Design Patterns & Software Architecture
Strategy pattern

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- What are patterns
- The Strategy pattern
- Design Objectives & Design Principles

What are software design patterns?

- We already saw the definition…
- They help when you're in the situation where you think: *"Oh I've seen this problem before and I solved it by …"*
- They generally require some modification when you apply them…

What are software design patterns?

Non-software example: “*Last time I put up a book shelf it was straight, but this time I need a corner bookshelf*”…

In the same way patterns describe how to solve a general problem

- You modify and interpret for your context…

What patterns are not…

- Not concrete designs…
  - Patterns must be instantiated in a particular context…
- They do not remove creativity or human judgment…
- Do not remove implementation…
- They are not frameworks…

Why use design patterns?

- Reuse of design expertise…
- Improve communication between engineers: A common vocabulary…
- Many software design patterns lead to software that is flexible to change…

Are there any problems with software design patterns?

- Trade-off: design can become a little more complicated…
- Inexperienced users often try to use more software design patterns than they need to
  - You need to ask yourself why you’re using the design pattern…

Pattern origins

- Software design patterns were inspired by the work of Christopher Alexander (architect)
  - Developed a pattern language for describing architectural features in buildings (1977) …
- Seminal work of 23 software design patterns in:
  - E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1996
  - Not all 23 design patterns of the original Design Patterns book will be treated during the course.
- Many other books and web sites:
  - www.hillside.net/patterns
  - PLoP events: Pattern Languages of Programming

Strategy pattern

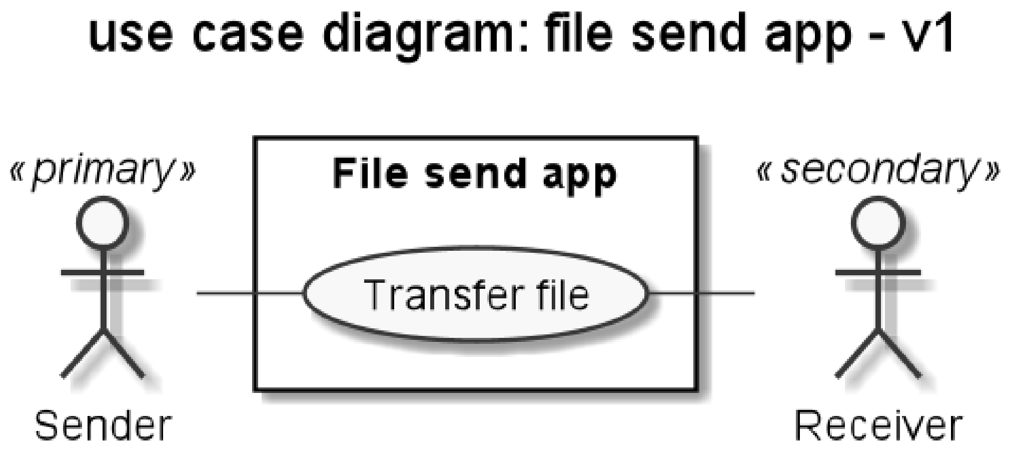
Let’s find a design pattern: Strategy pattern

Will now present, on the board, a problem that we wish to solve and that will lead to the application of a design pattern.

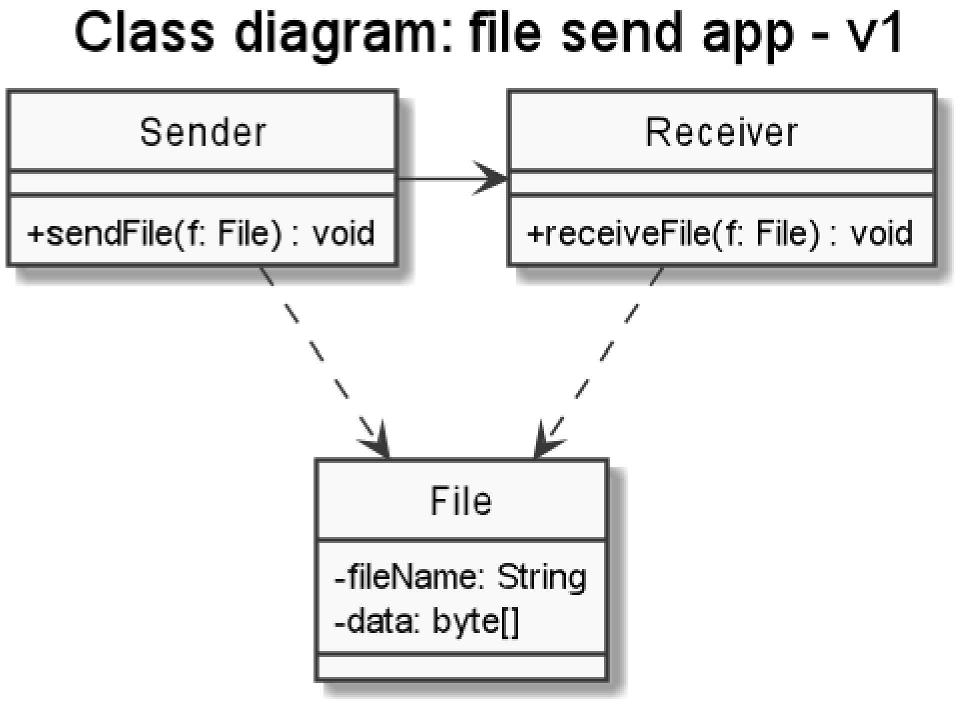
File sender application: Requirements - Initial

