Software Engineering

Lesson Design Pattern
Strategy
v1.0a

Uwe Gühl

Winter 2007/ 2008

Strategy

- Intent:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it
  - known as “Policy” as well
  - is a Behavioral Pattern
  - It describes a way to abstract out often changing code from stable code. Strategy encapsulates this often changing code so that it can be used or interchanged easily
  - It encapsulates an algorithm inside a class

Strategy

- Motivation
  There exist many variants to save data for different applications, e. g.
  - File compression (zip, rar)
  - Video data compression (MPEG, AVI, Quicktime, ...)
  - Savings of image data (BMP, GIF, JPEG, ...)
  - Save files in different formats

  Additional examples
  - Evaluation strategy for multiple choice tests
  - Different line-breaking strategies to display text data

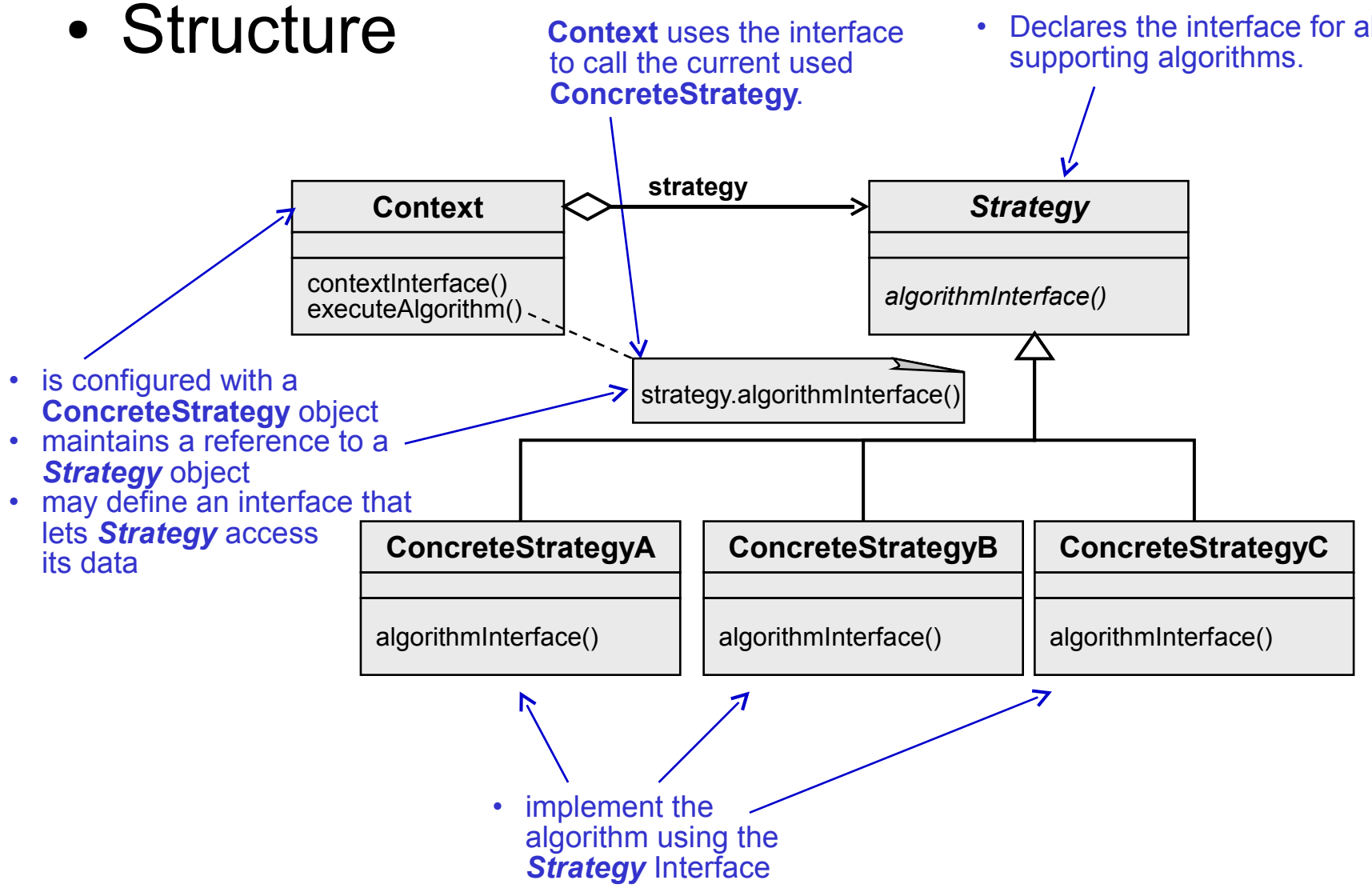
Strategy

- Idea
  - Outsourcing of the variants of an algorithm in own classes, all with the same interface to an application

