efficiency code re-use

- Famous sentences about optimization:
  - "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"
  - "In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"

But...

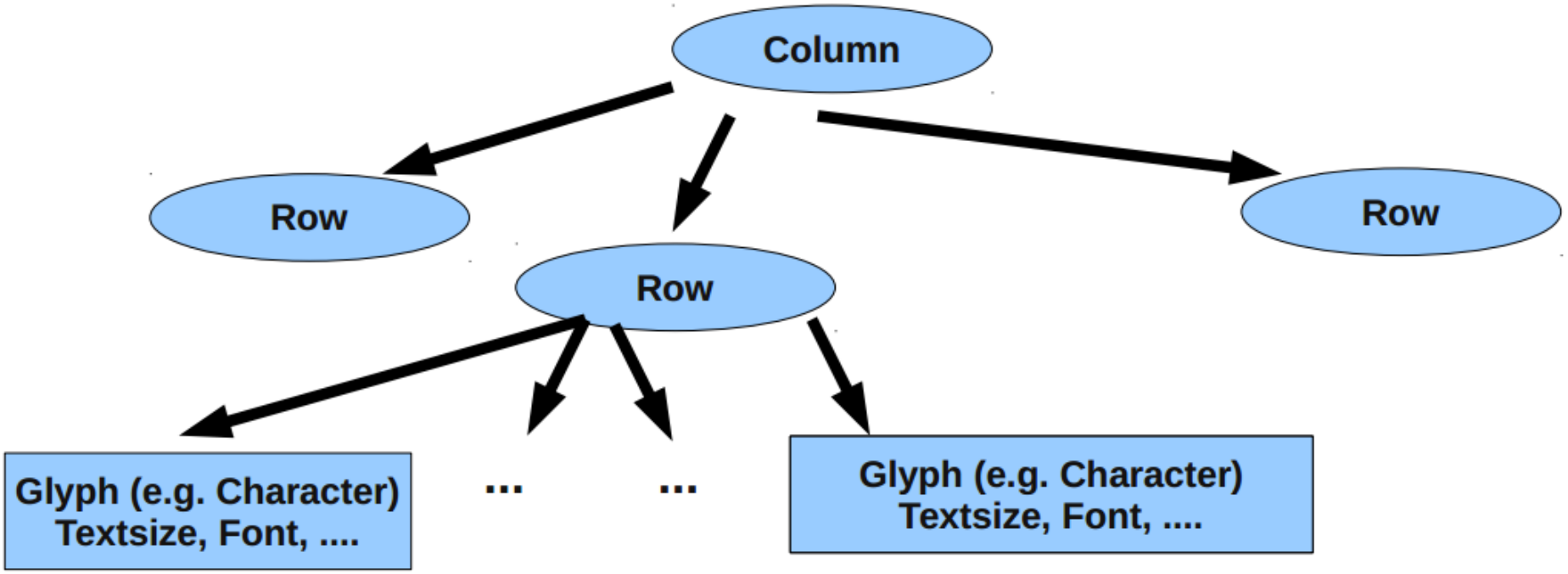
- Some performance relevant Design Decisions e.g.
  - Data-Centric approach (tends to be easier to optimize, but harder to start for newcomers)
    vs.
    strict OO design with data hidden in objects, and all algorithms working on the data in the object interface (better encapsulation --> easier for newcomers),
  - Supported Platforms,
    need to be done early on.
- However, for big chunks of the code performance usually is not critical, so a defensive programming style is better.

Design Patterns

Lightweight Pattern

- See “Design Patterns” by Gamma et al. 37th printing, p. 195, 207
- Sometimes one wants to have the full flexibility of objects, but using full objects would be prohibitive to use, e.g. glyphs in an editor, that are simply too numerous.
  - Idea of lightweight Pattern:
    - If objects there are so many objects, many may have similar values for their members, e.g. many glyphs in an editor will have the same character, text size, font etc.
    - Now instead of creating the same objects over and over, return just a reference to some element of a flyweight pool.