Lets make an application that can transfer files between the computer and a mobile phone, with the following requirements:

- Should be an app able to send files
- Should be an app able to receive files
- There should be a File class to store the data
- The system needs to be able to do a transfer



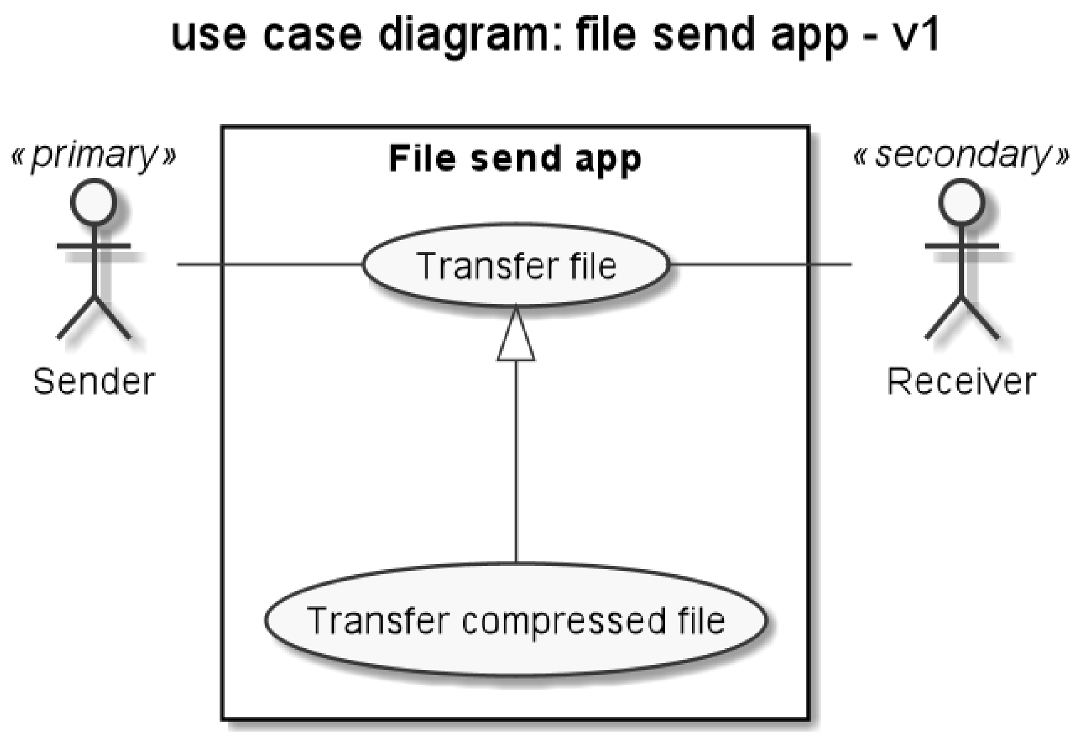
File sender application: Design - Initial


Class diagram: file send app - v1
Sender
+sendFile(f: File) : void
Receiver
+receiveFile(f: File) : void
File
-fileName: String
-data: byte[]

