of **Design Patterns Tutorials**

THANKS FOR LISTENING

Editor: Nguyễn Đức Minh Khôi

@Feb 2012, HCMC University of Technology

Email: nguyenducminhkhoi@gmail.com