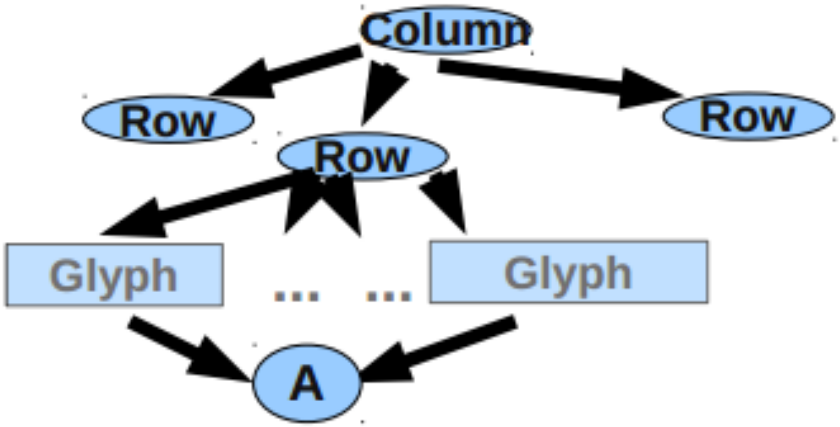
Before Usage of Flyweight



Creates the Need to Manage

- If someone changes e.g. the font type, the function has to check, if there is a flyweight with appropriate properties or create new flyweight in the pool.

- Some properties of an object may not be storable in the flyweight, e.g. the position of a glyph. This has to be managed by the object handling the reference, e.g. the row, which should give position information to the glyph, when it is drawn.



Proxy

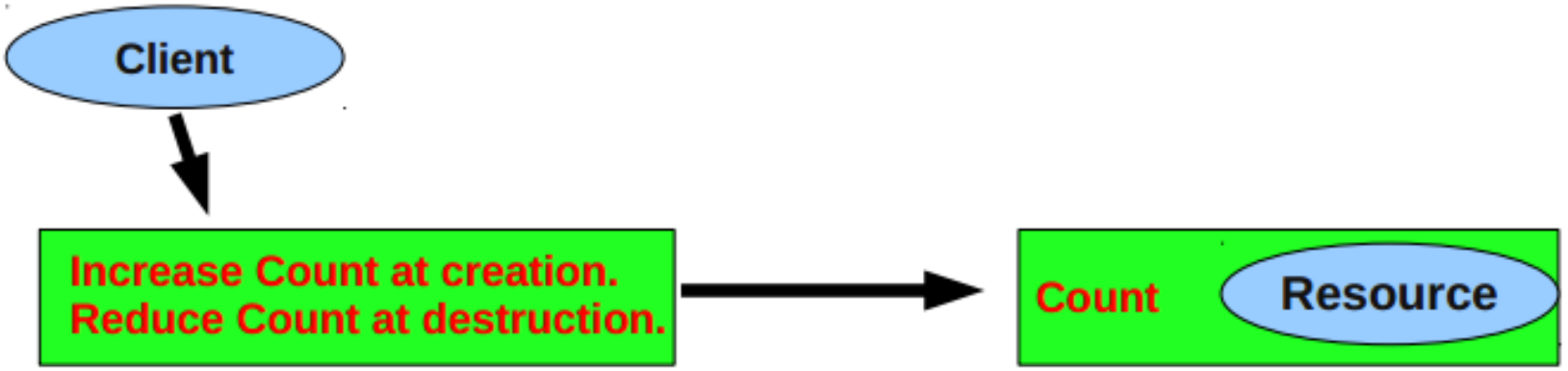
- Also known as surrogate, because
  - it stands in place of something.
    - e.g. in a text editor instead of a picture, that takes uses a lot of memory, there maybe just a box. If you actually want to see the picture it is loaded on demand.
    - e.g. instead of loading data from a file, there maybe a Proxy, that caches recently used values and returns them, when asked. Only when the value is new, it loads from the file.