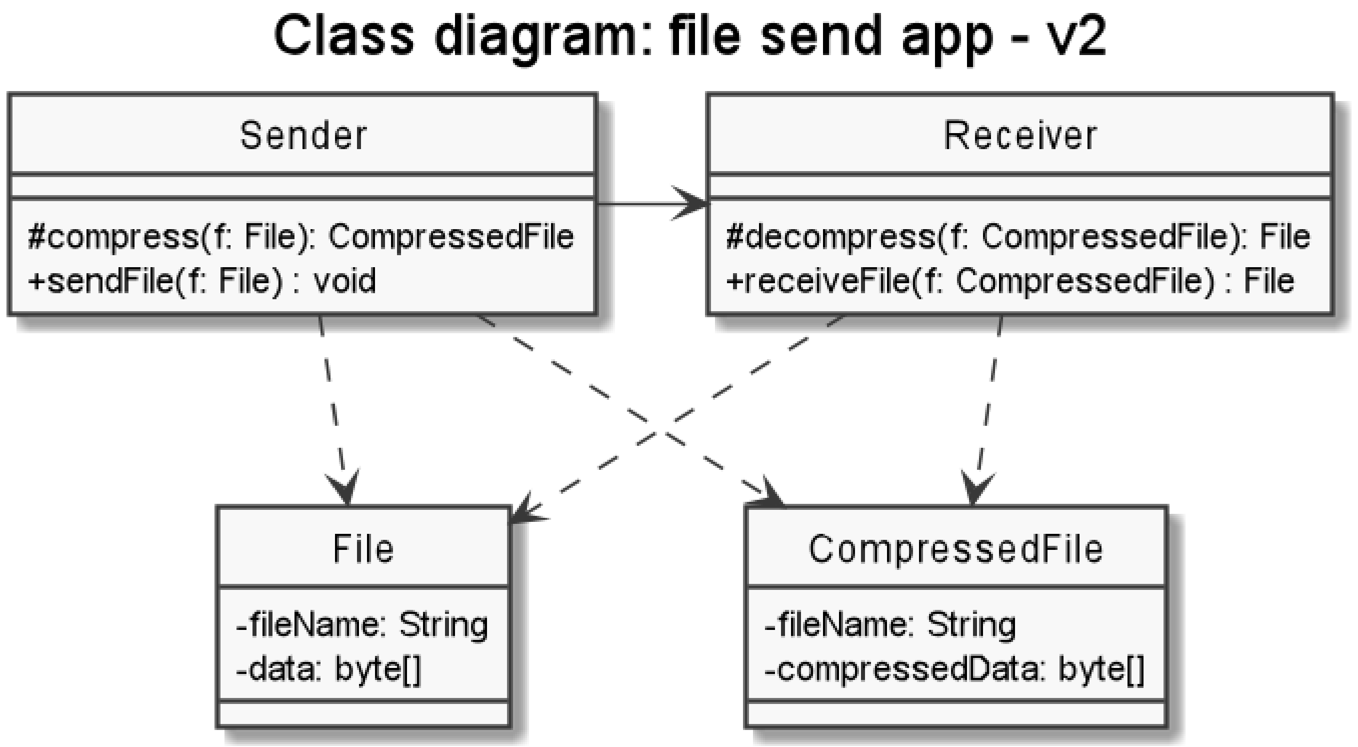
File sender application: Requirements – Additional requirements

Lets make an application that can transfer files between the computer and a mobile phone, with the following requirements:

- Should be an app able to send files
- Should be an app able to receive files
- There should be a File class to store the data
- The system needs to be able to do a transfer
- The system needs to be able to do a transfer with compression



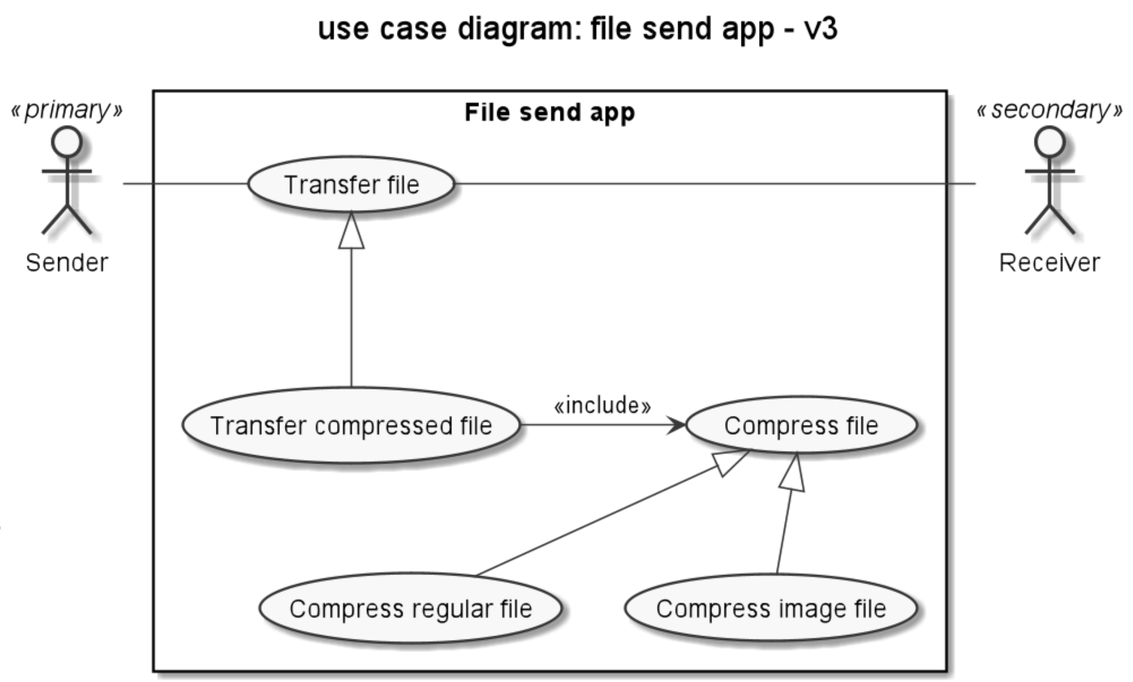
File sender application: Design – Next iteration


