Decorator

DEVELOPMENTOR

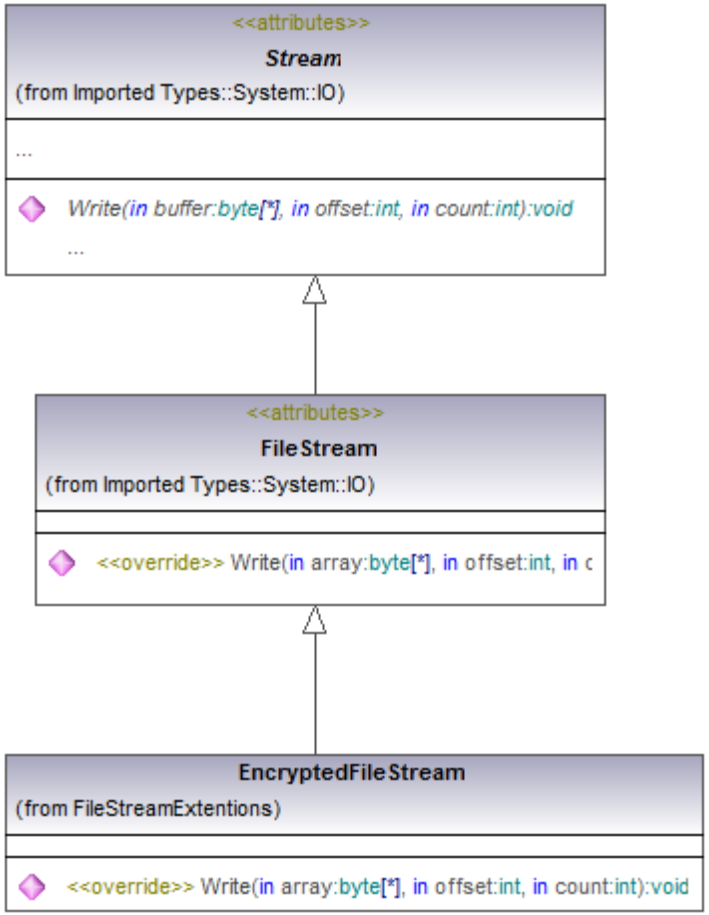
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

The desire to extend or modify behaviour

- To extend or modify a type's behaviour we often think inheritance
  - Design time
- However
  - Not always practical, can lead to type explosion.
  - What if base type is sealed.
  - What if you have no control over object creation
- The Decorator pattern may allow us to overcome these issues by extending at runtime.

Encrypted FileStream

- When asked to build an encrypted file stream type we may very well opt for something like this



How many combinations…

- Using inheritance to aggregate behaviours can soon break down
  - FileStream → Encrypted Stream
  - FileStream → EncryptedStream → EncryptedAndSignedStream
  - FileStream → EncryptedStream → EncryptedAndSigned → EncryptedAndSignedAndDuplicatedStream
  - But what if you only want Signed stream or a DuplicatedSignedStream.

