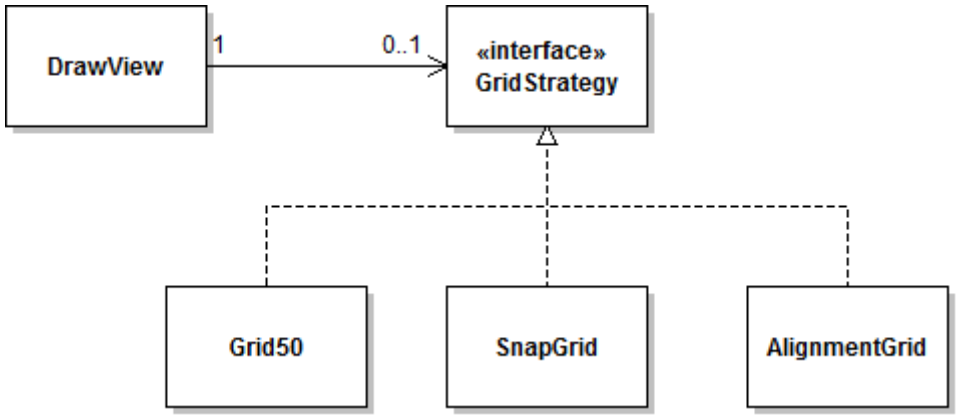
Arbeitsblatt: Strategy Pattern

Wenn man verschiedene Algorithmen verwenden möchte, um ein bestimmtes Problem zu lösen, dann ist die Verwendung des Strategy-Patterns angebracht. Der Kontext ruft dann die verschiedenen Implementierungen über ein gemeinsames Strategy-Interface auf.

Ein Beispiel, bei welchem wir dies betrachtet haben, ist das Grid in *JDraw*, mit dem die Mauskoordinaten auf ein „Grid“ abgebildet werden können.



Eine erste Version des Interfaces `GridStrategy` könnte wie folgt aussehen:

```
interface GridStrategy {
   Point mapPoint(Point p);
}
```

Die Frage ist jedoch, ob die Parameterliste für alle denkbaren Strategien ausreicht. Untersuchen Sie dies an folgenden Beispielen. Überlegen Sie dazu zunächst, welche Informationen die verschiedenen Grids für ihre Aufgaben benötigen und dann, wie sie diese Informationen bekommen können.

- **SnapGrid**
  Bei diesem Grid sollen die Mauskoordinaten auf die Handle-Positionen der anderen Figuren gelegt werden, falls sich diese genügend nahe bei der aktuellen Mausposition befinden.

- **ModifierGrid**
  Das Grid soll sich unterschiedlich verhalten, je nachdem ob die CTRL-Taste gedrückt ist oder nicht. Wenn z.B. in PowerPoint eine Figur mit den Maustasten verschoben wird, dann kann die Figur bei gedrückter CTRL-Taste feiner platziert werden als wenn diese Taste nicht gedrückt ist.

- **HistoryGrid**
  Wir könnten uns auch ein Grid vorstellen, das die letzten zehn Maus-Down und Mouse-Up-Positionen als Grundlage für das Grid verwendet.

Wie würden Sie die Schnittstelle der Methode `mapPoint` erweitern, dass diese (und auch weitere Grid-Implementierungen) realisiert werden können?

Beachten Sie, dass gewisse Parameter einer konkreten Grid-Klasse auch per Konstruktor übergeben werden könnten.

Weiterführende Aufgabe: Überlegen Sie sich auch konzeptionell, wie das SnapGrid implementiert werden kann. Wäre Ihre Implementierung zustandsbehaftet oder zustandslos?