Software-Praktikum

SS 2021, Blatt 3: Design Patterns (Teil 1)

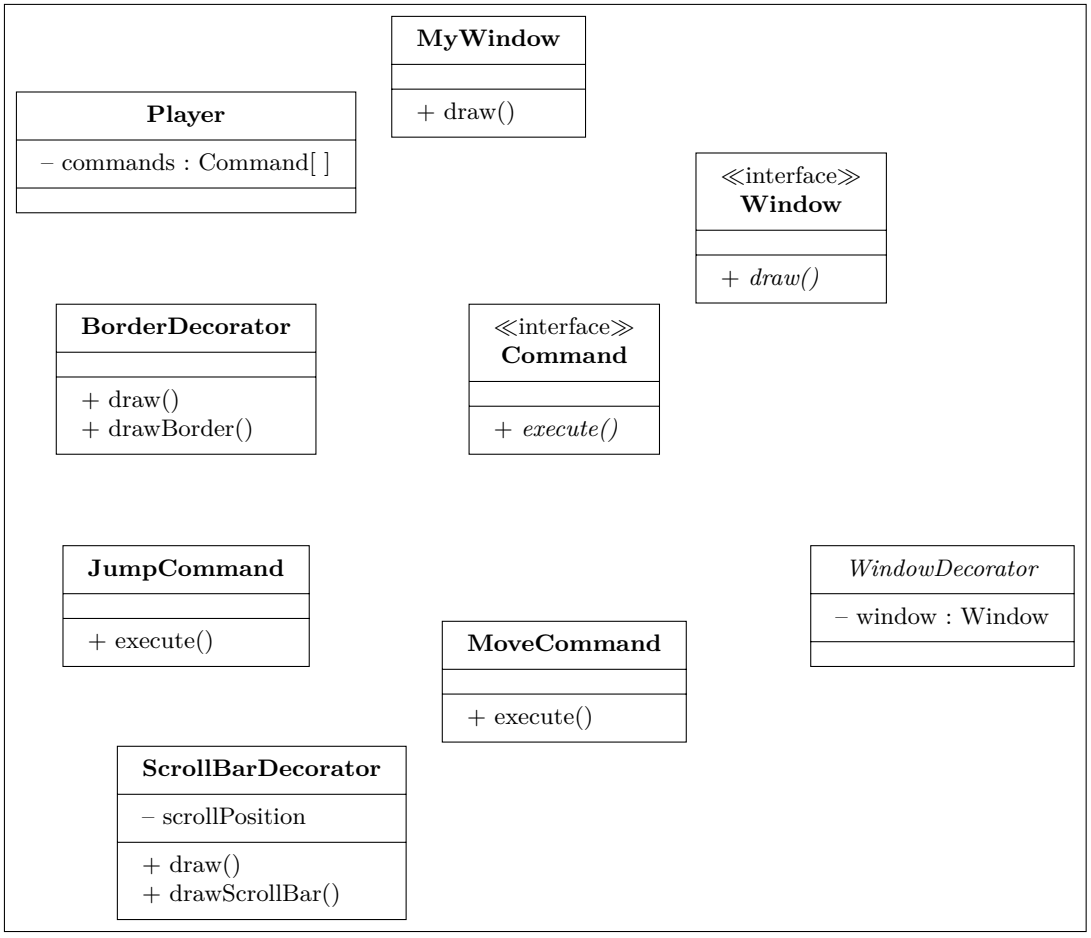
Ausgabe: 01.09.2021

Prof. Dr. Sven Apel

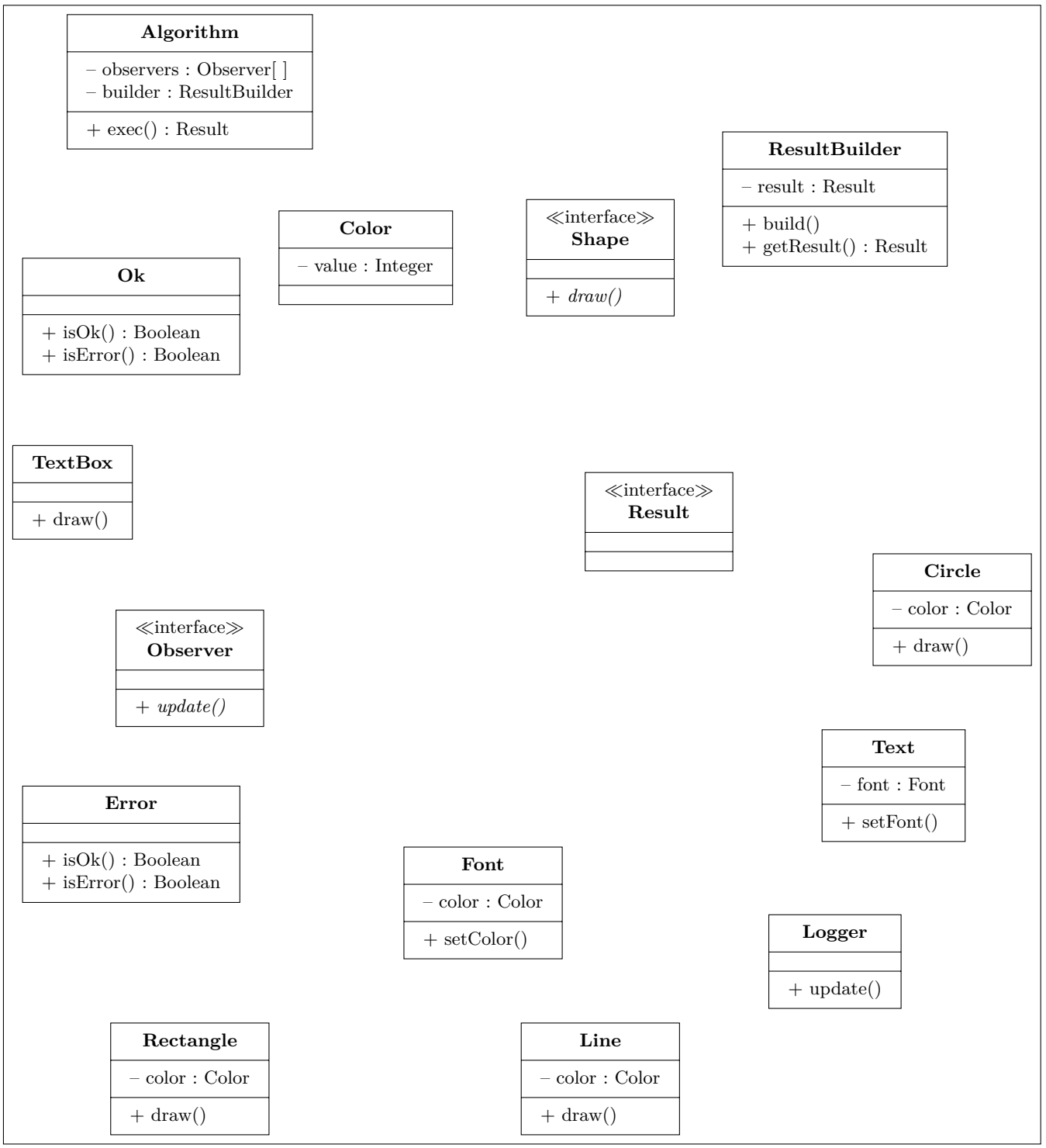
Aufgabe 1 Entwurfsmuster aus Elementen zusammensetzen