Lazy Copy

- If you copy a large object, e.g. a long string, and you are not sure you ever modify it, you may just use a proxy with a pointer to the original string.
  - That one of course needs to know, that its value is referenced by some other object... → Reference Counting, shared_ptr ...

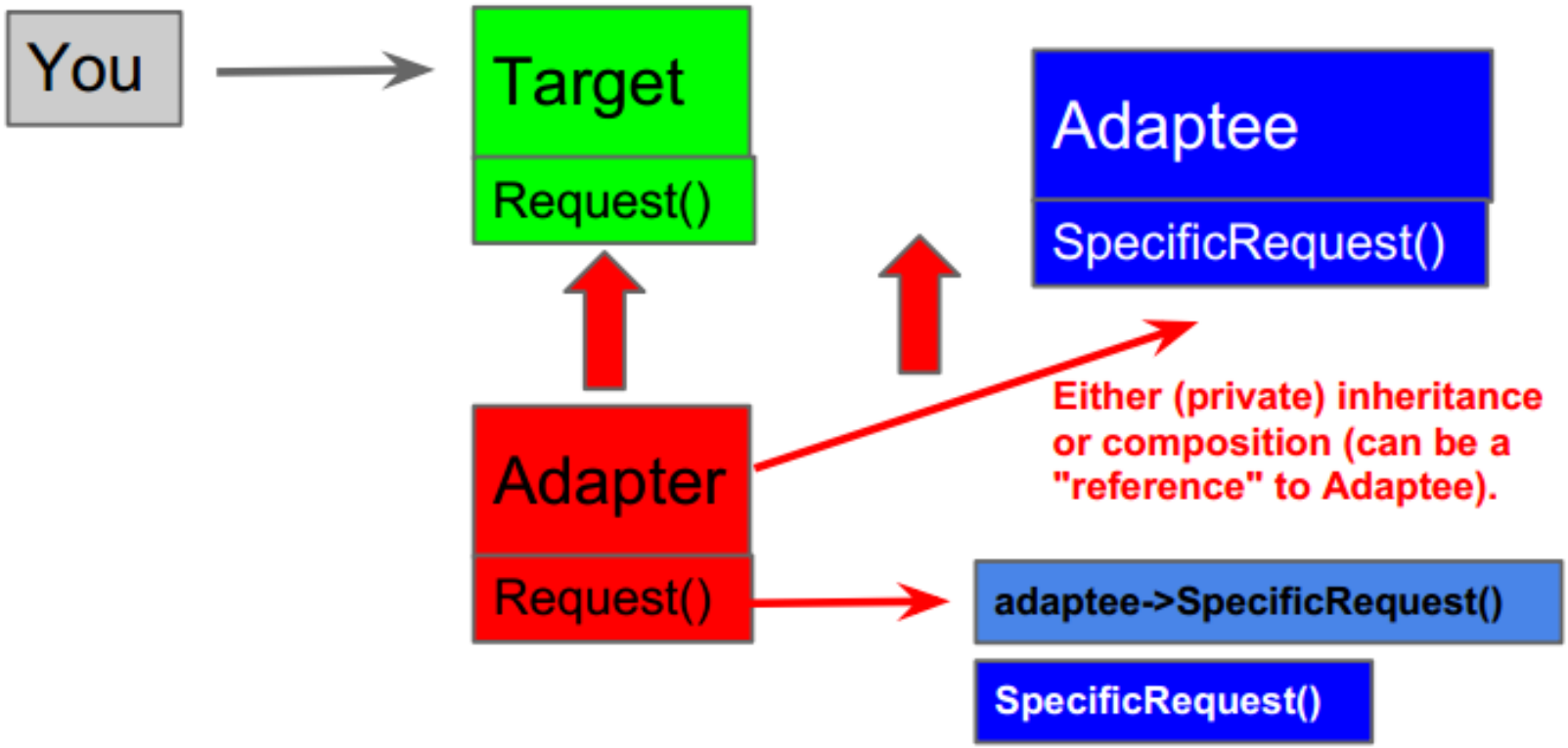
Resource Managing Objects

- In general it is useful to have resources like memory, database accesses etc. managed by objects, that make sure, that the resource is freed, after it isn't used anymore.