All Singing and Dancing FileStream

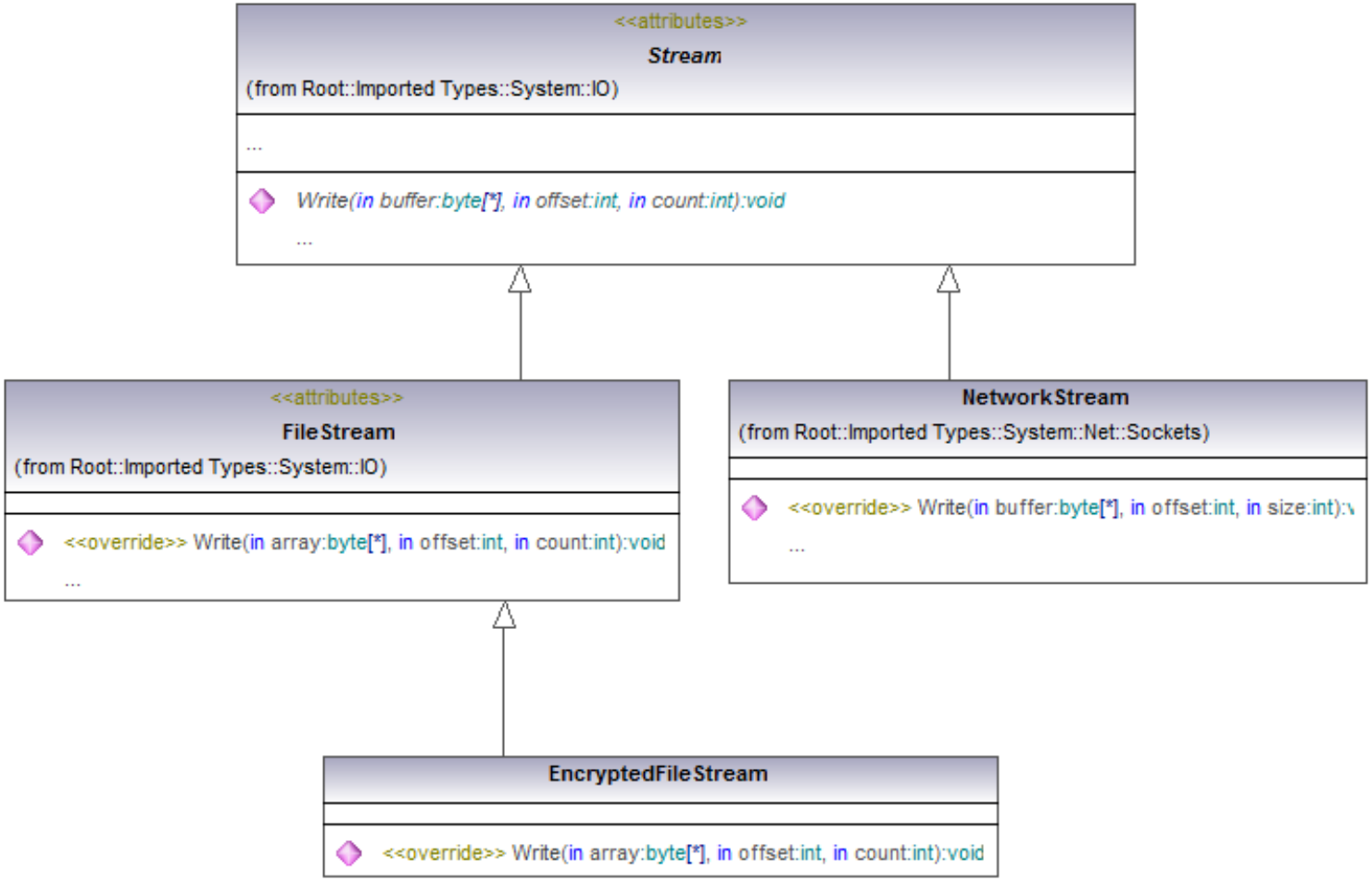
- How about adding feature fields?
- Now add support for compressed stream, what Pattern principal are we breaking?

```
public class FancyFileStream : FileStream  {
  private bool SignStream;
  private bool EncryptStream;
  private FileStream duplicateStream;

  public override void Write(byte[] array, int offset, int count) {
    if (SignStream) {
      // Do Signing stuff
    }
    if (EncryptStream) {
       // Do Encrypting stuff
    }
    if (duplicateStream != null) {
       // Do Duplicate stuff
    }
    base.Write(array, offset, count);
 }
```