Ergänzen Sie folgende Klassendiagramme um Vererbungen, Implementierungen, Aggregationen (gegebenenfalls Kompositionen) und benennen Sie die gefundenen Entwurfsmuster.

a)

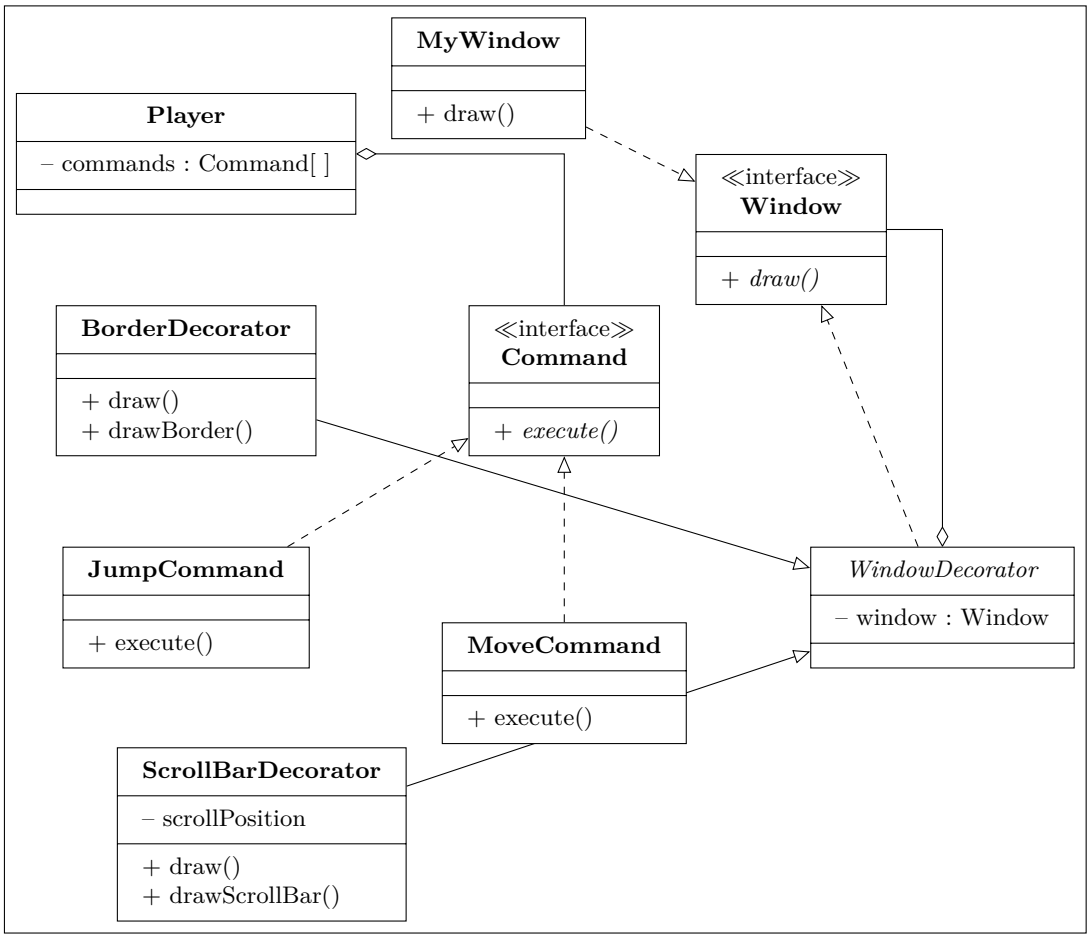


b)


Algorithm
– observers : Observer[ ]
– builder : ResultBuilder
+ exec() : Result
ResultBuilder
– result : Result
+ build()
+ getResult() : Result
Color
– value : Integer
≪interface≫
Shape
+ draw()
Ok
+ isOk() : Boolean
+ isError() : Boolean
TextBox
+ draw()
≪interface≫
Result
Circle
– color : Color
+ draw()
≪interface≫
Observer
+ update()
Text
– font : Font
+ setFont()
Error
+ isOk() : Boolean
+ isError() : Boolean
Font
– color : Color
+ setColor()
Logger
+ update()
Rectangle
– color : Color
+ draw()
Line
– color : Color
+ draw()