Source: robgarrett.com

Strategy

- Structure

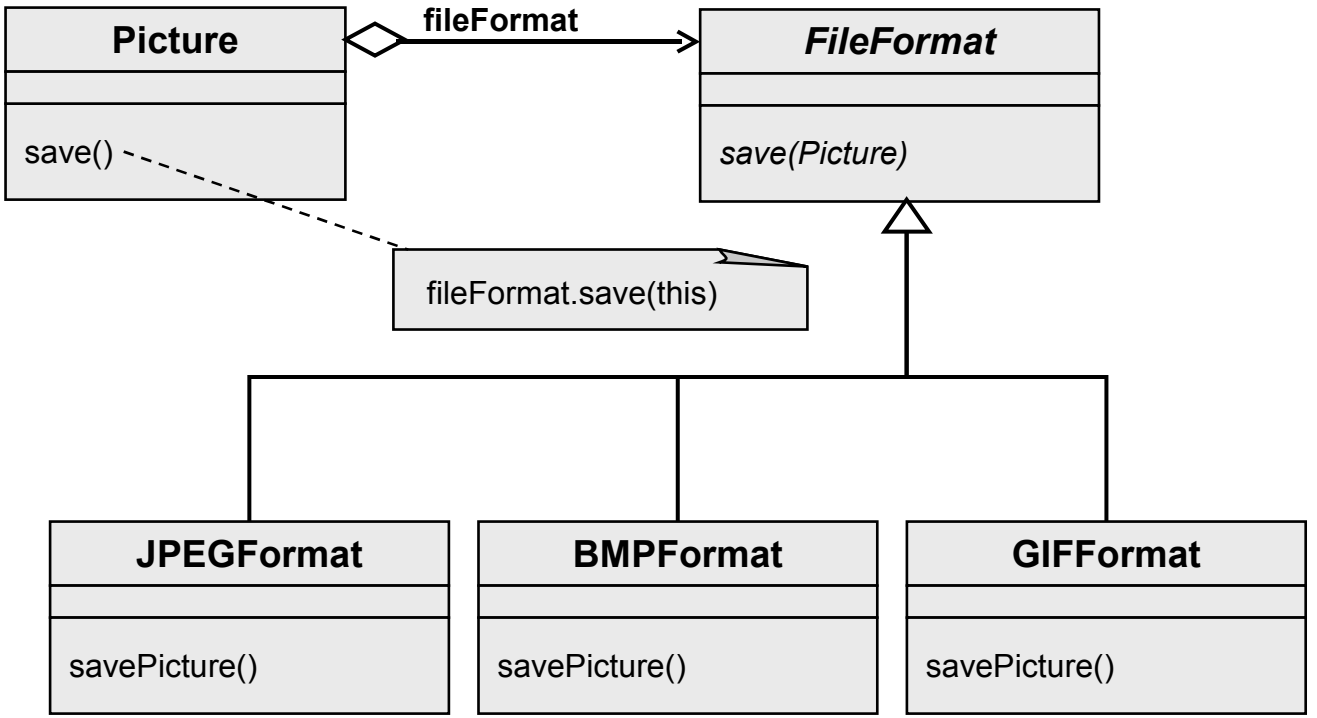


Strategy

- Collaboration
  - ***Strategy*** and **Context** collaborate to implement the chosen algorithm. **Context** could
    - either pass all necessary data for the algorithm
    - or pass itself, so that ***Strategy*** could call back.
  - **Context** could pass requests from its clients to its **strategy**.
    Often exists a family of **ConcreteStrategy** classes a client could choose from.

Strategy

- Example



Strategy

- Applicability
  Use the Strategy Pattern when
  - many similar classes differ only in behavior. Strategy could configure a specific class with one of many possible behaviors
  - different variants of an algorithm are needed. For example to offer different storage or time properties
  - an algorithm needs data, that clients should not know about. These data could be associated to particular strategies.

Strategy

- Applicability
  Use the Strategy Pattern when
  - a class has different behaviors, structured over many complex condition levels.
    Instead of administrating many different conditions, the use of own strategy classes is easier to handle

Strategy

- Consequences

+ Families of related algorithms get extracted and combined for potential reuse
+ More flexible alternative to subclassing of the context object. A dynamical configuration offers variability for different algorithms or behaviour. With Strategy it's easier to understand, to change, and to extend
+ Alternative to conditions (if else, switch)
  The choice of implementation alternatives of the same behaviour could be made with the Strategy Pattern instead of many condition statements

Strategy

- Consequences

\+ Clients could choose among strategies

o Clients must be aware of different strategies

– Communication overhead between Strategy and Context, e. g. if a complex ConcreteStrategy needs more information from Context then a simple ConcreteStrategy.
Reason: The interface has to fit for all strategies

– Increased number of objects
A workaround could be implementing strategies as stateless objects that contexts can share
(→ Flyweight Pattern)

Strategy

- Implementation
  - Definition of Strategy and Context interfaces; possible strategies:
    - Pass related data as parameter
    - Context passes itself as parameter (“`this`”)
    - Two way reference with closer coupling
  - Strategy as option
    The default behaviour is described in context.
    Only if a strategy object is available, an external algorithm is used.

Strategy

- Known Uses
  - `java.util.zip package`
    CheckedInputStream / CheckedOutputStream use different checksum calculation algorithms*
    - CRC32 – very reliable, but the algorithm is computationally expensive
    - Adler-32 – almost as reliable as CRC32, but the calculation of the checksum can be done more quickly than using CRC32

* A checksum could be used to ensure a file was correctly transferred. Typically, before a file is transferred, you calculate the checksum for the file. Once it has been received, you calculate the checksum again. If the two values match, the file has been transferred correctly.
If not, an error occurred during the transmission.
(Source: http://people.westminstercollege.edu/faculty/ggagne/may2006/lab7/index.html)

Strategy

- Related Patterns
  - State Pattern
    The class diagrams look similar, but the intension of the patterns is different
    - The State Pattern defines one particular implementation for changing behaviors that differ from one state to another.
    - The Strategy Pattern defines a way to have many, separately encapsulated algorithms that are interchangeable.
  - Flyweight
    Strategy objects often make good flyweights