


Design Patterns

A decorative horizontal bar with a wavy, ribbon-like shape spans the width of the slide. It is composed of various colored segments including black, blue, light blue, teal, and yellow.

Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.

www.fandsindia.com

fands@vsnl.com

Ground Rules

- **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- **If you have questions or issues, please let me know immediately.**
- **Let us be punctual.**

Contents

A Short History

- Christopher Alexander - 1970s
 - *A Pattern Language*
 - *A Timeless Way of Building*
- The Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides - 1995
- *Design Patterns: Elements of Reusable Object-Oriented Software*

What is Design Pattern?

- A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it.
[Buschmann et al., 1996]
- *A design pattern is a well worn and known good solution to a common problem.*

Categorization of Patterns

- **Creational Patterns:** Used to create objects instead of direct instantiation
- **Structural Patterns:** Compose objects into larger structures
- **Behavioural Patterns:** Communication between objects

Why Design Patterns ?

- Reuse solutions that have worked in the past.
- Basing new designs on prior experience
- Once you know the pattern, a lot of design decisions follow automatically
- Makes successful design solutions more accessible to developers of new systems
- Help you identify less-obvious abstractions and the objects that can capture them

What are Design Patterns ?

- Generically: 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.
- Specifically: Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

Design Patterns consist of ..

- Pattern Name and Classification
- Intent: What does the design pattern do?
- Applicability: What are the situations in which the design pattern can be applied?
- Model: A graphical representation of the classes in the pattern
- Collaborations: The classes and/or objects participating in the design pattern and their responsibilities. How the participants collaborate to carry out their responsibilities.
- Consequences: What are the trade-offs and results of using the pattern?

Design Patterns help you to avoid ...

- Explicitly naming a class when creating an object ...
- Dependence on specific hardware/ software platform ..
- Dependence on object implementations ...
- Dependence on specific algorithms ..
- Tight coupling among classes ..
- Inheritance by using composition ..
- Dependence on specific operations ..

Resources: Principles and Patterns

- Designing Object-Oriented C++ Applications using the Booch Method By Robert C. Martin (1994)
- Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch
- http://www.cetus-links.org/oo_patterns.html
- <http://hillside.net/patterns/patterns.html>
- <http://theserverside.com/home/index.jsp>
- <http://www.cs.wustl.edu/~schmidt/patterns.html>

Objects

- Defined as a concept, abstraction or thing with crisp boundaries and meaning for problem at hand
- Simply, something that makes sense in an application context.
- Objects promote understanding of the real world and provide a practical basis for computer implementation.
- All objects have identity and are distinguishable
- Identity means that objects are distinguished by their inherent existence and not by their descriptive properties

Classes

- An object class describes a group of objects with similar properties (attribute), common behaviour (operations), common relationships to other objects and common semantics.
- Person, Company, process are all classes
- Each object knows its class implicitly
- Objects and classes appears as Nouns in problem descriptions

Classes & Objects

Person

Class

Person
Monica

Person
Rahul

(Person)

Objects

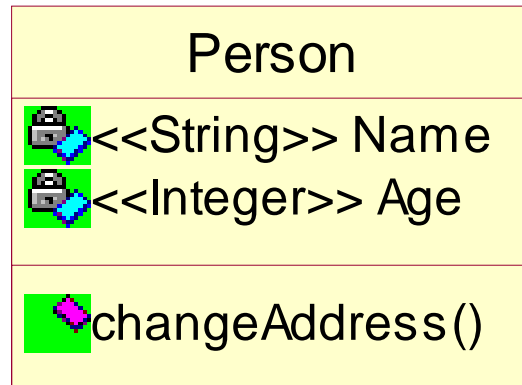
Attributes

- An attribute is a data value held by the objects in a class
- attributes encapsulate structural characteristics

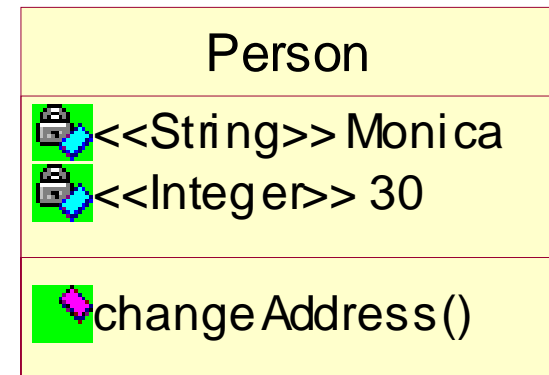
Operations

- An operation is a function or transformation that may be applied to or by objects in a class.
- All objects in a class share the same operations
- Operations encapsulate behavioral characteristics
- Method is an implementation of an operation of the class