Open for extension closed for modification

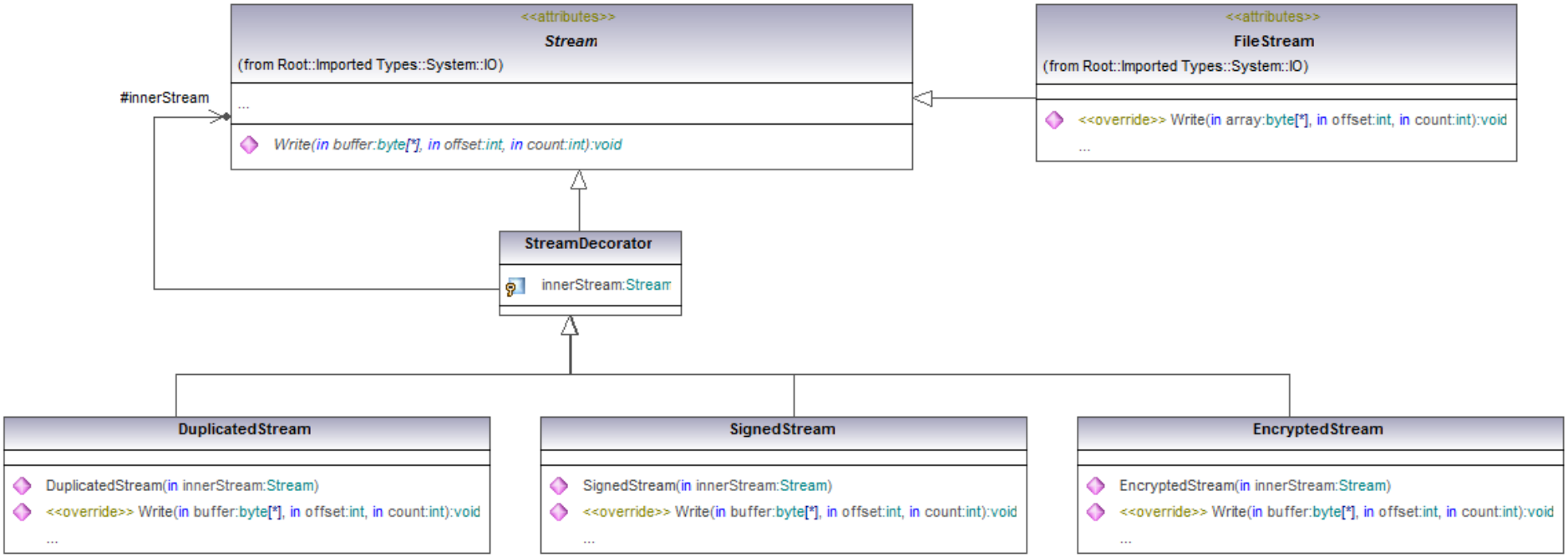
- The field approach has two obvious disadvantages
  - Adding more features means modifying existing code
  - Its not easy to add similar functionality to other types of stream