Class diagram: file send app - v2
Sender
#compress(f: File): CompressedFile
+sendFile(f: File) : void
Receiver
#decompress(f: CompressedFile): File
+receiveFile(f: CompressedFile) : File
File
-fileName: String
-data: byte[]
CompressedFile
-fileName: String
-compressedData: byte[]

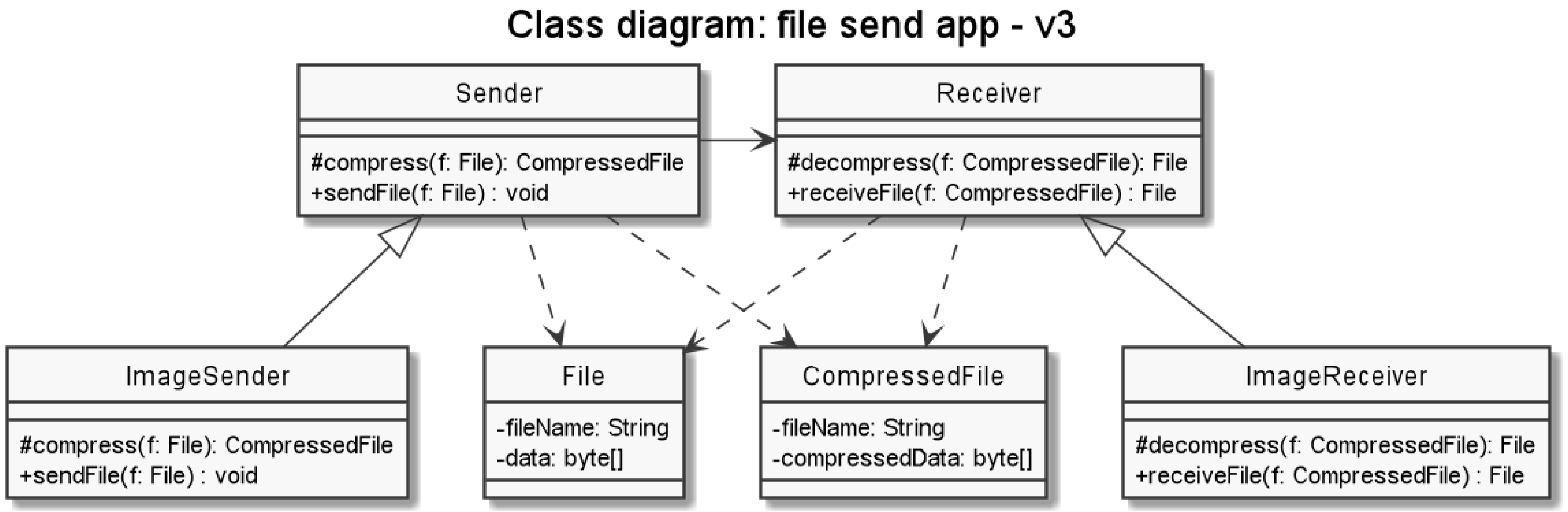
File sender application: Requirements – Additional requirements 2

Lets make an application that can transfer files between the computer and a mobile phone, with the following requirements:

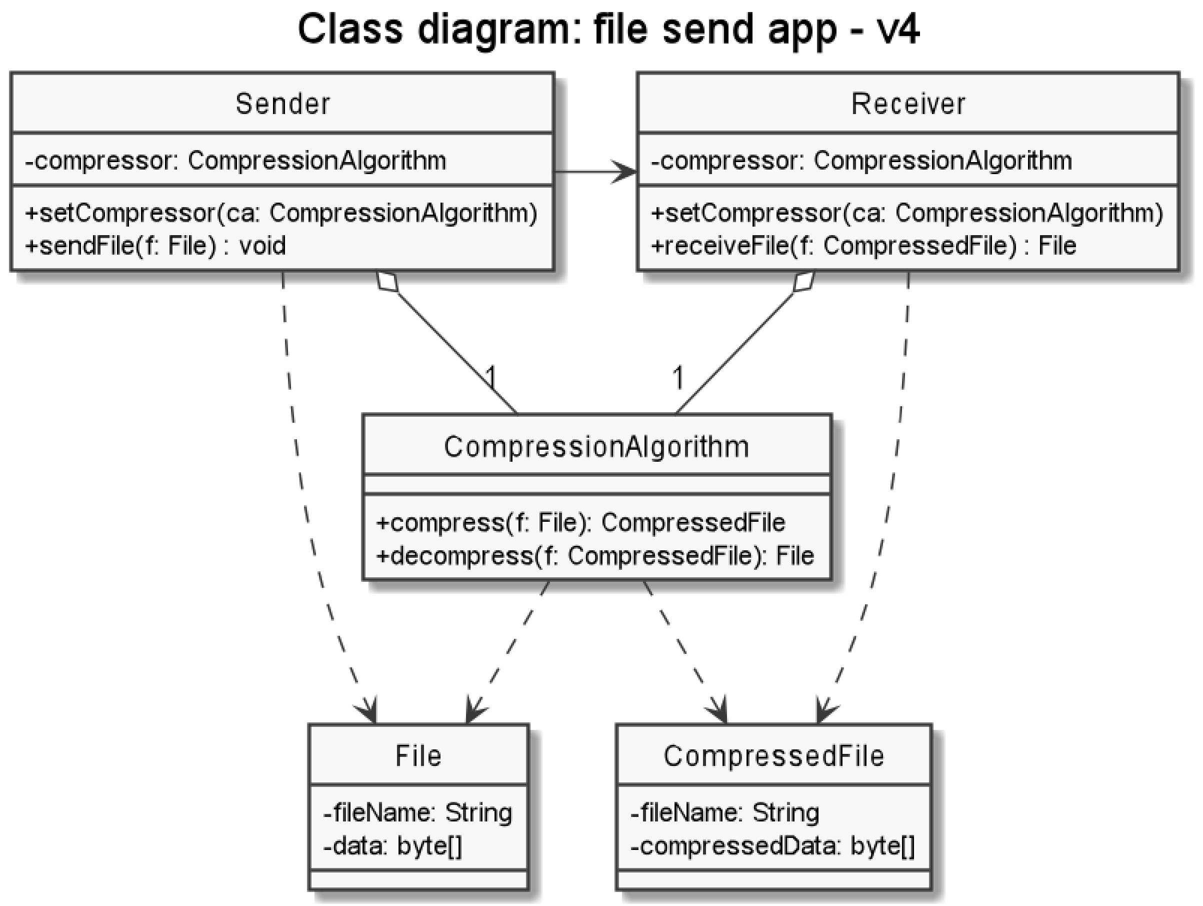
- Should be an app able to send files
- Should be an app able to receive files
- There should be a File class to store the data
- The system needs to be able to do a transfer
- The system needs to be able to do a transfer with compression
  - Compression should be optimized for regular files and image files



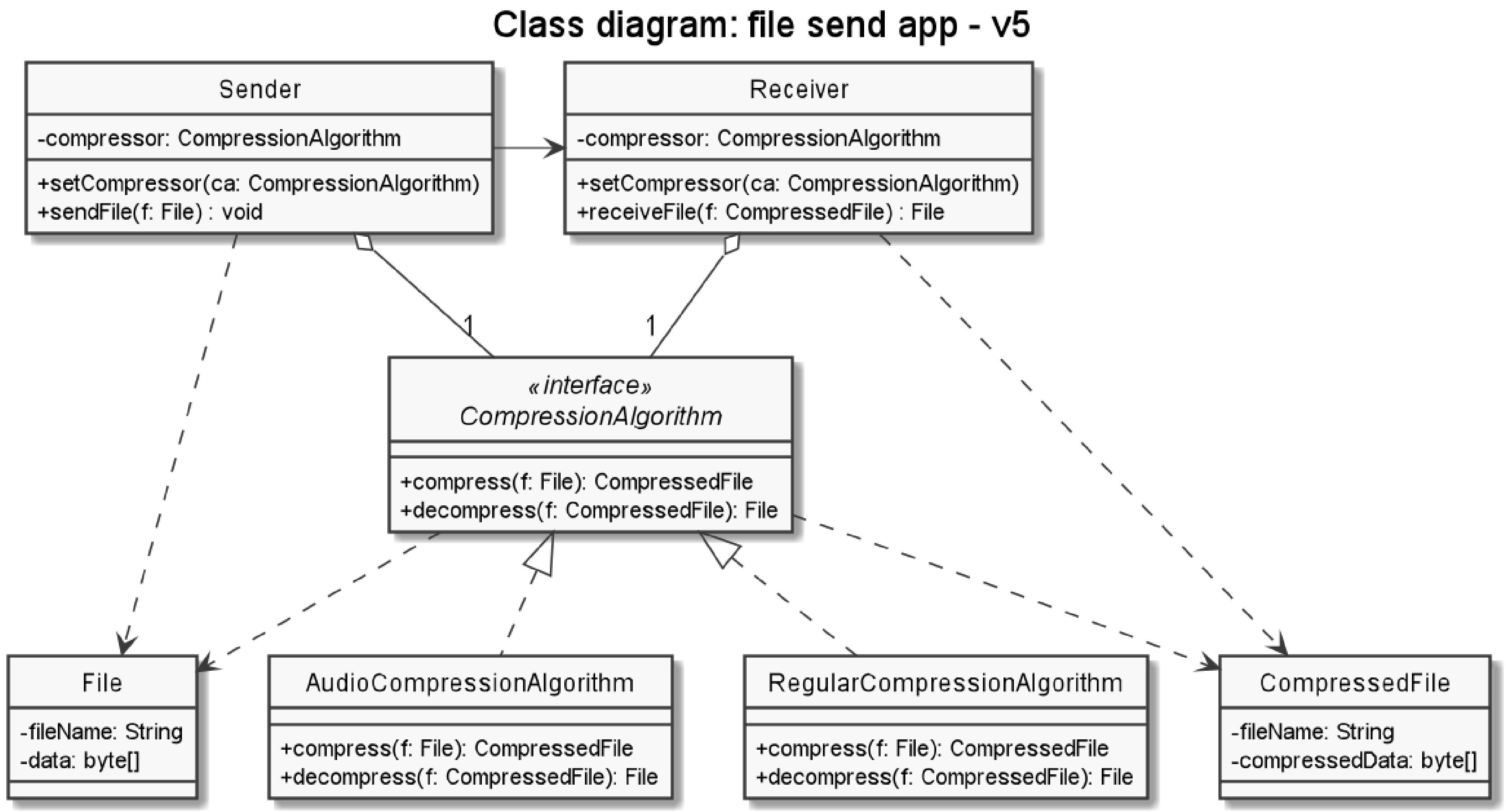
File sender application: Design – Next iteration (not so good)