Attributes & Operations



Class

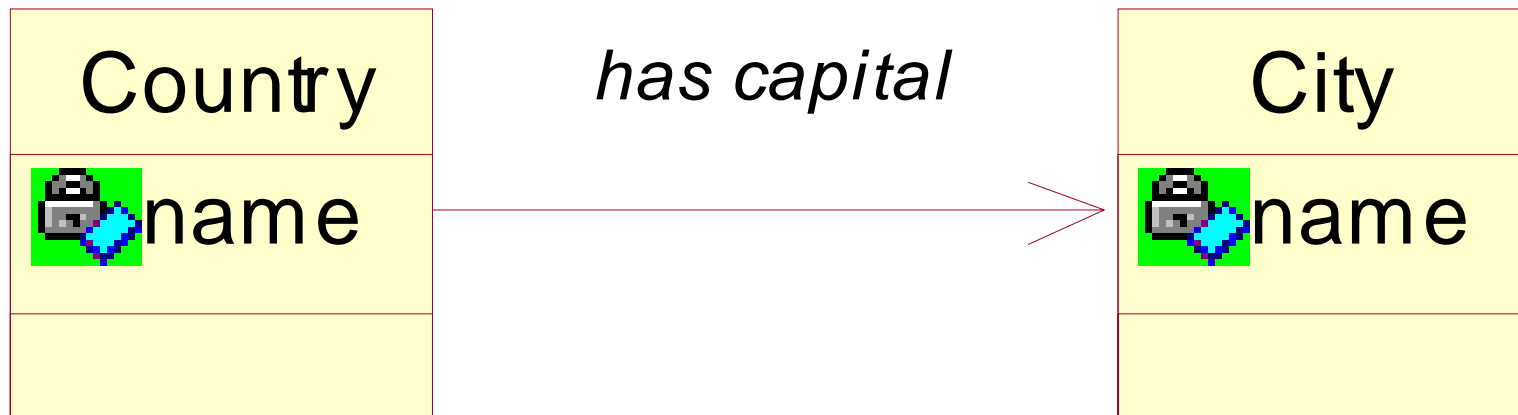


Object

Links & Associations

- A link is a physical or conceptual connection between object instances
- A link is an instance of an association
- An Association describes a group of links with common structure and common semantics
- Associations and Links appear as Verbs in the problem statements

Links & Associations



Object Orientation

- Object Orientation is a new way of thinking about problems using models organized around real world concepts.
- The fundamental construct is the “object”, which combines both data structure and behaviour in a single entity.
- Object Oriented, means that we organize software as a collection of discrete objects that incorporate both data structure and behaviour in a single entity.

Key Characteristics

- Identity
- Classification
- Polymorphism
- Inheritance

Identity

- Identity means that the data is quantized into discrete, distinguishable entities called “objects”
- Example : White board, Video Projector
- Objects can be concrete like a file in a directory or conceptual like a scheduling policy in a OS
- In a programming language each object has a unique handle by which it can be uniquely referenced. E.g., address, array index or unique value of an attribute.

Classification

- Objects with the same data structure (attributes) and behaviour (operations) are grouped into a class
- Example : boards, projectors
- A class is an abstraction that describes properties important to an application and ignores the rest.
- Each class describes a possibly infinite set of individual objects.
- Each object is said to be an instance of its class having its own values for attributes but sharing the attribute and operation names with other instances of the class.
- An object contains an implicit reference to its own class i.e. it “knows what kind of thing it is”

Polymorphism

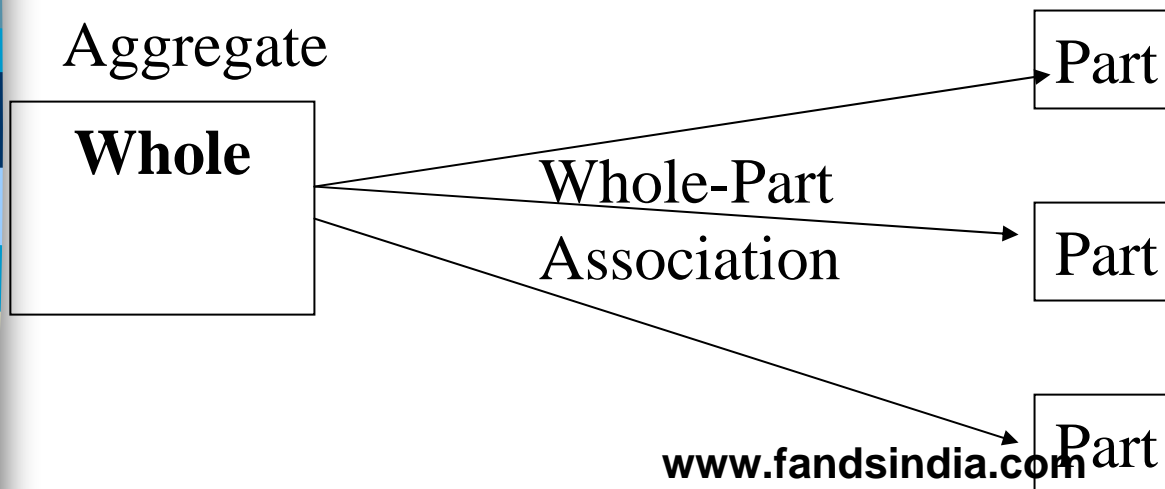
- Same operation may behave differently on different classes
- Example : createShape () method of Shapes class may have different implementations for a triangle and a square.
- An operation is an action/transformation that a object performs or is subject to.
- Each object knows how to perform its own operations.