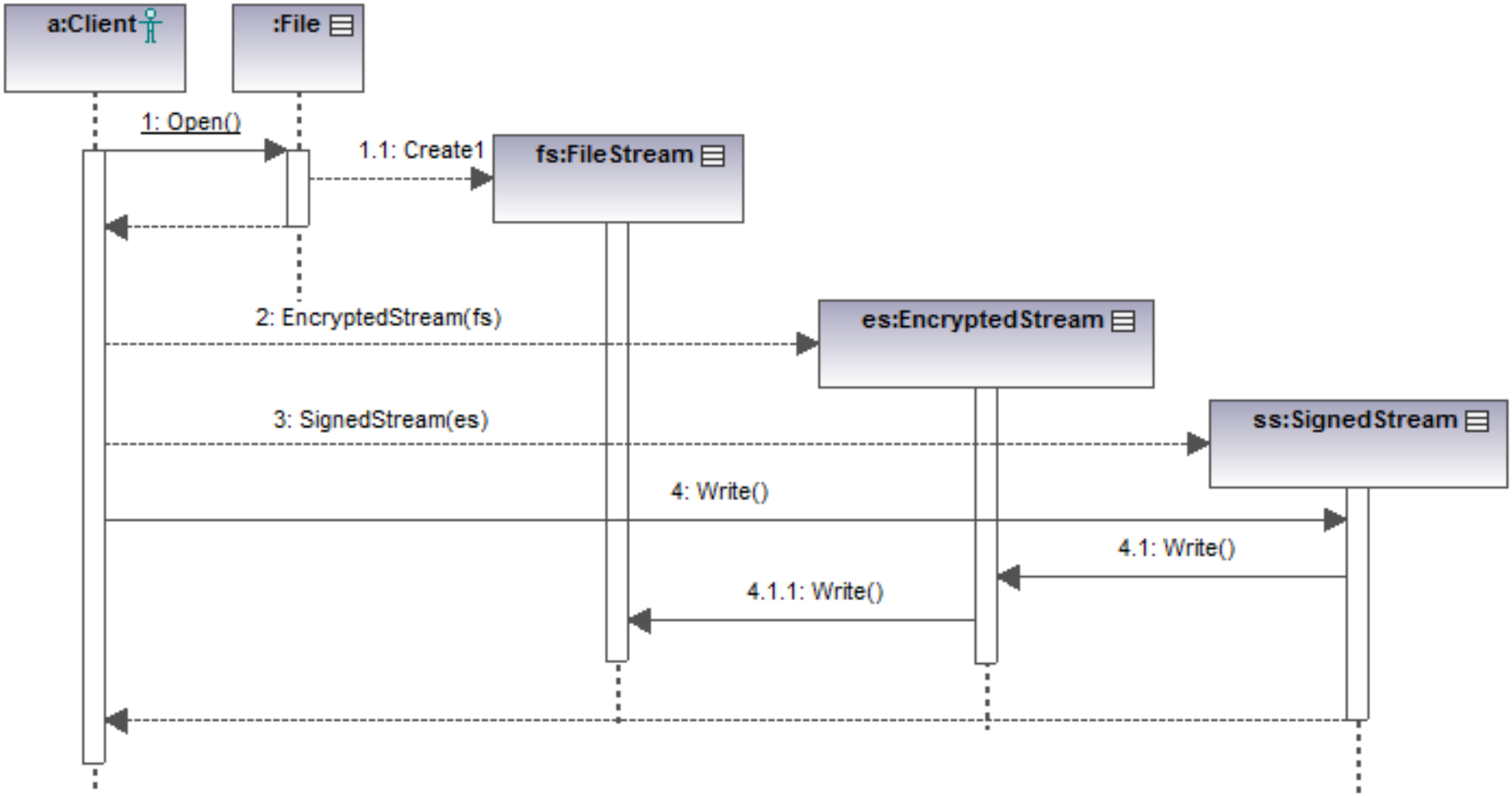
Extensible streaming

- Step back
  - In the case of the FancyStreaming type we literally wish to chain the various write operations together
    - Take buffer sum bytes
    - Take buffer and write to stream one
    - Take buffer and write to stream two
    - When all done write signed value and flush
  - The definition of the chain we wish to leave until runtime
- Solution
    - Make each piece in the chain be a “kind of” Stream
    - Make each piece in the chain hold a reference to the previous “kind of” stream object
    - Make the client work against stream class