Class diagram: file send app - v3
Sender
#compress(f: File): CompressedFile
+sendFile(f: File) : void
Receiver
#decompress(f: CompressedFile): File
+receiveFile(f: CompressedFile) : File
ImageSender
#compress(f: File): CompressedFile
+sendFile(f: File) : void
File
-fileName: String
-data: byte[]
CompressedFile
-fileName: String
-compressedData: byte[]
ImageReceiver
#decompress(f: CompressedFile): File
+receiveFile(f: CompressedFile) : File

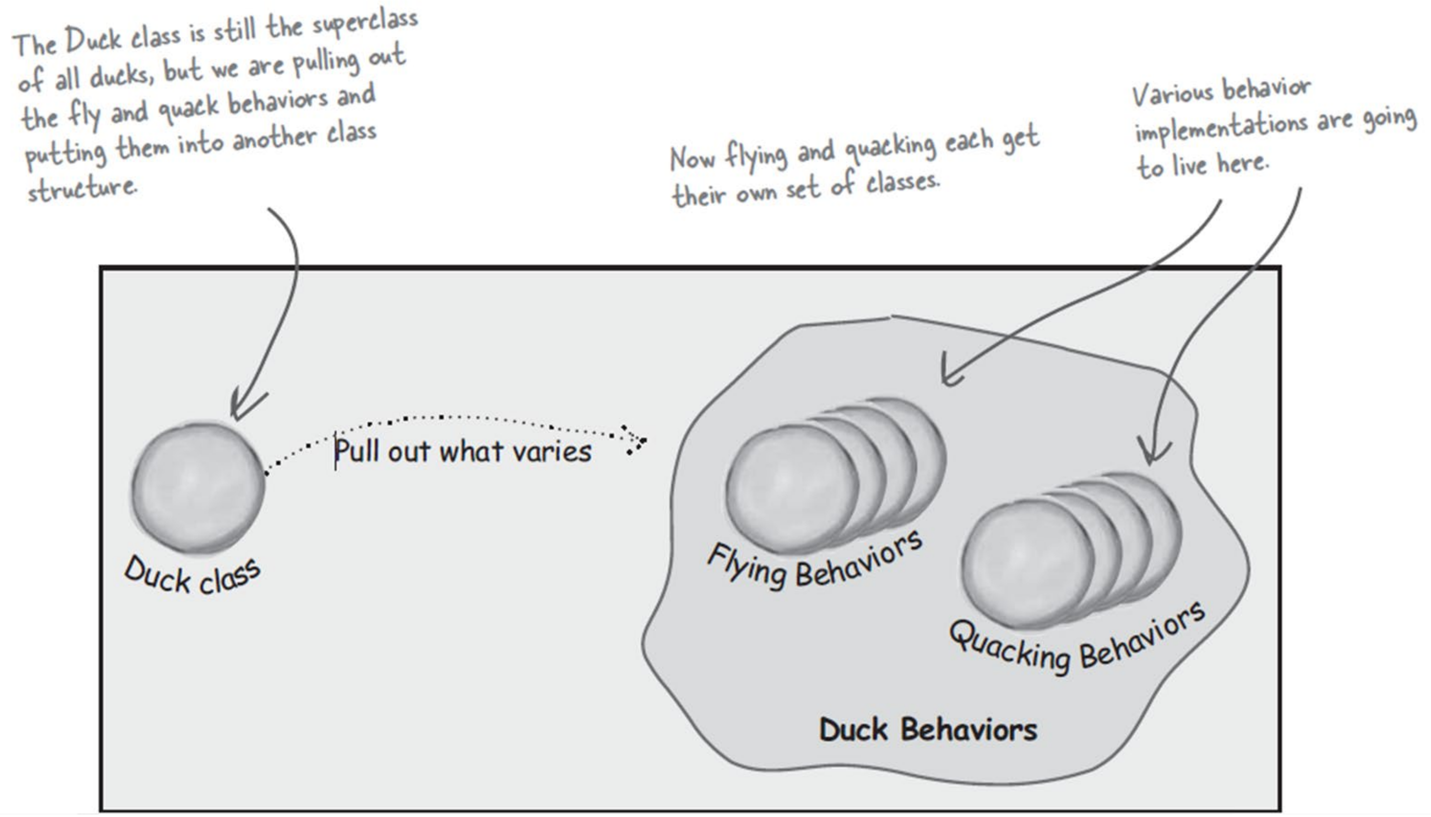
File sender application:
Design – Next iteration (better)