Inheritance

- Sharing of attributes and operations among classes based on a hierarchical relationship.
- Example : Child inherits some attributes like skin color, temperament from their parents. Child adds his own attributes like speech, features, intelligence.
- A class can be defined broadly and then refined successively into finer subclasses.
- Each subclass incorporates or inherits all of the properties of its super class and adds its own unique properties.

Aggregations

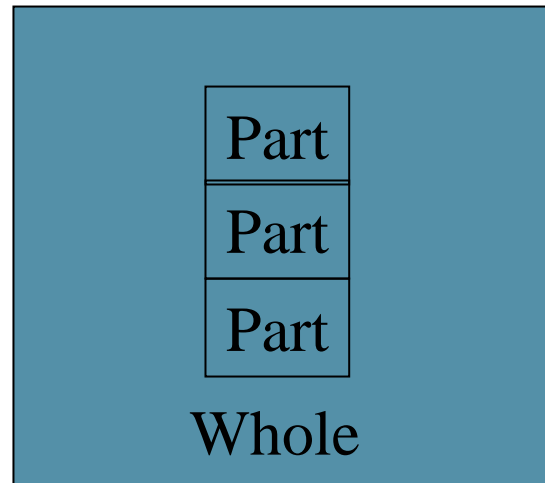
- Aggregations are known as “has-a” relationship
- These indicate a loosely coupled relations
- Parts exist independent of their Aggregate or whole.
- Example : A computer comprises of a monitor, system box, mouse and keyboard.



Compositions

- Compositions are contain-a relationship
- Component parts exist or live and die with their composite owner
- These indicate a tightly coupled relation
- Example : A document contains many paragraphs and a paragraph has many sentences

Composite



Generalizations

- Generalizations & inheritance are powerful abstractions for sharing similarities among classes while preserving their differences.
- Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called super class and each refined version is called subclass.
- Generalizations are called “is-a” relationship because each instance of a subclass is an instance of the super class as well.
- Inheritance is the mechanism of sharing attributes and operations using the generalization relationship.

Generalizations

- Are used to indicate OO sharing principles of Inheritance and Polymorphism
- Involve delegation, i.e., ability of of an object/class to issue a message to another object/class in response to a message.
- Example : Vehicles is the super class having attributes like color, no. of wheels, engine capacity and scooter, car, truck, bus are subclasses of Vehicles

Aggregation Vs Generalization

- Aggregation relates instances; Two distinct objects are involved; one of them is a part of the other.
- Generalization relates classes and is a way of structuring the description of a single object; both super class and subclass refer to properties of a single object
- Aggregation is sometimes called an “and-relationship”
- Generalization is sometimes called an “or-relationship”

OO Concept: Interface

- An object's interface characterizes the complete set of requests that can be sent to the object.
- An object's interface says nothing about its implementation—different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

Types

- Type is an abstraction of behavior: IDBConnection,
- A type typically denotes a particular Interface (in C#).
- An object may have many types (A Person object may be of types Employee and Parent)
- Widely different objects can share a type
- An object's interface says nothing about its implementation.
- We say that a type is a sub type of another if its interface contains the interface of its super type (inheritance among Interfaces)

Types vs. Classes

- Class is an abstraction of implementation.
- However, languages like C++ use classes to specify both an object's type and its implementation. In Java Types are interfaces and Classes are classes.
- Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

Advantages of Types over Classes

- Type hierarchy is as important class hierarchy. Why? Because polymorphism depends on it.
- Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect
- Clients remain unaware of the specific classes of objects they use

Inheritance vs. Composition

- Alternatives for reusing functionality
- Sub-classing is white-box reuse: With inheritance, the internals of parent classes are often visible to subclasses
- In Composition, new functionality is obtained by assembling or composing objects to get more complex functionality
- Composition is black-box reuse: No internal details of reused objects are visible

Inheritance - Advantages/ Disadvantages

■ Advantages

- Straight forward
- Easy to modify the implementation being reused (At individual function level - Over riding operations)
- supported directly by the programming language

■ Disadvantages

- Can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time
- Inheritance breaks encapsulation
- change in the parent's implementation will force the subclass to change: Fragile Base Class Problem
- What if inherited implementation is not appropriate ? Rewrite instead of reuse, limits flexibility and hence reusability

Composition : Advantages/Disadvantages

■ Advantages

- defined dynamically at run-time through objects acquiring references to other objects
- Forces objects to respect each others' interfaces, promoting encapsulation; preserves encapsulation
- Any object can be replaced at run-time by another as long as it has the same type
- fewer implementation dependencies
- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task

■ Disadvantages

- Not very straight forward; Dynamic software is harder to understand than more static software (run-time vs. compile time)
- system's behavior will depend on their interrelationships instead of being defined in one class