Lösung

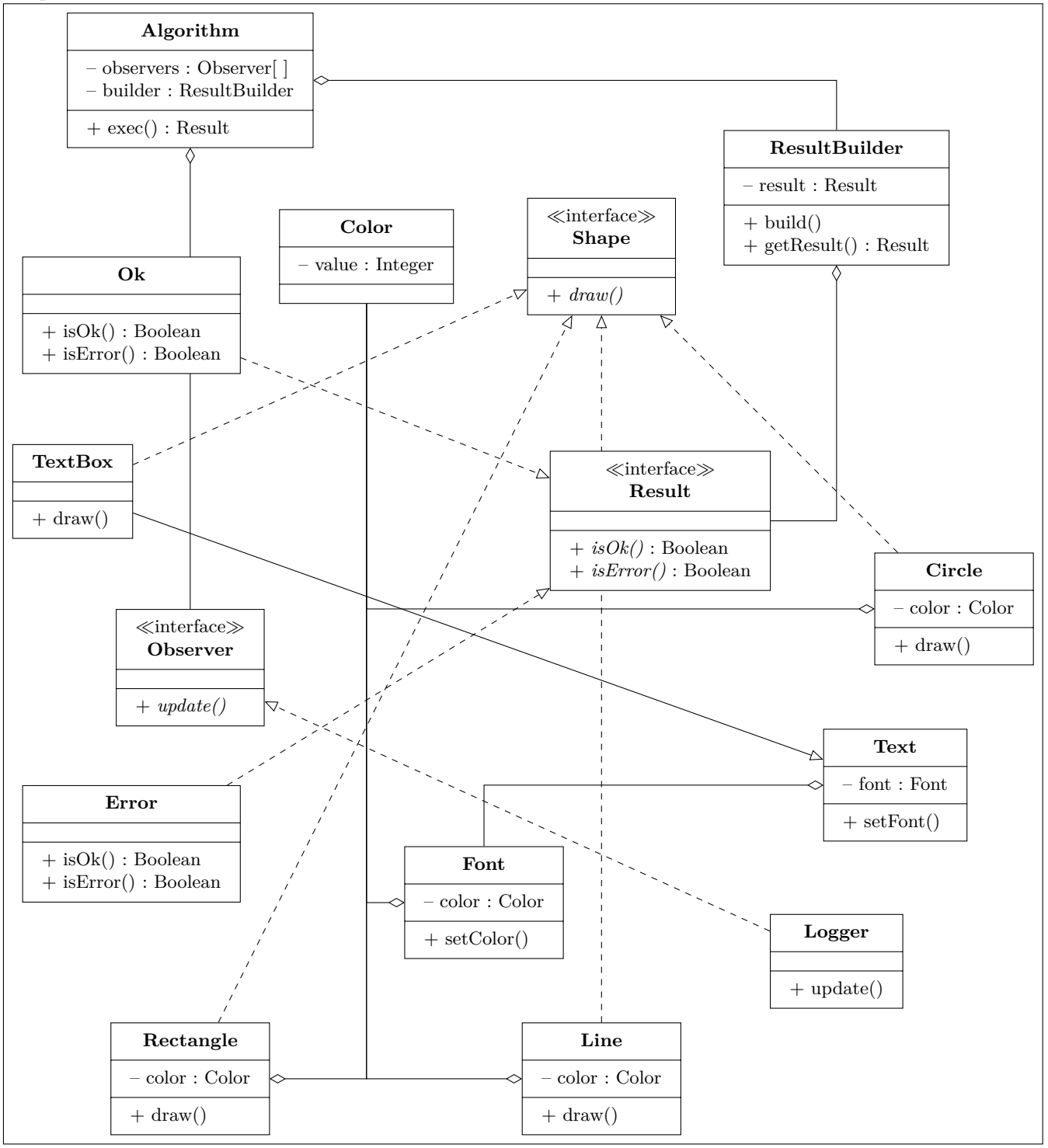
a) Decorator und Command



Das *Decorator Pattern* wird in der Vorlesung detailliert vorgestellt. Interessant ist für die Implementierung, dass wenn der Dekorierers selbst Definitionen bereitstellt (in diesem Beispiel ist `WindowDecorator` abstrakt) durch die (dekorierten) Unterklassen Methoden möglicherweise nurnoch überschrieben werden, sodass nach außen zunächst immer das Verhalten des Dekorierers sichtbar ist. Häufig wird dieses Entwurfsmuster nicht explizit eingeführt, sondern ergibt sich natürlich aus einem Objekt orientierten Entwurf.

Das *Command Pattern* ist ein Verhaltensmuster das unter einem Interface verschiedene Klassen mit verscheidener Funktionalität bereithält. Vorteile gegenüber einer naiven Implmentierung, sind die Modularisierung und Kontrolle über den Datenfluss (gegebenfalls durch an `execute()` übergebene Parameter). Listen von Commands können gespeichert werden und erst zu einem späteren Zeitpunkt ausgeführt werden.