Class diagram: file send app - v4
Sender
-compressor: CompressionAlgorithm
+setCompressor(ca: CompressionAlgorithm)
+sendFile(f: File) : void
Receiver
-compressor: CompressionAlgorithm
+setCompressor(ca: CompressionAlgorithm)
+receiveFile(f: CompressedFile) : File
1
1
CompressionAlgorithm
+compress(f: File): CompressedFile
+decompress(f: CompressedFile): File
File
-fileName: String
-data: byte[]
CompressedFile
-fileName: String
-compressedData: byte[]

File sender application: Design – Next iteration (better)


Class diagram: file send app - v5
Sender
-compressor: CompressionAlgorithm
+setCompressor(ca: CompressionAlgorithm)
+sendFile(f: File) : void
Receiver
-compressor: CompressionAlgorithm
+setCompressor(ca: CompressionAlgorithm)
+receiveFile(f: CompressedFile) : File
1
1
«interface»
CompressionAlgorithm
+compress(f: File): CompressedFile
+decompress(f: CompressedFile): File
File
-fileName: String
-data: byte[]
AudioCompressionAlgorithm
+compress(f: File): CompressedFile
+decompress(f: CompressedFile): File
RegularCompressionAlgorithm
+compress(f: File): CompressedFile
+decompress(f: CompressedFile): File
CompressedFile
-fileName: String
-compressedData: byte[]

Strategy pattern: The duck example


The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 10

Elements of patterns

In general, a pattern has four essential elements:

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern. It explains the problem and its context.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation.
- The **consequences** are the results and trade-offs of applying the pattern.

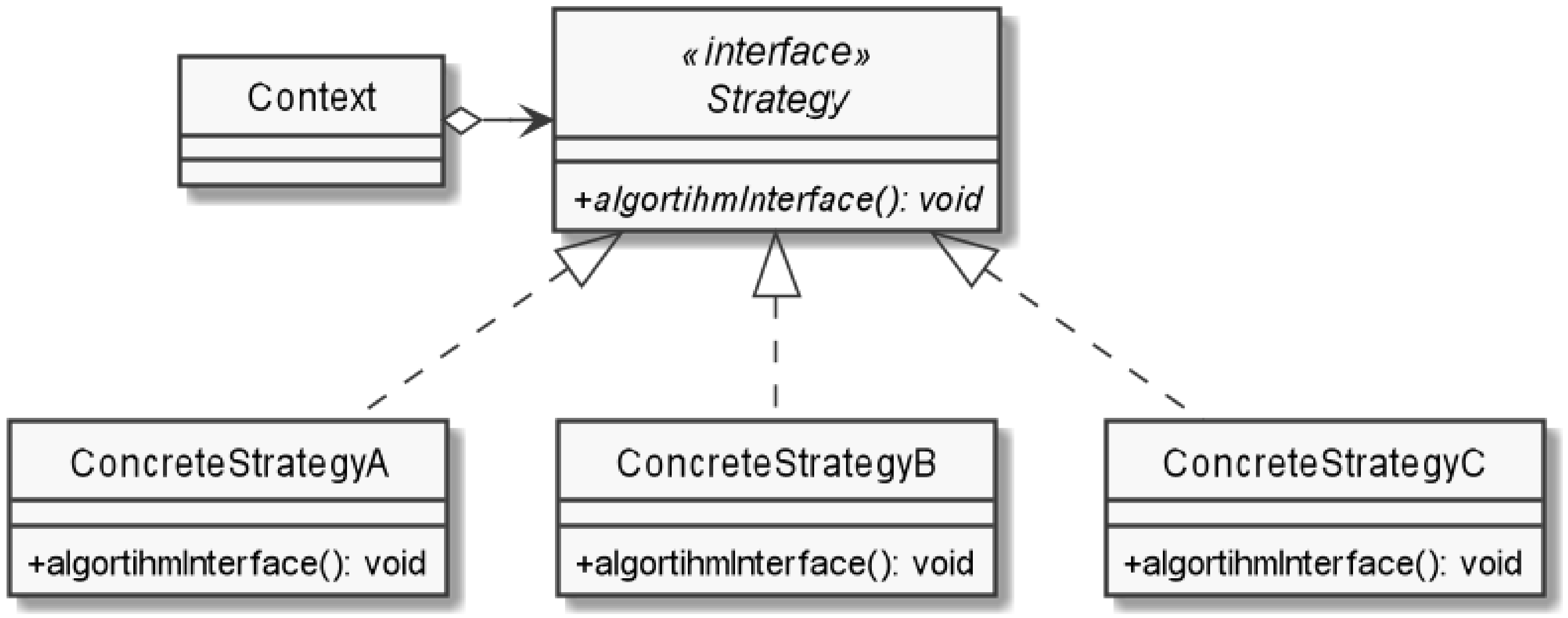
Gemma et al.(1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Documenting patterns
(format by Gemma, Helm, Johnson and Vlissides)