Run-time vs. Compile-time Structures

- The compile-time code structure is frozen and it consists of classes in fixed inheritance relationships
- A program's run-time structure consists of rapidly changing networks of communicating objects
- An object-oriented program's run-time structure often bears little resemblance to its code structure
- The system's run-time structure must be imposed more by the designer than the language.
- Think aggregation (ownership, permanent) vs. acquaintance (dynamic and temporary)

Summarizing the Design Themes

*Think differently about Types Vs. Classes
Program to an interface (Type), not an
implementation (Class).*

*Favor object composition over class inheritance,
black box over white box reuse.*

*Design relationships between
objects and their types to achieve good run-time
structure. Don't limit yourself to compile time
structures.*

Before Patterns



From Great Design

- Designers and architects always create their first version of the system as clean, elegant and compelling
- But then down the line something happens

Symptoms of Rotting Design

- There are four primary symptoms that tell us that our designs are rotting.
 - Rigidity
 - Fragility
 - Immobility
 - Viscosity

Rigidity

- Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules.
- What begins as a simple two day change to one module grows into a multi week activity of changes in modules as the engineers chase the thread of the change through the application

Fragility

- Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way.

Immobility

- Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote.
- However, it also often happens that the module in question has too much baggage that it depends upon.
- After much work, we discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

Viscosity

- Viscosity comes in two forms: viscosity of the design, and viscosity of the environment. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

Prevention better than cure

- How to cope with ?
 - Changing Requirements
 - Dependency Management
- Principles for good OOP/OOD

Principles of OO Class Design

- The Single Responsibility Principle
- The Open-Closed Principle (OCP)
- The Dependency Inversion Principle (DIP)
- Principle of Least Knowledge or Law of Demeter (LoD)
- The Interface Segregation Principle (ISP)

The Single Responsibility Principle

- *THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.*
- *This means*
 - *Class should have only a single purpose to live and all its methods should work together to help achieve this goal*
- *More than one responsibilities attached to the class make it*
 - *Difficult to understand (needlessly complex)*
 - *Difficult to change (rigid)*
 - *Difficult to reuse*

The Open Closed Principle

- *A module should be open for extension but closed for modification.*
- Most important principle
- It says that software entities should be designed so that they would not allow for changes to old code later on. New code can be added via extensions provided during design.

The Dependency Inversion Principle

- *Depend upon Abstractions. Do not depend upon concretions.*
- High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

Principle of Least Knowledge / Law of Demeter

- Any object receiving a message in a given method must be one of a restricted set of objects.
- When applied to object-oriented programs, the Law of Demeter can be more precisely called the “Law of Demeter for Functions/Methods” (LoD-F).
- An object A can call a method of an object instance B, but object A cannot “reach through” object B to access yet another object, C, to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B’s internal structure.

The Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they do not use.
- Many client specific interfaces are better than one general purpose interface.
- Impact of changes to one interface is relatively smaller.

Patterns



Categorization of Patterns

- **Creational Patterns:** Used to create objects instead of direct instantiation
- **Structural Patterns:** Compose objects into larger structures
- **Behavioural Patterns:** Communication between objects

Creational Pattern

Abstract the Instantiation Process.

Makes a system independent of how its objects are created, composed and represented.

A *Class* creational pattern uses inheritance to vary the class that has been instantiated.

An *Object* creational pattern delegates instantiation to another object.

Creational Patterns

- **Factory** – depending on the data provided the Factory pattern returns an instance of a particular class
- **Abstract Factory** – a further abstraction of Factory that determines a group of instances to return
- **Singleton** – maintains only a single instance of a class
- **Builder** – a more complex pattern than Factory that returns an entire user interface based on the context of the data
- **Prototype** – clones an existing instance rather than creating new instances of a class

Structural Pattern

These are concerned with how classes and objects are composed to form larger structures.

Structural *Class* patterns use inheritance to compose interfaces or implementations.

Structural *Object* patterns describe ways to compose objects to realize new functionality.

Structural Patterns

- **Adapter** - changes the interface of one class to that of another one.
- **Bridge** – keeps the program's interface constant while changing the actual class that is displayed or being used. The interface and the underlying class can be changed separately.
- **Composite** – is a collection of objects which may be either a Composite or just a primitive object
- **Decorator** - a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- **Façade** - groups a complex object hierarchy together and provides a simpler interface to the underlying data.
- **Flyweight** - limits the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- **Proxy** - provides a simple place-holder class for a more complex class which is expensive to instantiate.

Behavioral Patterns

These are concerned with algorithms and the assignment of responsibility between objects. These describe not just patterns of objects or classes but also the patterns of communication between them.


Behavioral *Class* Patterns use inheritance to distribute behavior between classes.

Behavioral *Object* patterns use object composition.