b) Adapter, Observer und Builder

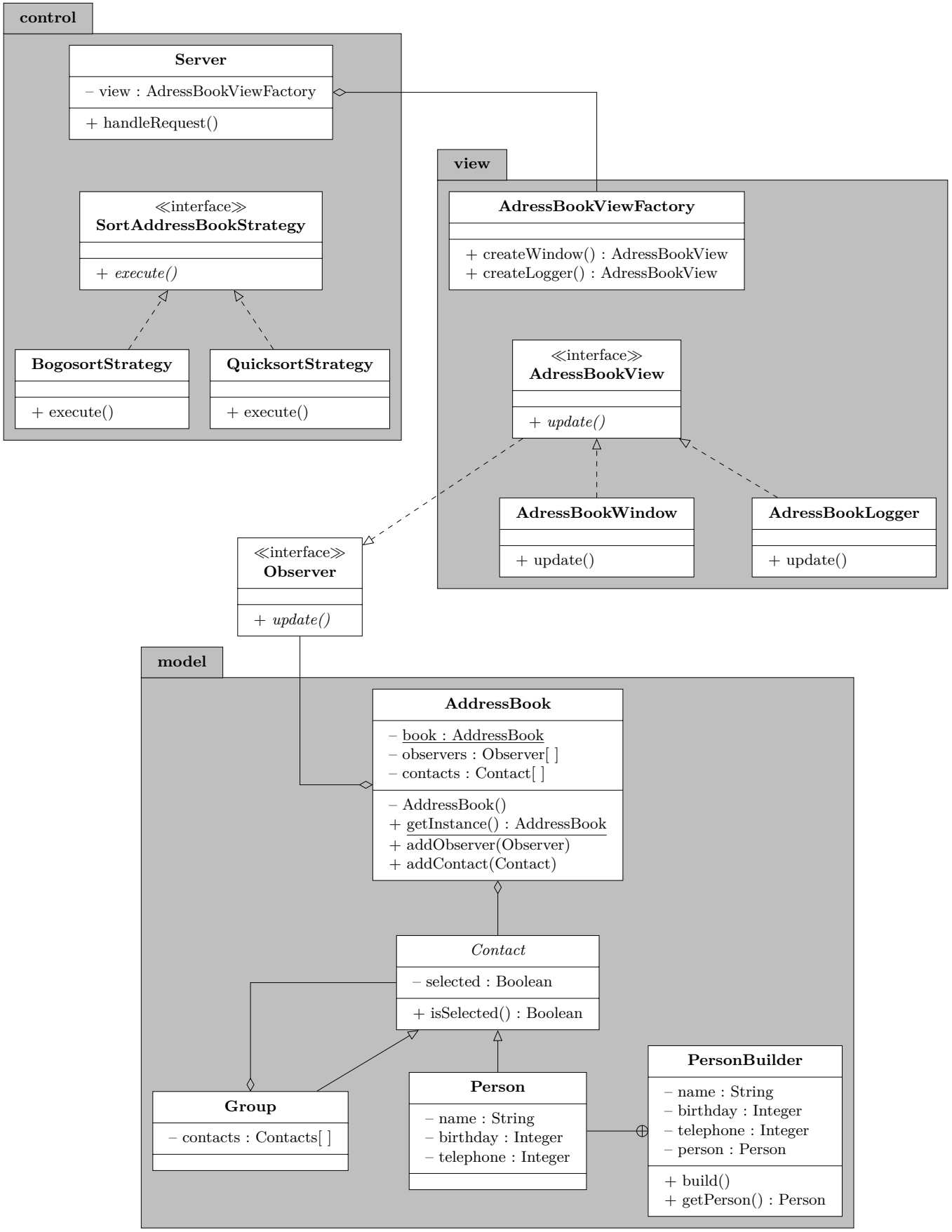


Adapter und *Observer Pattern* wurden in der Vorlesung detailliert vorgestellt. Observer können gut genutzt werden um Logger oder einfache grafische Oberflächen zu bauen.

Das *Builder Pattern* ermöglicht es dem Entwickler komplexe Abhängigkeiten bei der Erstellung einer Klasse hinter einfachen Methoden zu verstecken. Parameter werden zwischengespeichert und erst der Aufruf von `build()` erstellt das eigentliche Objekt für den Aufrufenden. Die `Result`-Klasse in dieser Aufgabe stammt aus einem realen Software-Projekt und kann zum Zurückpropagieren von Fehlern verwendet werden.

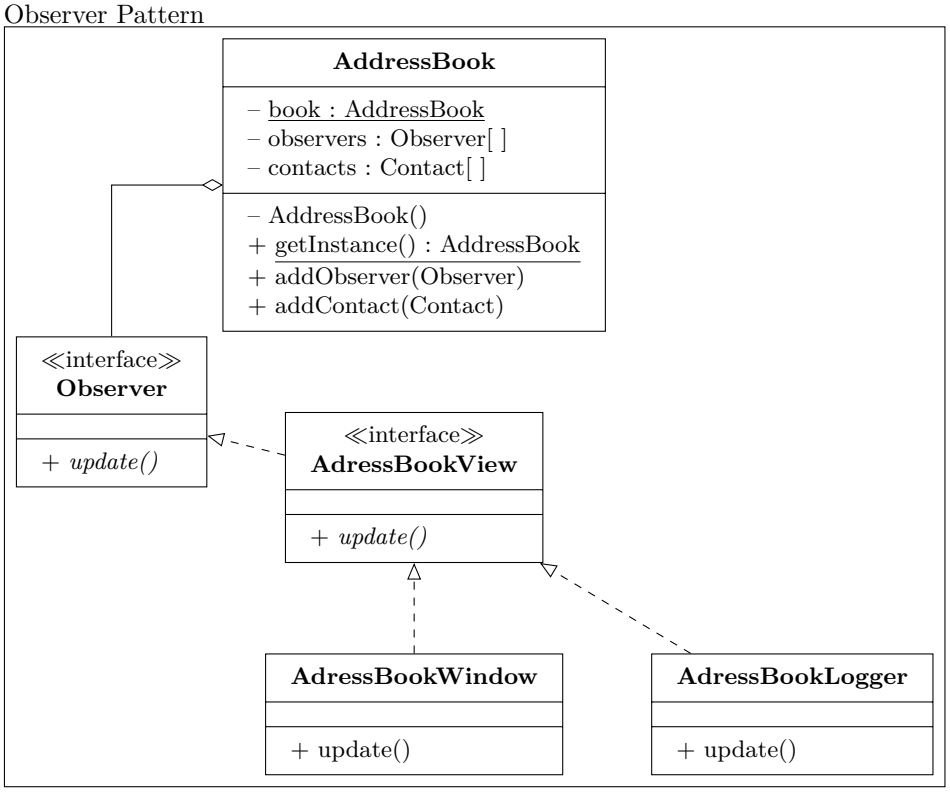
Aufgabe 2 Entwurfsmuster in einem UML-Diagramm erkennen

Identifizieren Sie im folgenden Klassendiagramm die *Design-Pattern*.



