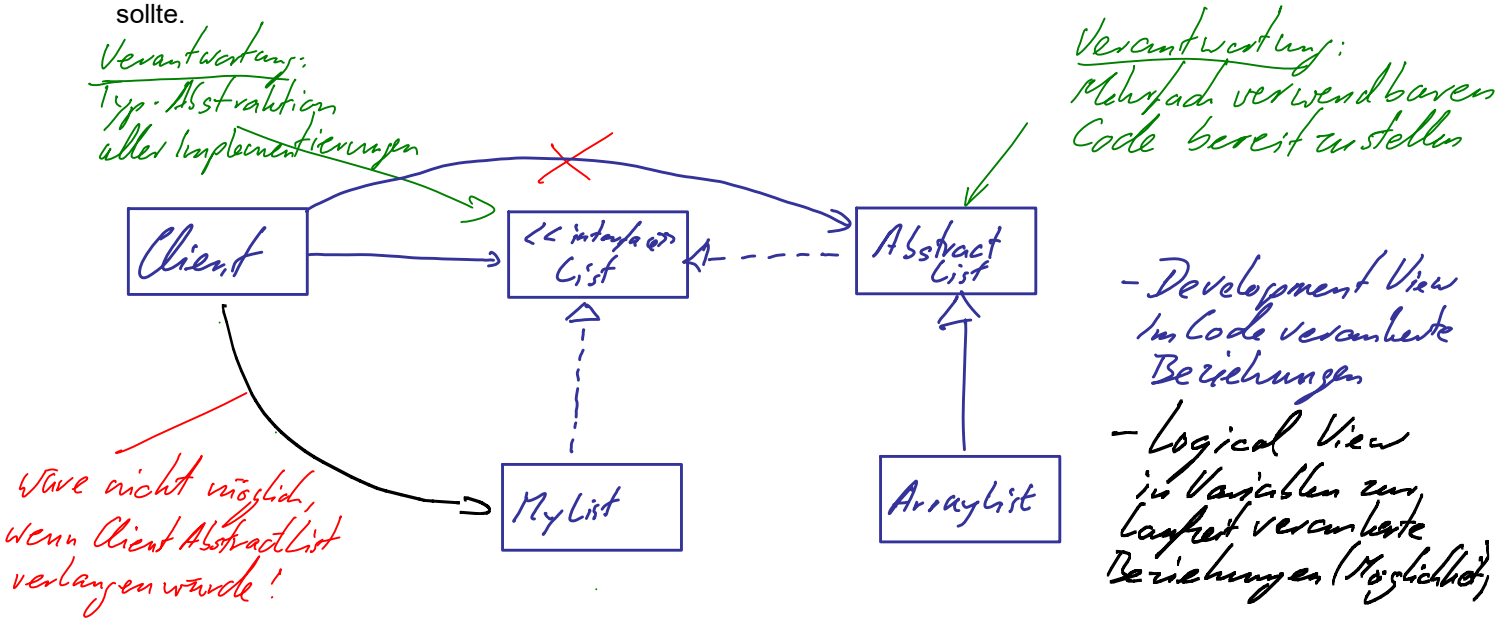
Abstraktionen zur Schaffung von Unabhängigkeit sollten grundsätzlich mittels *Interfaces* und nicht *abstrakten Klassen* als Typen formuliert werden; jedenfalls solange nicht sehr spezifische Gründe des Schnittstellenentwurfs etwas anderes notwendig machen. Ein formaler Grund dafür ist, dass man sich mit abstrakten Klassen in den meisten aktuellen Programmiersprachen die Möglichkeit zum *multiple Subtyping* vergibt.

Der primäre Zweck von abstrakten Klassen ist, für andere Klassen Code über Vererbung zur Verfügung zu stellen. Das ist aber eine vom *Subtyping* unabhängige Aufgabe, die deswegen abgetrennt werden sollte.



Schliesslich muss man dann natürlich auch konsequent darauf achten, bei Variablen und Parameter-Definitionen auch überall möglichst abstrakte Typen zu verwenden. Diese Typen sollten so abstrakt wie möglich gewählt werden, aber ohne dass deswegen Laufzeit Typ-Tests nötig werden.

4.5. Zusammenfassung

Die objektorientierte Programmierung vereinigt zwei wichtige Möglichkeiten. Erstens eignen sich Objekte gut zur Modellierung der Anwendungs-Domäne. Die Struktur dieses Modells liefert ein Gerüst für die Struktur der Software (*Low Representational Gap*), das mittels *Responsibility-Driven Design* verfeinert werden kann. Ziel ist Software zu erhalten, die leichter verständlich ist und deren Struktur durch ihre Ähnlichkeit mit der Anwendungs-Domäne leichter zu plausibilisieren ist. Darüber hinaus lassen sich über die Modellierung von *Rollen*, die von Objekten eingenommen werden, unter Einhaltung des *Interface Segregation Principle* Typhierarchiemodelle ableiten.

Zweitens eignen sich aus Objekten aufgebaute Systeme gut zur Weiterentwicklung und Anpassung an sich ändernde Gegebenheiten. Primär ist dabei, dass man einzelne Objekte auswechseln kann, die dann mit dem (unveränderten) Rest des Systems weiterhin zusammenarbeiten. Hier helfen die Einhaltung von Prinzipien wir dem *Single Responsibility Principle*, denn dann kann man hoffen, dass nötige Änderungen sich gerade auf den Scope eines einzelnen Objektes beschränken.

Erweiterungen und Weiterentwicklungen müssen Objektverhalten verändern, sollen aber so weit möglich bereits vorhandenen Source Code in einzelnen Objekten wiederverwenden. Voraussetzung dafür ist vor allem die Einhaltung des *Open / Closed Principle*. Entstehen neue Objekttypen als Sub-Typen bestehender Objekte, ist auf die Einhaltung des *Liskov Substitution Principle* zu achten.