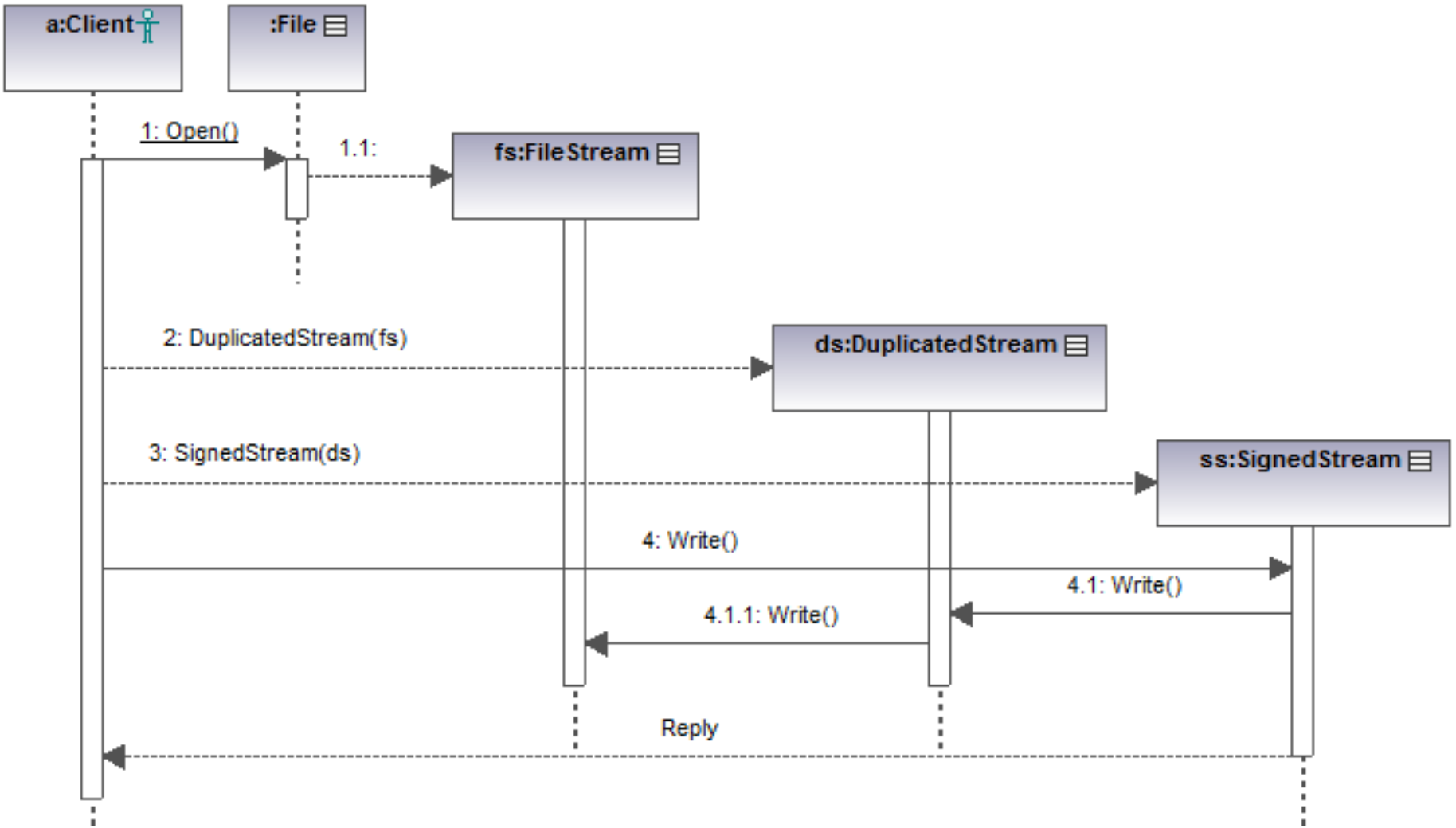
Extensible Streaming



Encrypted and Signed File Stream

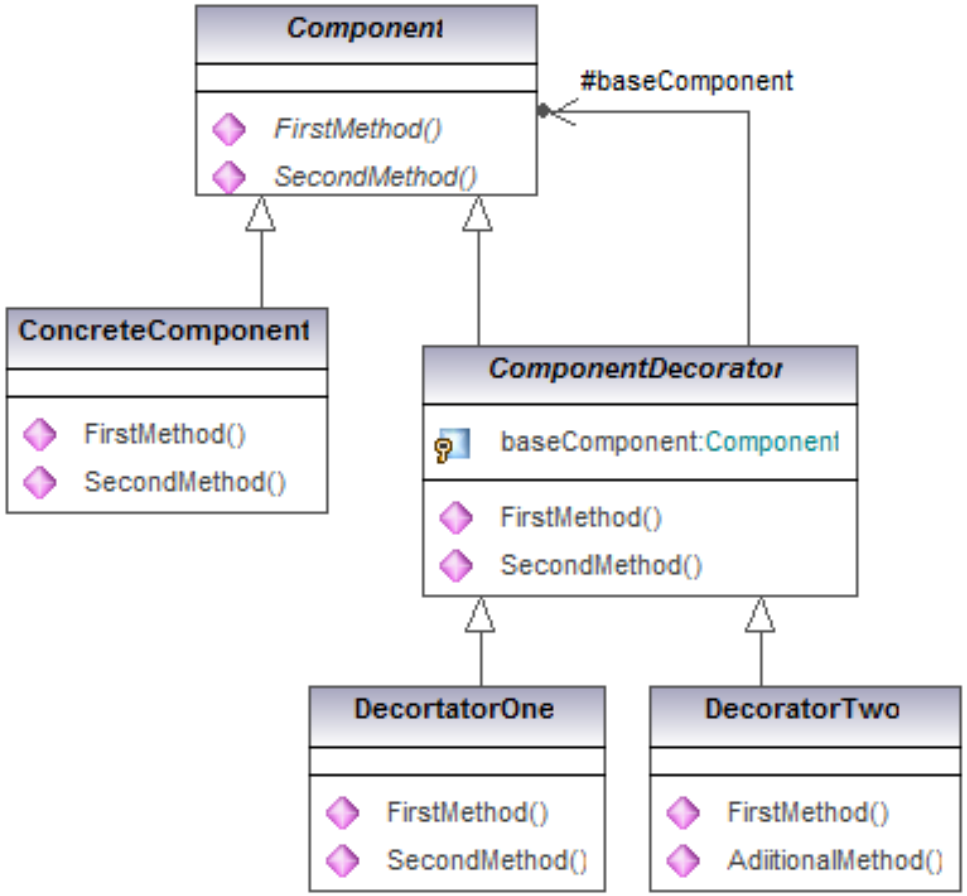


Duplicated and Signed File Stream



The Decorator Pattern

- Attaches additional responsibilities to an object dynamically
- More scalable than type inheritance