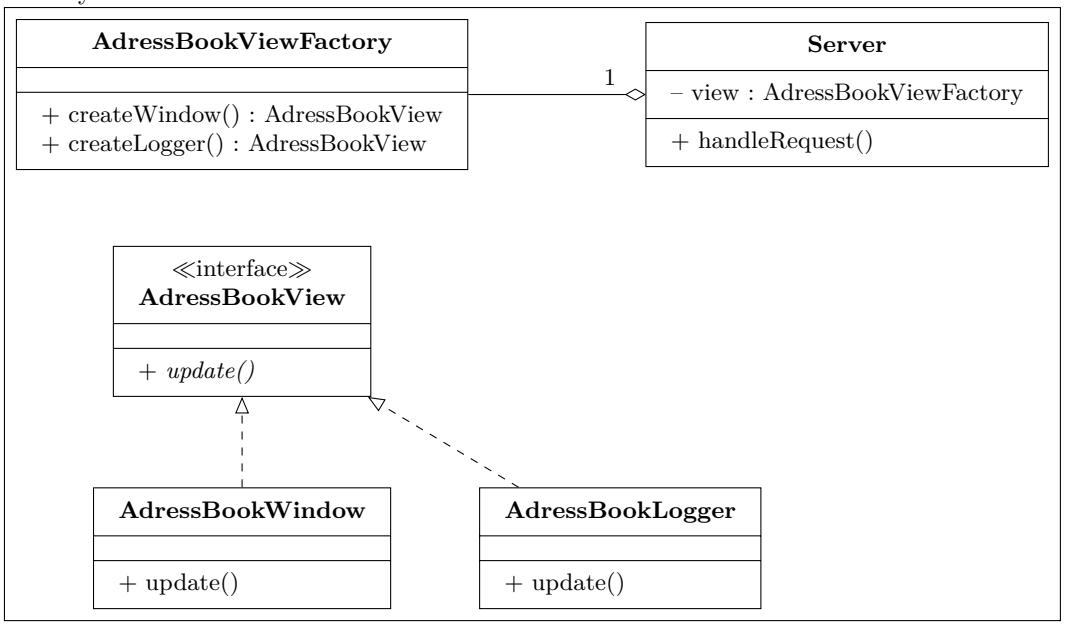
Lösung

- Observer Pattern



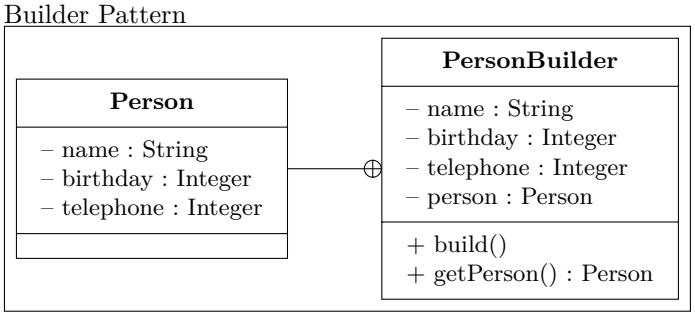
Design Patterns: Elements of Reusable Object-Oriented Software S. 293 ff.

- Factory Pattern



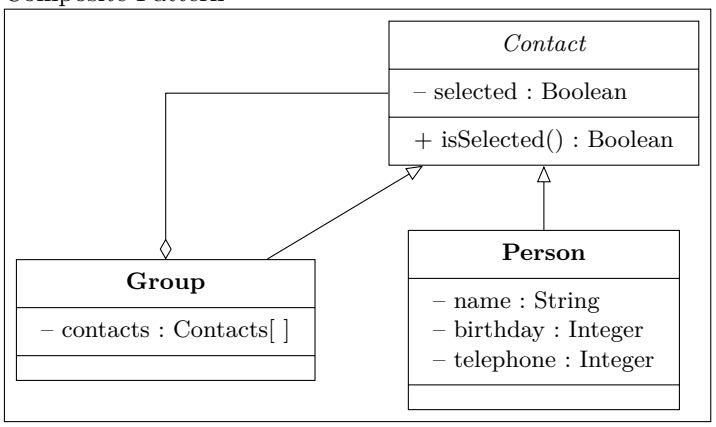
Design Patterns: Elements of Reusable Object-Oriented Software S. 87 ff.

- Builder Pattern



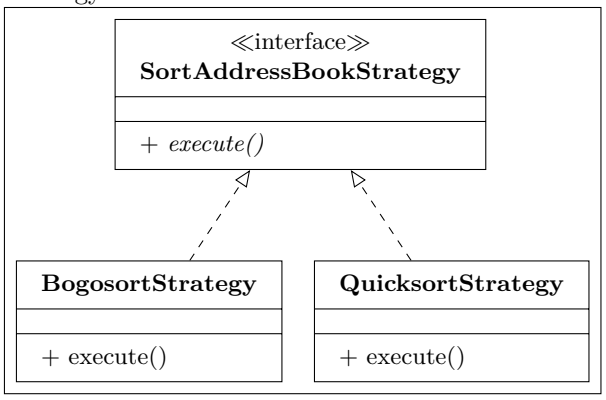
Design Patterns: Elements of Reusable Object-Oriented Software S. 97 ff.

- Composite Pattern



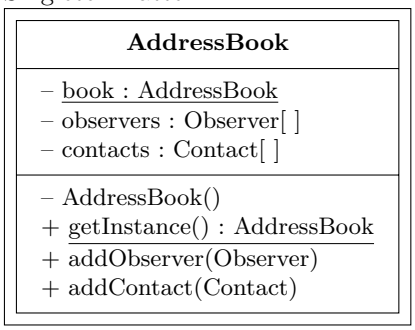
Design Patterns: Elements of Reusable Object-Oriented Software S. 163 ff.

- Strategy Pattern



Design Patterns: Elements of Reusable Object-Oriented Software S. 315 ff.