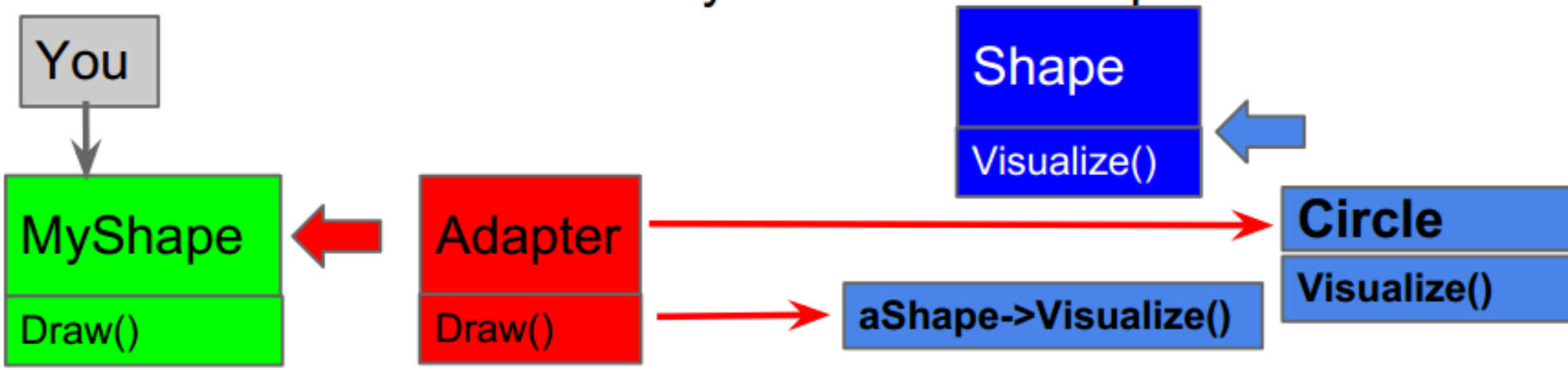
Adapter/Wrapper

- Interface an existing class with an Interface, that doesn't match your requirements.