Behavioral Patterns

- **Observer** - defines the way a number of classes can be notified of a change,
- **Mediator** - defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other.
- **Chain of Responsibility** - allows an even further decoupling between classes, by passing a request between classes until it is recognized.
- **Template** - provides an abstract definition of an algorithm
- **Interpreter** - defines how to include language elements in a program.
- **Strategy** - encapsulates an algorithm inside a class,
- **Visitor** - adds function to a class
- **State** - provides a memory for a class's instance variables.
- **Command** - provides a simple way to separate execution of a command from the interface environment that produced it
- **Iterator** - formalizes the way to move through a list of data within a class.
- **Memento** - Define an object that encapsulates how a set of objects interact

Adapter Pattern

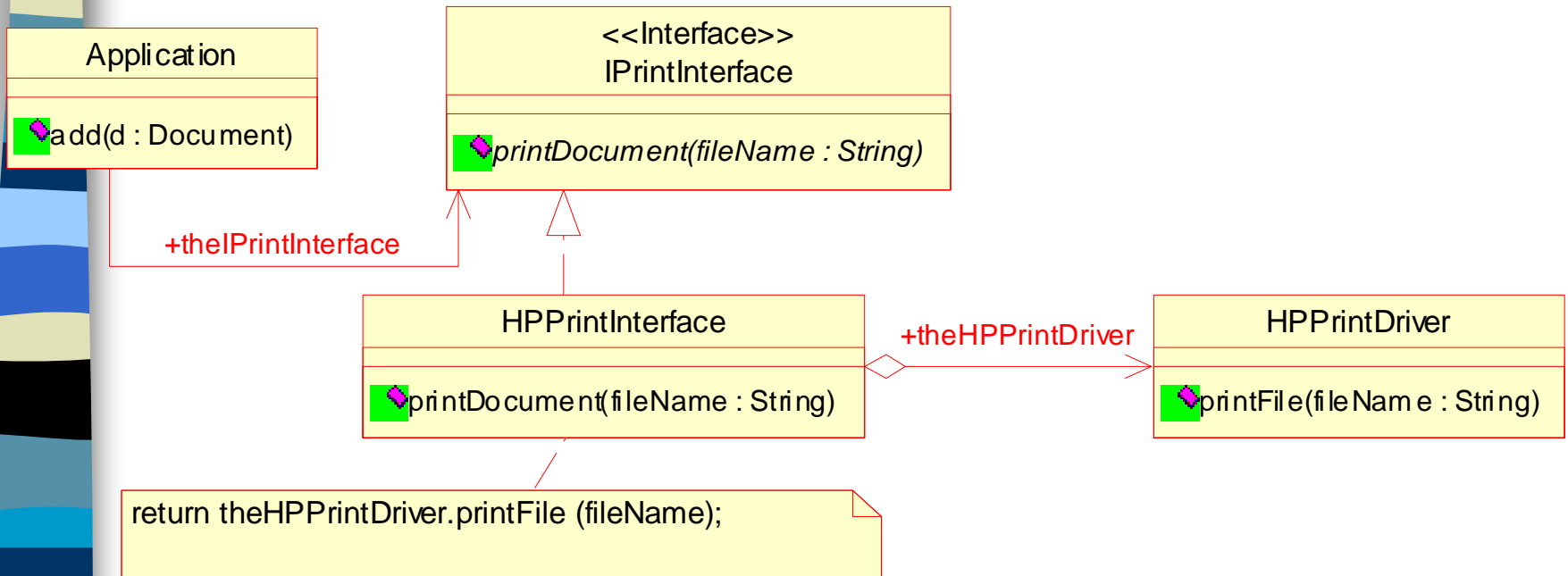


Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

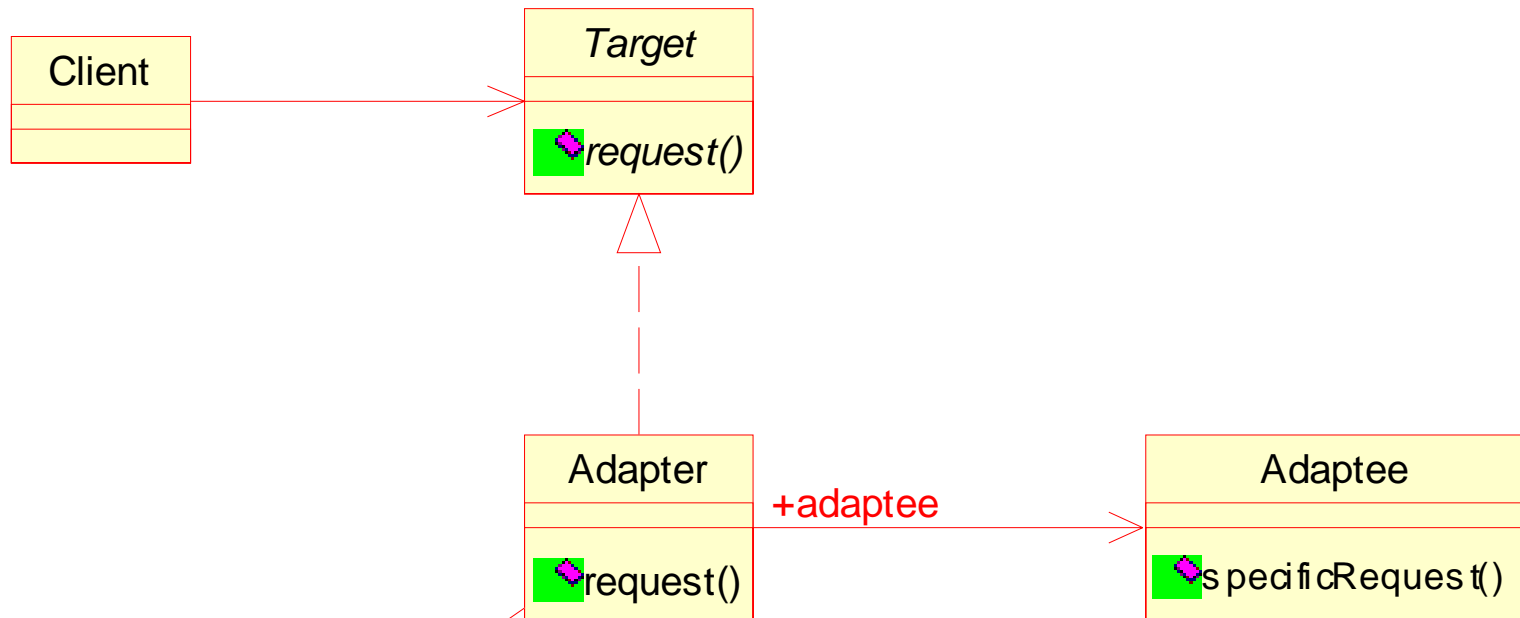
Adapter: Applicability

- 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.
- you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

Adapter: Example



Adapter: Model



return adaptee.specificRequest ();

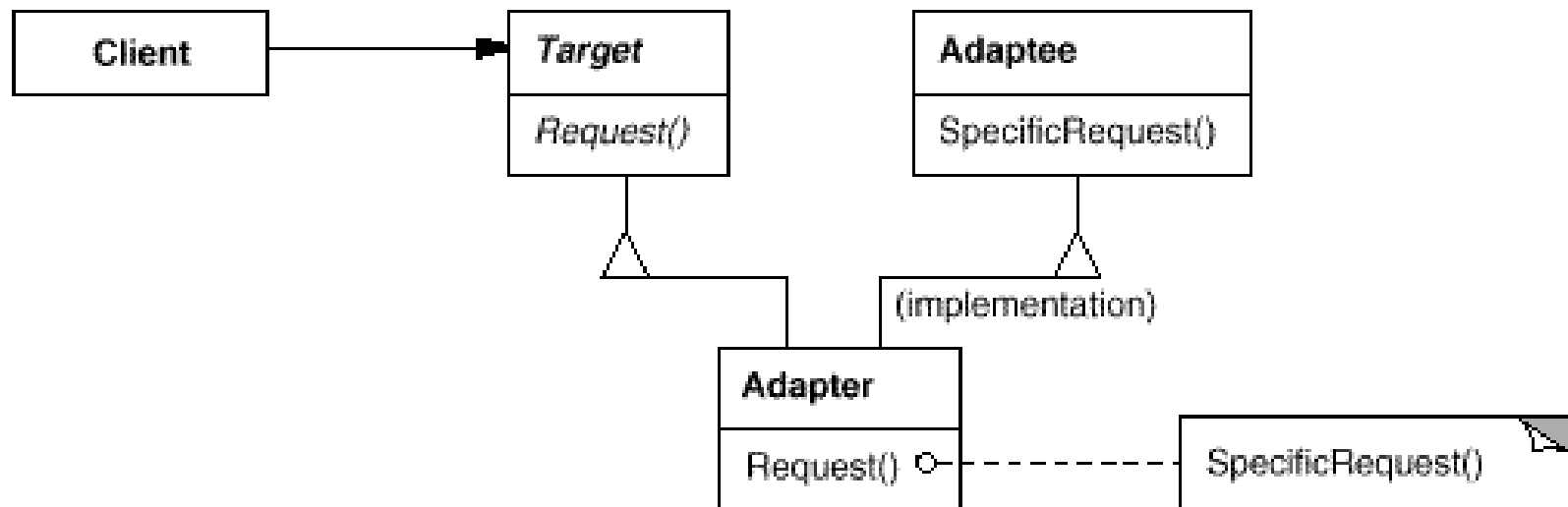
Adapter: Collaborations

- Target (PrintInterface) defines the domain-specific interface that Client uses.
- Client (Application) collaborates with objects conforming to the Target interface.
- Adaptee (HPPrintDriver) defines an existing interface that needs adapting.
- Adapter (HPPrintInterface) adapts the interface of Adaptee to the Target interface.