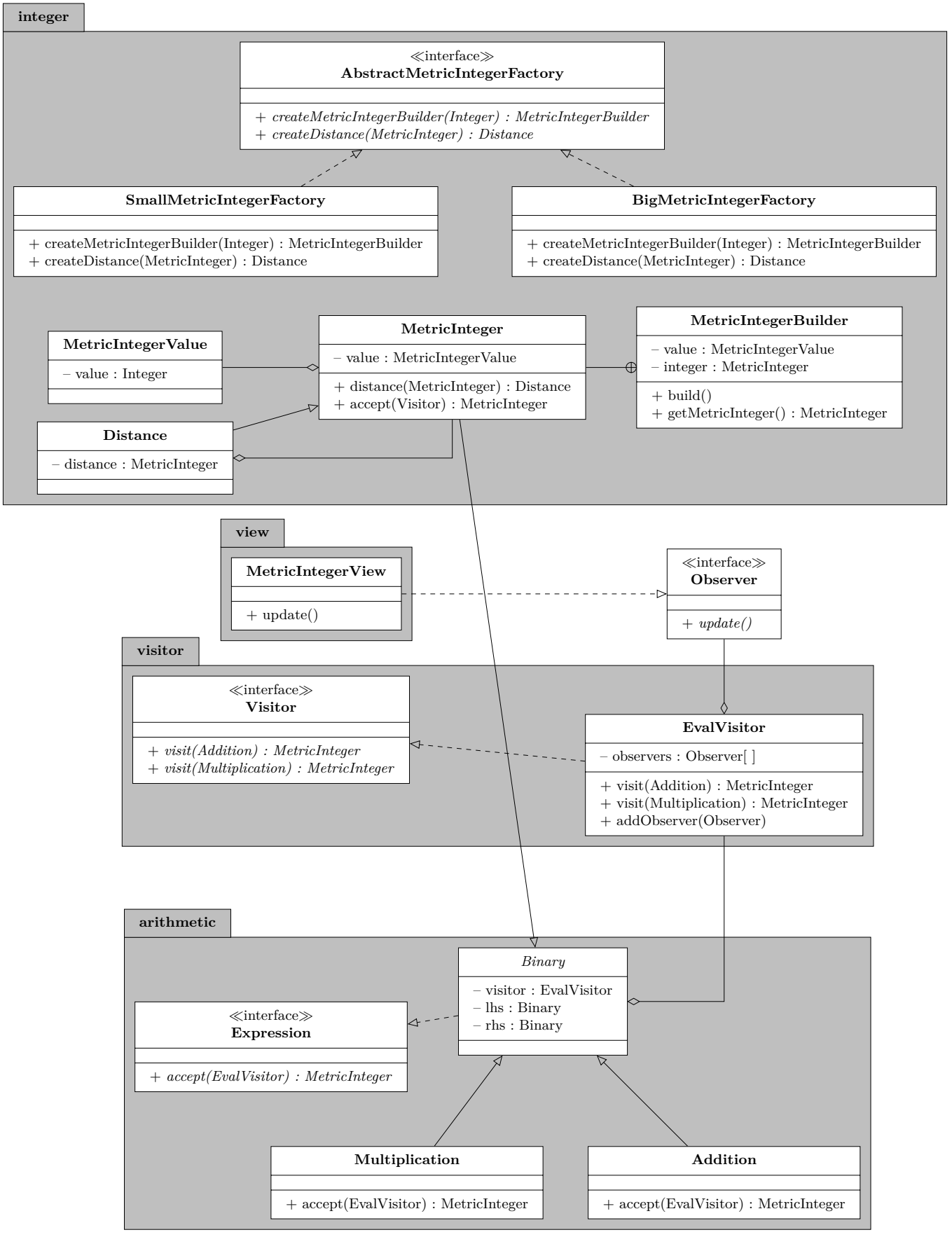
- Singleton Pattern



Design Patterns: Elements of Reusable Object-Oriented Software S. 127 ff.

Aufgabe 3 Entwurfsmustern zur strukturellen Optimierung verwerfen

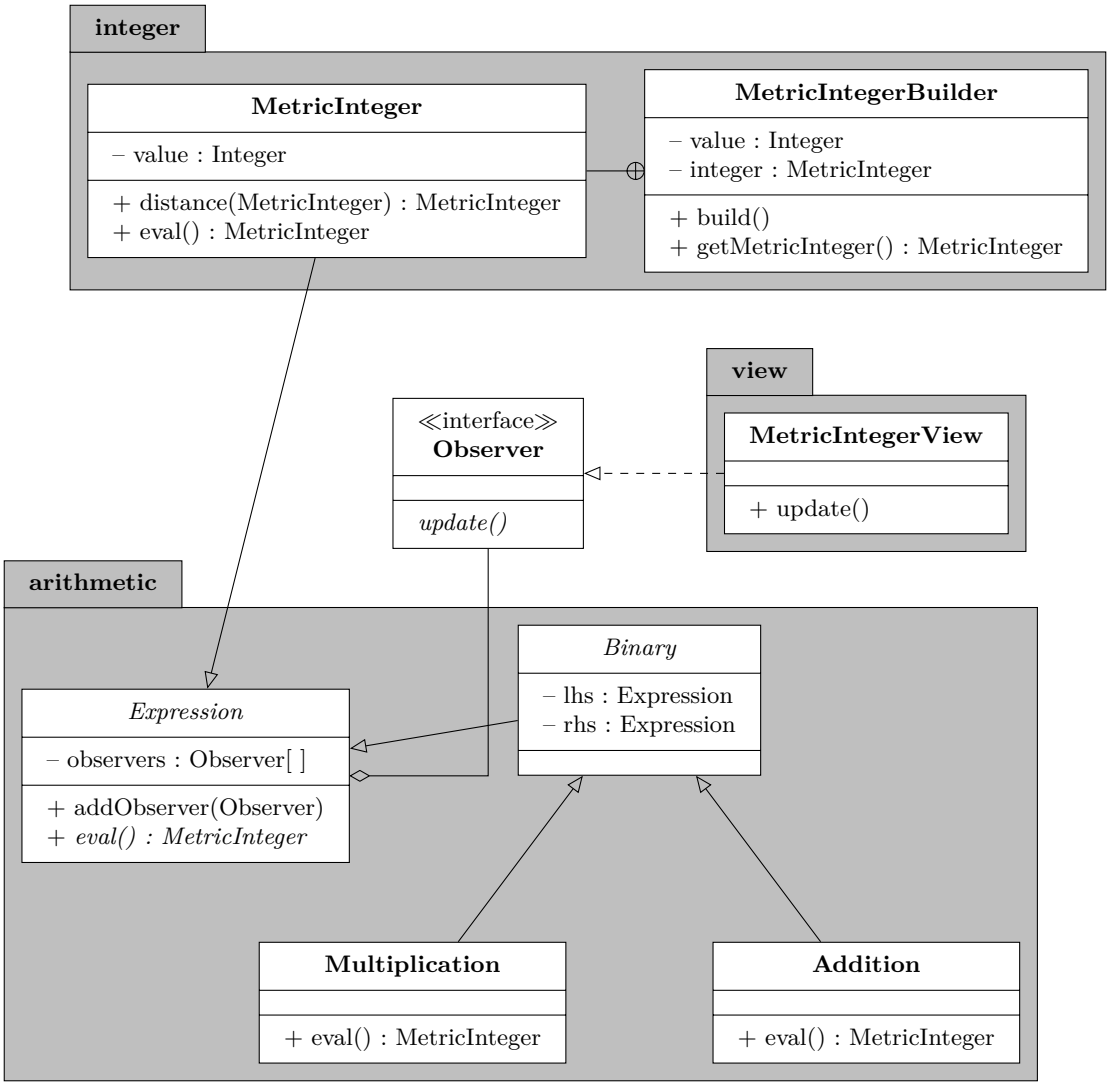
In dieser Aufgabe geht es darum, die Grenzen einer sinnvollen Strukturierung mit *Design Pattern* auszuloten, betrachten Sie hierzu das folgende Klassendiagramm. Finden Sie, dass für einzelne Muster deren Verwendung im gegebenen Kontext notwendig ist? Argumentieren Sie, wie Sie die Struktur gegebenenfalls vereinfachen bzw. verbessern würden (es gibt hierbei kein richtig oder falsch). Schauen Sie sich hierzu insbesondere an, ob Sie benötigte Klasseneigenschaften auch ohne Pattern erhalten können.