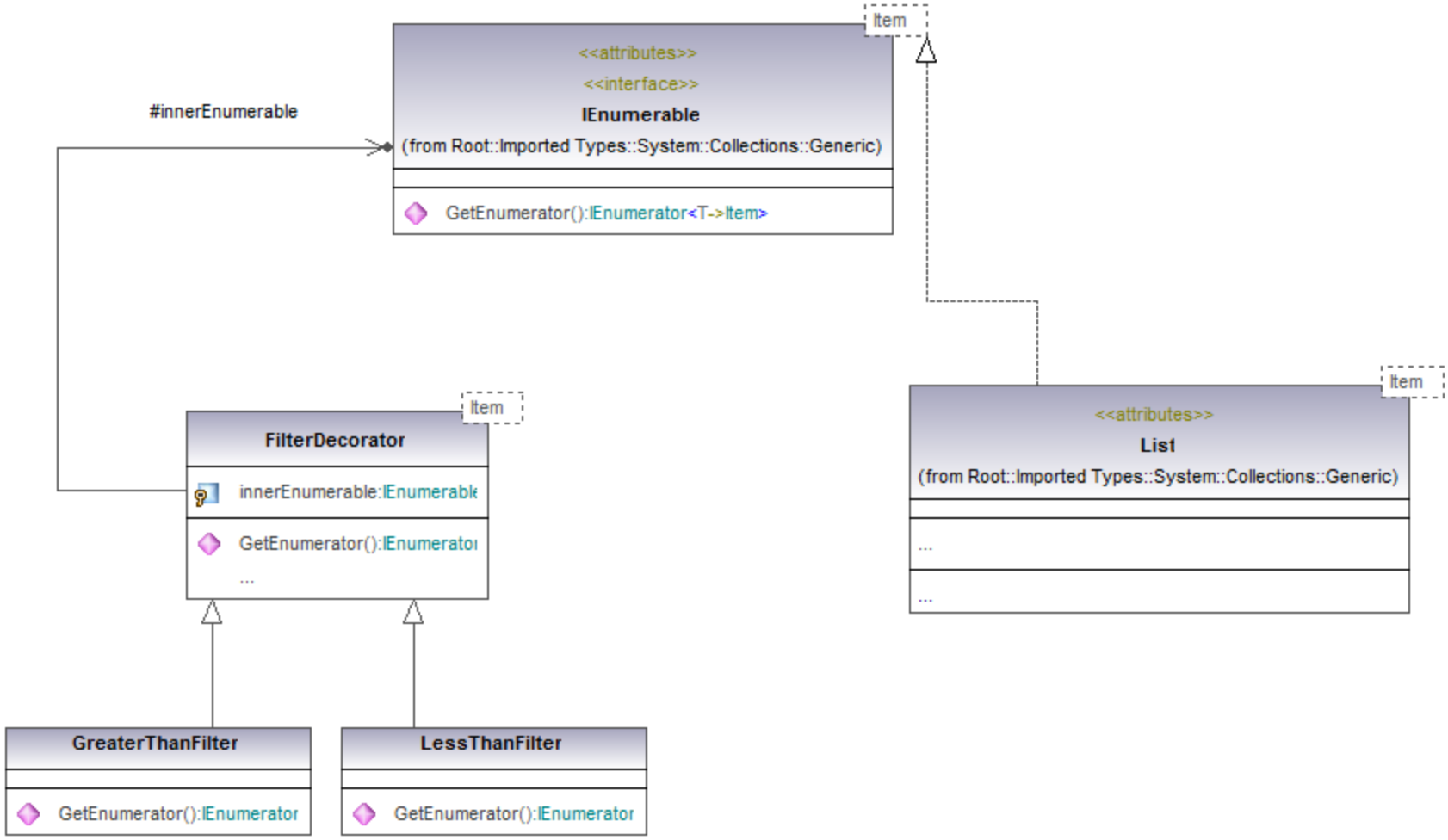
Adding Decorator responsibilities

- With type inheritance we can add new responsibilities to our derived type
- The Decorator pattern can take this a step further allowing us to dynamically add responsibilities to an object
- Requirement
  - I wish to know
    - How many bytes have been transferred through a stream
    - What is the bandwidth of the stream
- Solution
  - Create a decorator that times the Read/Write and counts number of bytes transferred
  - Add additional method to the decorator to obtain the statistics

IEnumerable Decorators

- .NET has a standard iterator types, programming languages and parts framework leverage.
  - C# foreach keyword
  - Collections
  - Data binding in ASP.NET, Windows Forms, WPF
- When returning a series of items its often desirable to filter the series.
  - Return all items created between 1/1/2007 and 30/12/2007
- Create a series of IEnumerable Decorators that can be combined to produce the desired overall filter function.

IEnumerable Filter Decorator



Decorators in the framework

- NegotiateStream class
  - Adds session encryption and client credentials to the stream
- XmlValidatingReader
  - Validates a XML stream
- System.IO.BufferedStream
  - Implements buffered stream functionality for any type of stream
- BindingLists
  - Enhances an existing IList to provide events when the list is modified producing an observable list

Summary

- If inheritance is causing type explosion consider the Decorator pattern
- If you need to extend a type past the point of creation consider using the decorator pattern
- The Decorator pattern accomplishes these goals without modifying existing working code
  - "Closed for modification Open for Extension..."
- Client must be coded to interface