Adapter: Consequences

- Variation: Instead of delegating to an Adaptee, can an Adapter inherit from an Adaptee ? What are the consequences ?
- The amount of work Adapter does depends on how similar the Target interface is to Adaptee's; Varies from simple interface conversion to providing complex operations


Class Adapter



Class Vs. Object Adapter

- Class Adapter:
 - adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
 - lets Adapter override some of Adaptee's behavior
 - introduces only one object, and no additional pointer indirection
- Object Adapter
 - lets a single Adapter work with many Adaptees; that is, the Adaptee itself and all of its subclasses (if any)
 - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself

Strategy Pattern

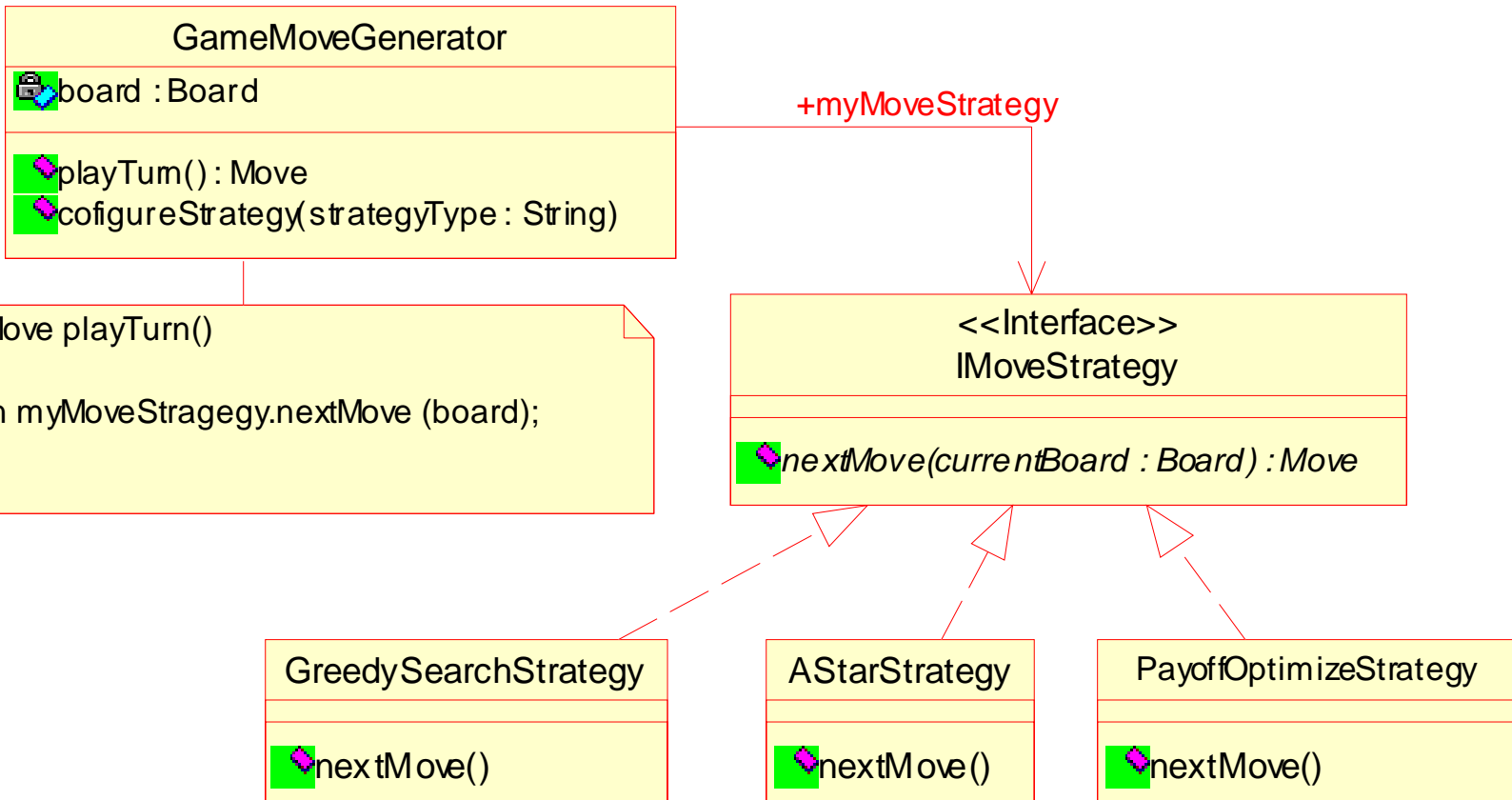


Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

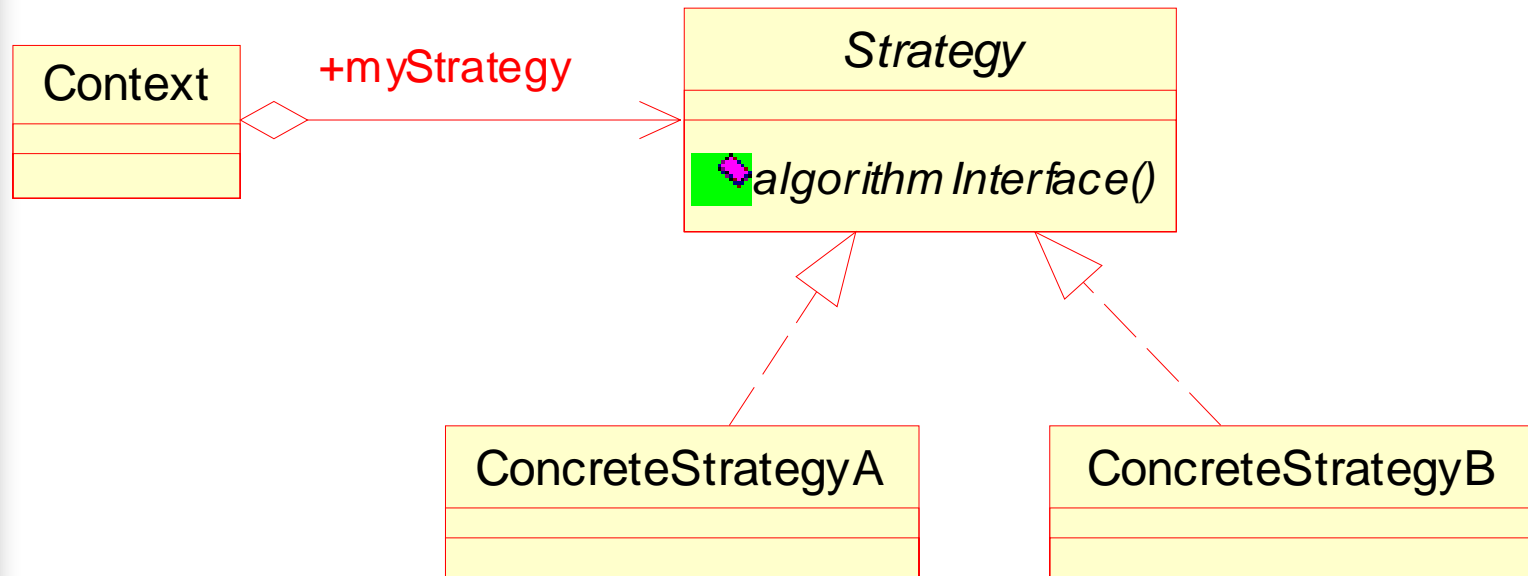
Strategy: Applicability

- 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

Strategy: Example



Strategy: Model



Strategy: Collaborations

- Strategy (IMoveStrategy) declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a concreteStrategy.
- ConcreteStrategy (GreedySearchStrategy/ AStarStrategy) implements the algorithm using the Strategy interface.
- Context (GameMoveGenerator)
 - is configured with a ConcreteStrategy object.
 - maintains a reference to a Strategy object.
 - may define an interface that lets Strategy access its data.

Strategy: Consequences

- Families of related algorithms. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
- Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend. Using inheritance of Context object to achieve difference in behavior hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically.
- Strategies eliminate conditional statements.
- The client can choose among strategies with different time and space trade-offs.
- Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share

Anti Patterns



AntiPattern

- An AntiPattern is a pattern that tells how to go from a problem to a bad solution. (Contrast to a pattern that tells how to go from a bad solution to a good solution.)
- A good AntiPattern also tells you why the bad solution looks attractive (e.g. it actually works in some narrow context), why it turns out to be bad, and what positive patterns are applicable in its stead.