Lösung

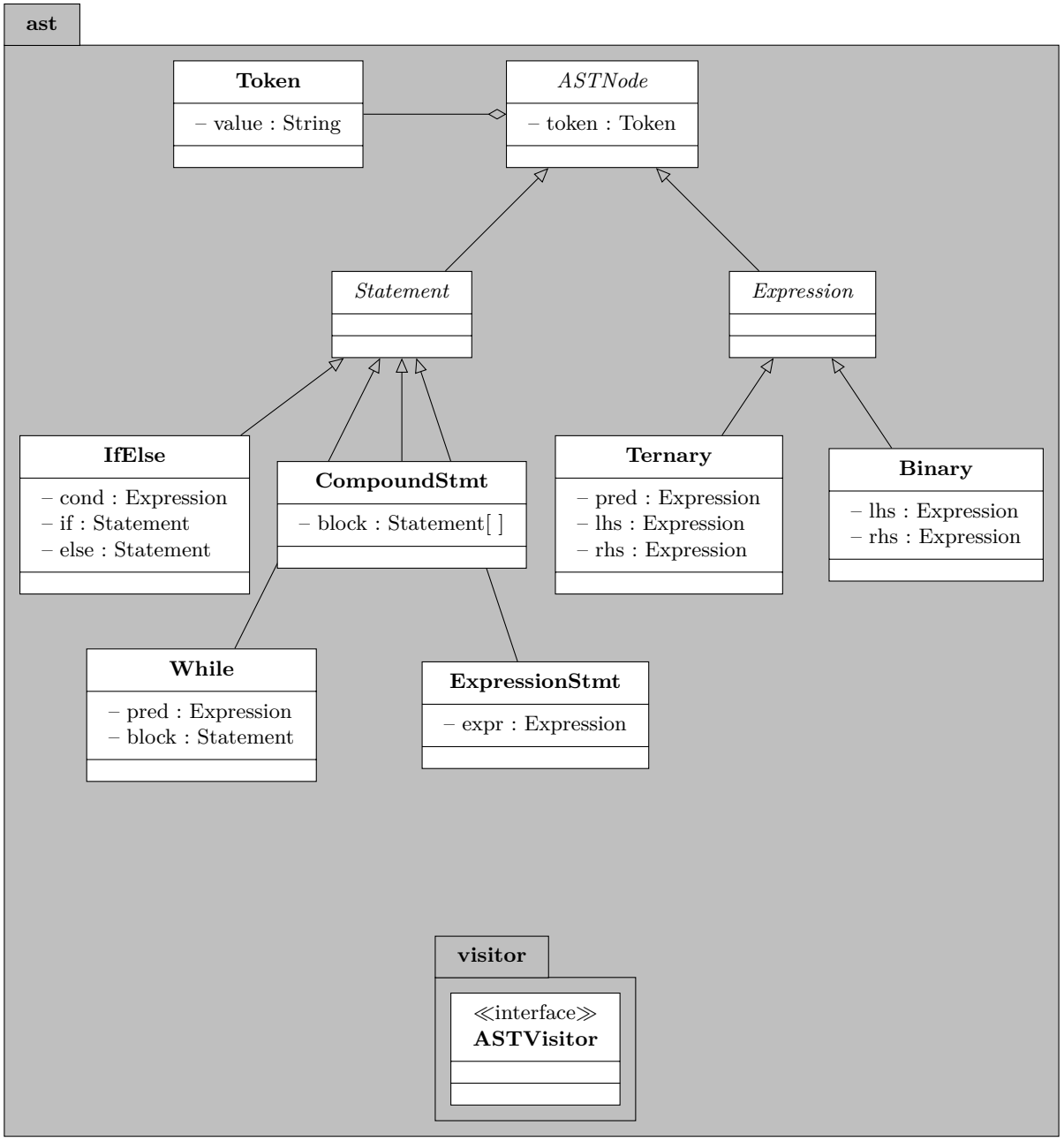
Hierbei handelt es sich lediglich um einen Vorschlag, die Klassenstruktur auf wesentliche Eigenschaften zu reduzieren, unter anderer Begründung kann sich Ihre Lösung deutlich unterscheiden.

