



AARHUS UNIVERSITET

Software Engineering and Architecture

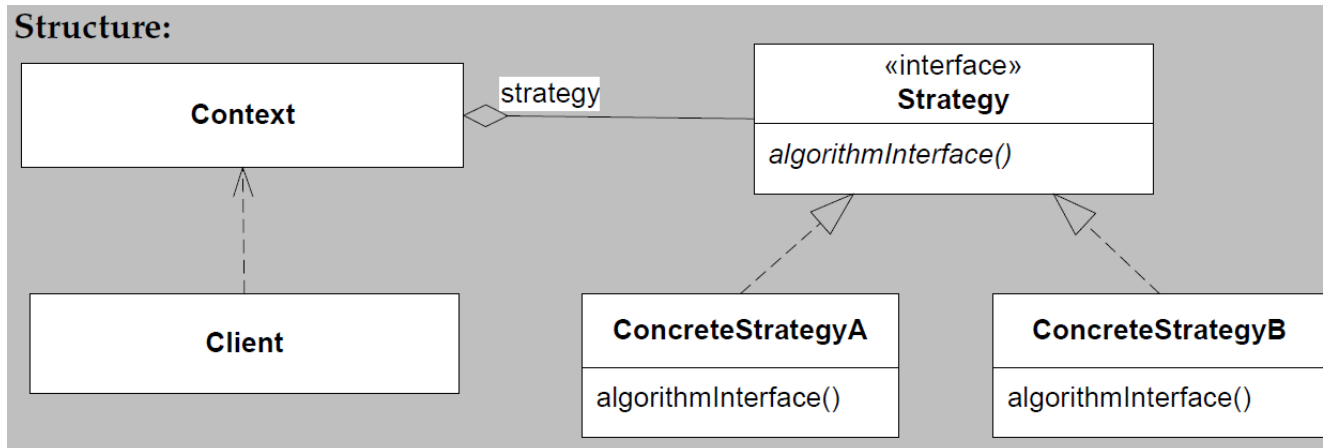
Design Patterns I

Gamma et al.'s definition

Definition: Design Pattern (Gamma et al.)

Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- Exercise: How does it relate to...





Beck et al.'s Definition

Definition: Design Pattern (Beck et al.)

A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future.

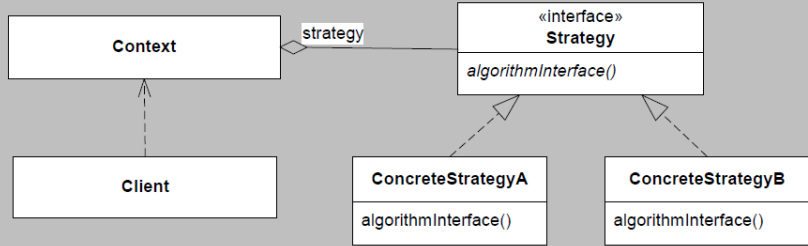
- Prose form = “writing template”
- The template varies from author to author.
- However, must contain
 - Name
 - Problem
 - Solution
 - Consequences

FRS's template

- Intent
 - Short description

- Roles
 - Responsibilities of each participating object/abstraction in the pattern

[7.1] Design Pattern: Strategy

Intent	Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.
Problem	Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.
Solution	Separate selection of algorithm from its implementation by expressing the algorithms responsibilities in an interface and let each implementation of the algorithms realize this interface.
Structure:	 <pre> classDiagram class Strategy { <<interface>> algorithmInterface() } class Context class Client class ConcreteStrategyA { algorithmInterface() } class ConcreteStrategyB { algorithmInterface() } Context o-- Strategy : strategy Client ..> Context Strategy < .. ConcreteStrategyA Strategy < .. ConcreteStrategyB </pre>
Roles	Strategy specifies the responsibility and interface of the algorithm. ConcreteStrategies defines concrete behavior fulfilling the responsibility. Context performs its work for Client by delegating to an instance of type Strategy .
Cost - Benefit	The benefits are: <i>Strategies eliminate conditional statements. It is an alternative to subclassing. It facilitates separate testing of Context and ConcreteStrategy.</i> The liabilities are: <i>Increased number of objects. Clients must be aware of strategies.</i>

- Simon Kracht: FRS Design Pattern Poster

Design Patterns
(According to Flexible, Reliable Software - Using Patterns and Agile Development by Henrik Baerbak Christensen)