AntiPattern

- Accordingly to JimCoplien: an anti-pattern is something that looks like a good idea, but which backfires badly when applied.
- In the old days, we used to just call these 'bad ideas'. The new name is much more diplomatic.
- An anti-pattern is a stereotyped "Hall of Shame" entry. (i.e., "My purpose in life is to serve as a warning to others.")

Warning Signs in Design/Code

- Duplicated Code/Functionality
- Long Methods
- Large Class
- Long Parameter Lists
- Shotgun Surgery
- Feature Envy
- Primitive Obsession

More Warning Signs

- Switch statements
- Temporary Field
- Middle Man
- Message Chains

What to Expect?

- Common Design Vocabulary.
- A documentation and Learning Aid.
- How to use primitive techniques such as objects, inheritance, polymorphism.
- Help in determining how to reorganize a design in the face of requirement changes.

Pros of Patterns

- **Design patterns enable large-scale reuse of software architectures**
- **Patterns explicitly capture expert knowledge and design tradeoffs, and make this expertise more widely available**
- **Patterns help improve developer communication**
- **Patterns help ease the transition to object-oriented technology**

Pattern Cons

- **Patterns do not lead to direct code reuse**
- **Patterns are deceptively simple**
- **Teams may suffer from pattern overload**
- **Patterns are validated by experience and discussion rather than by automated testing**
- **Integrating patterns into a software development process is a human-intensive activity**

References

1. EJB Design Patterns by Floyd Marinescu - Wiley.
(available online at **www.theserverside.com** and
\\Kalpa)
2. J2EE Design Patterns Catalog -
<http://java.sun.com/blueprints/patterns/catalog.html>
3. Core J2EE Patterns (online book) -
<http://java.sun.com/blueprints/corej2eepatterns/Patterns/>
4. Sun Java Center J2EE Patterns -
<http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html>

Conclusion

- The Idea behind Design Patterns is simple: to catalogue common interactions between objects that programmers have often found useful.
- Three step process in learning design patterns:
 - Acceptance
 - Recognition
 - Internalization

QUESTION / ANSWERS



QUESTION / ANSWERS



THANKING YOU !