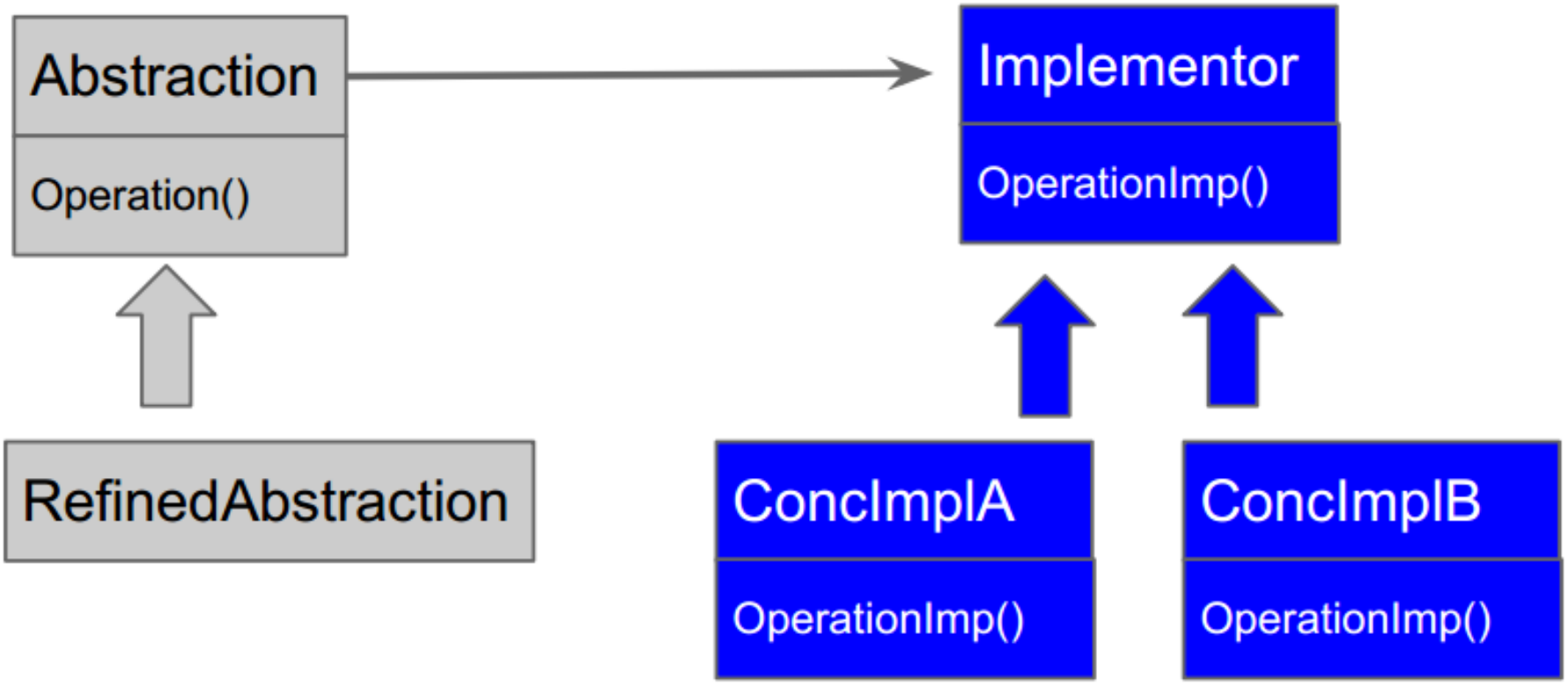
Consequences

- Using (private) inheritance ("Class Adapter").
  - Adapts to a concrete class.
  - Possibility to override some behaviour.
  - Introduces ONE object, no additional pointer indirection needed.
- Using composition ("Object Adapter")
  - Easier to use various subclasses of Adaptee.
  - Less trivial to modify behavour of Adaptee



