Decorator Design Pattern
A composable alternative to subclassing
S. Ducasse, L. Fabresse, G. Polito, and P. Tesone

http://www.pharo.org
Goals

- Present the Decorator Design Pattern
- Think about API
Decorator

From the book:

- **Dynamically** attach additional responsibilities to an object
- Decorators provide a flexible alternative to subclassing for extending functionality
Decorator core
Often mixed with inheritance
A decorator wraps a decoree

- It is placed between the client and the decoree
- It propagates or not messages to the decoree

Easier to understand when the Decorator is a subclass of Decoree but not necessary (think duck typing)
Decorator nesting

A decorator wraps an instance or decorated instance of the component
Transparent to the client

- A client manipulates **transparently** decorated and undecorated elements
- A client talks to the decorator which **delegates** to the decoree (a leaf object or another decorator)
- **Strong Implication:** decoree and decorator **must** expose the same API
Example of Stream

ZnStreams are decorators of Streams

ZnNewLineWriterStream
   on: (ZnCharacterWriteStream on: Stdio stdout encoding: 'utf8').

- ZnNewLineWriterStream **decorates** ZnCharacterWriteStream
Another use

AbstractFileReference >> readStreamEncoded: anEncoding

^ ZnCharacterReadStream
  on: self binaryReadStream
  encoding: anEncoding

- ZnCharacterReadStream is decorating another stream with an encoding
Implementation

WriteStream << #ZnNewLineWriterStream
slots: { #stream . #cr . #lf . #previous . #lineEnding};
package: 'Zinc~Character~Encoding~Core'

ZnNewLineWriterStream class >> on: aStream
^ self basicNew
  initialize;
  stream: aStream;
  yourself

ZnNewLineWriterStream >> close
stream close

ZnNewLineWriterStream >> flush
^ stream flush
testNextPutEnsureLineEndsAreWrittenCorrectly

| expectedString stream crStream |
expectedString := 'a', OSPlatform current lineEnding, 'b'.
{ String cr . String lf . String crlf } do: [ :lineEnd |
  stream := String new writeStream.
  crStream
    << 'a';
    << lineEnd;
    << 'b'.
  self assert: stream contents equals: expectedString ]
Example of Stream (II)

ZnNewLineWriterStream >> nextPut: aCharacter
"Write aCharacter to the receivers stream.
Convert all line end combinations, i.e cr, lf, crlf, to the platform convention"

(previous == cr and: [ aCharacter == lf ]) ifFalse: [
(aCharacter == cr or: [ aCharacter == lf ])
ifTrue: [ self newLine ]
ifFalse: [ stream nextPut: aCharacter ]].
previous := aCharacter.
All decorators should have the same API
- close, flush, nextPut:, contents, next, atEnd, on:
- Stream decorator individual behavior can be reused and composed
About dynamic behavior

Decorators attach additional responsibilities to an object

- The decorator is based on delegation
- We should control the creation of the decoration chain (the client reference)

**Strong Implication:** decorated objects **do not know** if they are decorated
  - Changing the decoration chain at runtime is not simple
When not to use decorator

- When decorations have different APIs
- When the decorations should change dynamically
- Think twice when the APIs are HUGE
Conclusion

- Decorators can represent composable facets of an object
- Pay attention all the decorators should implement the same API
- Decorator is modular but within a common API