Zunächst verzichten wir komplett auf die `AbstractMetricIntegerFactory` und behalten für `MetricInteger` lediglich den *Builder*, da durch die Hierachie die Unterklassen (und damit die *Factory* bzw. das *Composite*) obsolet sind. Wir entfernen den *Visitor*, da die Funktionalität keine Notwendigkeit für ein derartiges Pattern hat (insbesondere gibt es nur eine einzelne Implementierung durch *EvalVisitor*) - dies ist natürlich abhängig davon, ob das Modell gegebenenfalls erweitert werden sollte. Entsprechend wird auch die Verwaltung vom *Observer* in den `arithmetic`-Klassen angepasst. Die neue Grundklasse der Baumstruktur ist nun `Expression`.

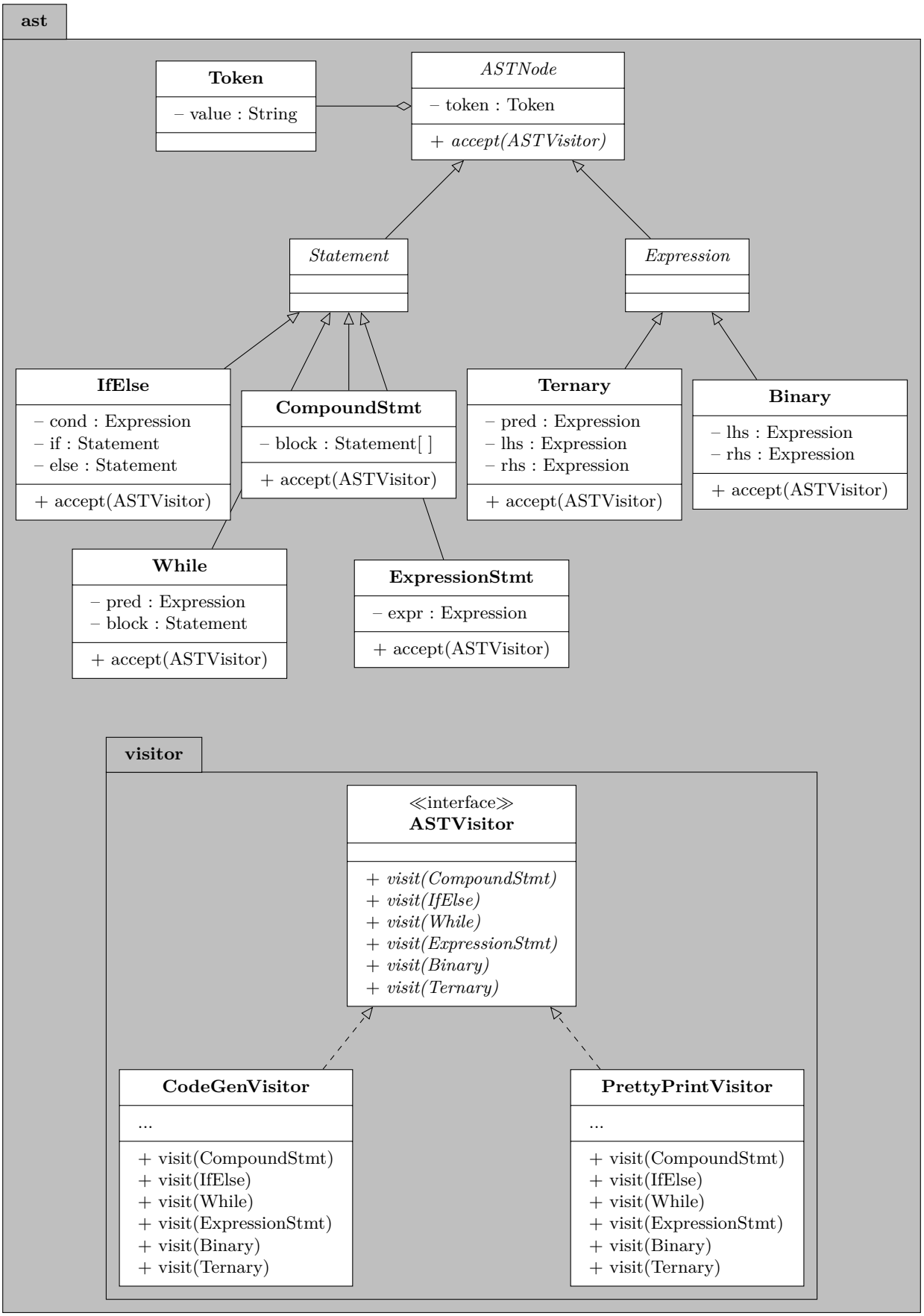


Aufgabe 4 Visitor für Baumstruktur in UML notieren

Entwerfen Sie für folgende Datenstruktur einen `CodeGenVisitor` sowie `PrettyPrintVisitor`, die jeweils das Interface `ASTVisitor` implementieren. Ergänzen Sie hierzu das Klassendiagramm und geben Sie für die *Visitor*-Klassen entsprechende Methoden an.



Lösung


ast
Token
– value : String
ASTNode
– token : Token
+ accept(ASTVisitor)
Statement
Expression
IfElse
– cond : Expression
– if : Statement
– else : Statement
+ accept(ASTVisitor)
CompoundStmt
– block : Statement[ ]
+ accept(ASTVisitor)
Ternary
– pred : Expression
– lhs : Expression
– rhs : Expression
+ accept(ASTVisitor)
Binary
– lhs : Expression
– rhs : Expression
+ accept(ASTVisitor)
While
– pred : Expression
– block : Statement
+ accept(ASTVisitor)
ExpressionStmt
– expr : Expression
+ accept(ASTVisitor)
visitor
≪interface≫
ASTVisitor
+ visit(CompoundStmt)
+ visit(IfElse)
+ visit(While)
+ visit(ExpressionStmt)
+ visit(Binary)
+ visit(Ternary)
CodeGenVisitor
...
+ visit(CompoundStmt)
+ visit(IfElse)
+ visit(While)
+ visit(ExpressionStmt)
+ visit(Binary)
+ visit(Ternary)
PrettyPrintVisitor
...
+ visit(CompoundStmt)
+ visit(IfElse)
+ visit(While)
+ visit(ExpressionStmt)
+ visit(Binary)
+ visit(Ternary)