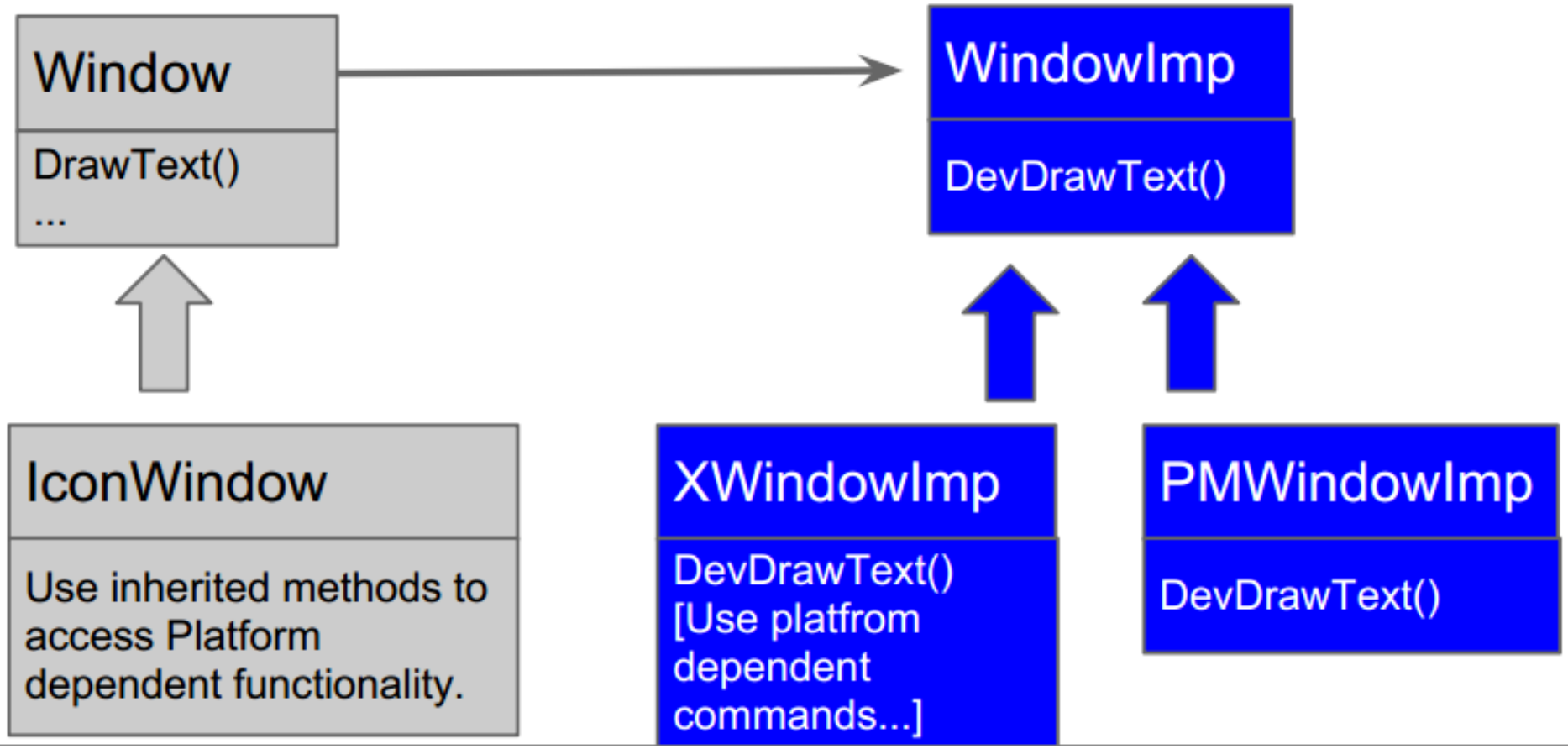
Bridge

- Decoupling of Abstraction and Implementation.



Example

- Possibility to use factory method to decide at run time on specific implementation.



Consequences

- For supporting a new type, you just need to write another ConcreteImplementor.
- You can introduce a new ConcreteImplementor without recompiling Abstraction.
- Improved extensibility, as you can work on the two hierarchies separately.
- Better shielding of implementation details from clients.

Bridge's Relation to Adapter

- Adapter is usually used in cases, when you use an existing library to adapt the interface, while Bridge is used up-front to separate abstraction from implementation.

  Maybe sometimes a policy approach can be used instead of a bridge, if you need flexibility only at compile time rather than run-time, e.g. in the example you instantiate the abstract hierarchy with template parameters of platform dependent objects.