<p>Abstract Factory</p> <p>Problem: "Families of related objects need to be instantiated. Product classes need to be consistently configured." Example of use: Testing the output of a Calculator when each calculator has its own factory.</p>	<p>Decorator</p> <p>Problem: "You want to attach responsibilities to an object without modifying its class." Example of use: Testing the output of a Printer.</p>	<p>Command</p> <p>Problem: "You want to configure objects with behaviors (methods) or other objects support used." Example of use: Start and stop action by your remote control in a GUI.</p>	<p>Strategy</p> <p>Problem: "Your product needs require an interchangeable algorithm or business logic and you want a flexible and reusable way of encoding the variability." Example of use: Pushbutton control and the remote control calculator in a calculator.</p>
<p>Builder</p> <p>Problem: "You have a single defined construction process but the output format varies." Example of use: Read generic record files.</p>	<p>Facade</p> <p>Problem: "The complexity of a subsystem should not be exposed to its clients." Example of use: Call to the Internet.</p>	<p>Iterator</p> <p>Problem: "You want to traverse a collection without worrying about the representation details of it." Example of use: Iterator over a collection of nodes in a graph of Cities.</p>	<p>Template Method</p> <p>Problem: "There is a need to have different behavior of some class of an algorithm but the structure of the algorithm is otherwise fixed." Example of use: Sort parts of the hierarchical web page, calculate similar probability algorithms.</p>
<p>Adapter</p> <p>Problem: "You have a class with desirable functionality but its interface isn't practical due not being able to fit into the existing API." Example of use: Connect an old calculator with a calculator interface.</p>	<p>Proxy</p> <p>Problem: "You need to control object's legitimate effects. The resource requirements of the client and the object are not equal or controlling the client's access to the resource is needed." Example of use: Store and retrieve pictures from a database and log each request to a database.</p>	<p>Observer</p> <p>Problem: "A set of objects needs to be notified if a condition object changes. Object to ensure appropriate response of all subscribers. They need to be notified by the object to receive appropriate new." Example of use: Control computer window handles by condition on a window.</p>	<p>Creational</p> <p>Abstract Factory - Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Page 17 (2002).</p> <p>Builder - Specify the construction of a complex object from its representation allowing the same construction process to create other representations. Page 18 (2002).</p> <p>Structural</p> <p>Adapter - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that don't otherwise because of incompatible interfaces. Page 19 (2002).</p> <p>Composite - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Page 20 (2002).</p> <p>Decorator - Attach additional responsibilities to an object dynamically by wrapping the same interface. Decorators contain a flexible mechanism to packaging for extending functionality. Page 20 (2002).</p> <p>Facade - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that shields the sub-system's complexity. Page 21 (2002).</p> <p>Iterator - Provide a simple and standard for another object to control access to it. Page 21 (2002).</p> <p>Null Object - Provide a null reference to objects that implement a particular interface. Page 22 (2002).</p>
<p>Composite</p> <p>Problem: "Handling of tree-like structures." Example of use: Represented folders and files in a file system, Custom controls collected in a Component.</p>	<p>Null Object</p> <p>Problem: "The absence of an object or behavior is often represented by referencing null. However, this leads to common code ensuring that the method is executed only." Example of use: Test the engine's behavior during an automatic testing.</p>	<p>State</p> <p>Problem: "You probably behavior when an condition depending on an external state." Example of use: Pushbutton for controlling strategy (Context object also implements State, State implements Strategy).</p>	<p>Behavioral</p> <p>Command - Represent a request as an object, thereby letting you parameterize clients with different requests, release the request and request responsible objects. Page 22 (2002).</p> <p>Decorator - Provide a easy to learn the structure of an aggregate object recursively without exposing its composing components. Page 22 (2002).</p> <p>Observer - Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically. Page 23 (2002).</p> <p>State - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. Page 23 (2002).</p> <p>Strategy - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Page 23 (2002).</p> <p>Template Method - Define the skeleton of an algorithm in an abstract, defining some steps to subclasses. The concrete subclasses override those steps that need to be customized. Page 24 (2002).</p> <p>Principles for Flexible Design</p> <ul style="list-style-type: none"> ⊗ Express an interface not an implementation ⊗ Favor object composition over class inheritance ⊗ Consider what should be variable in your design (precisely the behavior that varies) <p>UML</p> <ul style="list-style-type: none"> ⊗ -> Association ⊗ - > Generalization or local variable ⊗ - .. > Generalization or interface ⊗ - .. > Generalization or interface ⊗ - .. > Generalization or interface



Differentiating Patterns

- Be aware that many patterns are *structurally equal* – their UML class diagrams are more or less identical!
- Patterns are defined by the *problem they solve!*
- Strategy is the problem of
- *Handling variability of algorithms / business rules, making them interchangeable.*