Pattern name	The name of the pattern
Intent	Purpose of the pattern
Also known as	Any other names by which the pattern is known.
Motivation	Illustrates how pattern solves a particular problem.
Applicability	Explains where pattern is and is not applicable.
Structure	This is a description of the relationships.
Participants	Classes and objects and responsibilities in design.
Collaborations	How classes and objects work together.
Consequences	Benefits and drawbacks of pattern.
Implementation	Trade offs, design decisions etc.
Sample Code	Code illustrating how to implement the pattern
Known uses	How the pattern has been used in the past.
Related patterns	Closely related design patterns are listed here.

Gemma et al.(1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Strategy Pattern: The Essence


Context
«interface»
Strategy
+algortihmInterface(): void
ConcreteStrategyA
+algortihmInterface(): void
ConcreteStrategyB
+algortihmInterface(): void
ConcreteStrategyC
+algortihmInterface(): void

Gemma et. al (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. p. 316

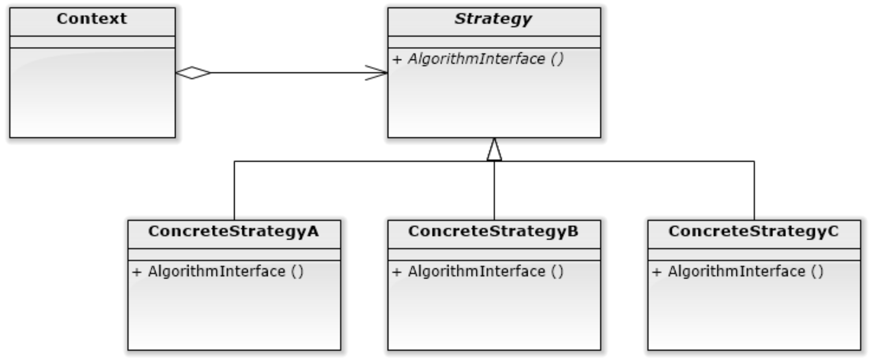
Exercise: could fill in this template?

- Pattern name
- Problem
- Solution
- Consequences

- Remember:
  - The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
  - The **problem** describes when to apply the pattern. It explains the problem and its context.
  - The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation.
  - The **consequences** are the results and trade-offs of applying the pattern.

Design pattern template filled in

- Pattern name
  - Strategy (also called Policy)
- Problem
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from clients that use it.
- Solution



Consequences

The Strategy pattern has the following benefits and drawbacks:

1. *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.
2. *An alternative to subclassing.*
   Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this 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. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. *Strategies eliminate conditional statements.*
   The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classeseliminates these conditional statements.

Gemma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 352

Design Objectives & Principles

Design Objectives & Principles

Design Objectives

- General properties of good designs that can guide you as you create your own designs

Design Principles

- Guidelines for decomposing a system’s required functionality and behavior into modules

Design Objectives

- Low Coupling, High Cohesion
  - Coupling is the amount of connections between one object and another…
- Information Hiding & Encapsulation
  - Encapsulation is hiding the internals of an object.
  - Through information hiding you can change your internal implementation without affecting other classes.
- Abstraction (i.e. generality)
- Low Redundancy (don’t repeat yourself)
  - Redundancy occurs when you have to copy/paste code in your program
- Modularity (don’t make methods to long)
- Simplicity (don’t make things to complex)
  - Complexity occurs when there are many if/then, switch, for or while statements in a method

Design Principle:
Encapsulate what varies

Identify the aspects of your application that vary and separate them from what stays the same.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 9

Design Principle:
Program to interfaces, not implementations

Program to an interface, not an implementation.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 1

Design Principle:
Favor composition over inheritance

Favor composition over inheritance.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 23

Reading

- Chapter 1 (Welcome to Design Patterns) of Head First Design Patterns

Session overview

- What are patterns
- The Strategy pattern
- Design Objectives & Design Principles

Reading

For this lesson you had to read:

- xxx

For tomorrow please read:

- xxx

Additional reading (not required):

- xxx