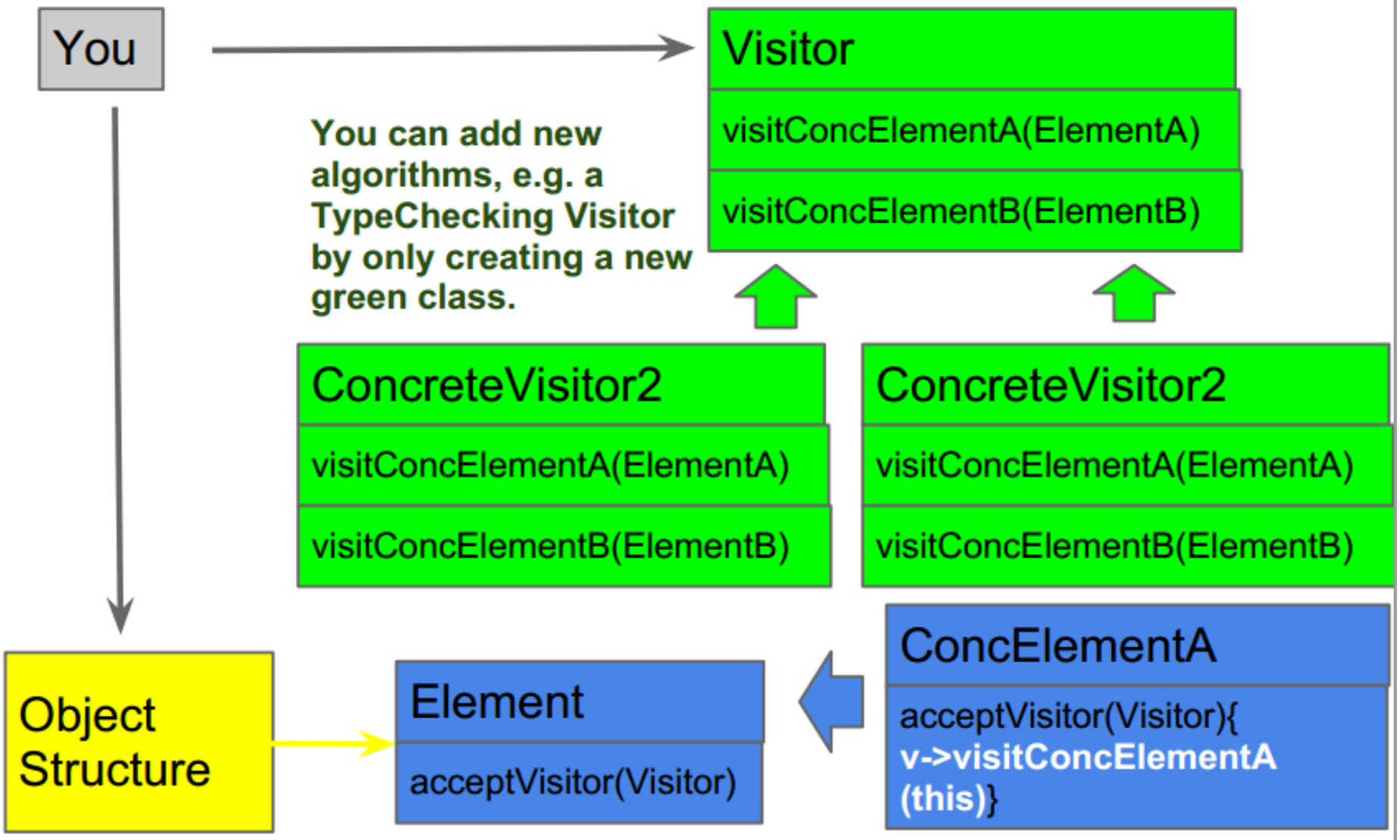
Visitor

- Assume you have an array of pointers to a Base class, e.g. "Node" (some statement handled by a compiler).
- Assume there are really specific Nodes, e.g. "Assignment Node", "VariableRefNode",... that are too different to handle some functionality the same way.
  --> First Ansatz: Common Base Class with virtual function, overwrite the function in every derived class.. but:
  - Adding a new operation means, you have to change every Derived** class (assuming common Base class function doesn't suffice.)
  - Algorithm is scattered over many files, which makes it hard to keep complete overview.

**

- Big projects often are divided in many packages with different access rules.
  - --> changing lots of classes in potentially different packages can annoy a lot of people.

Keep Control Over New Algorithm in One File



Consequences

- Visiting makes **adding new operations** easy.
- It **accumulates related behaviour** in one file.
- Visitors can **accumulate state**/gather information during visiting.

- **Adding new ConcreteElement** classes is hard. If one additional concrete element class is added, all visitors have to be updated, e.g. in this case the information is spread across many files.
- **Visitors may break encapsulation**, as they may have to act intrusively on the ConcreteElements.