**Key Points**

* Interfaces promote consistency and use of heterogeneous collections
* Inheritance gives software no more computational power than any other structural relation
* When an internal data structure provides some functionality but could easily be replaced without compromising functionality, a holds-a relation may suffice
* Aggregation merely addresses form and less so intent and effect
* Composition: - the subobject is stable. Lifetimes are often directly correlated; that is, the subobject exists for the lifetime of the object. The cardinality of a subobject is usually fixed within the lifetime of the object. For example, if a plane model is designed with two engines then it must have two engines. Likewise, if the plane design dictates four engines.
* An Aggregation relationship provides much flexibility because of its variability in cardinality, ownership, and association
* Inheritance of implementation is often called contraction because the child class represents a more restricted form of the parent class.
* Substitutability: an instance of a derived class can stand in for an instance of a base class Not a symmetric relation: parent cannot stand in for child
* Multiple inheritance increases software complexity. Minimally, cohesion decreases because the child class definition is spread across three or more different classes. Coupling increases because the child class is tied to two or more parent classes. Multiple inheritance also may yield two specific design difficulties: **ambi­guity** and **redundancy**
* Design should depend on goals and priorities rather than comfort due to experience
* Inheritance for specialization models refinement
* Inheritance for specification models abstraction
* Inheritance for extension models type extension
* Design options require choice
  + Tradeoffs must be evaluated
  + Immediate use versus anticipated change
* Class Types
  + *A* ***concrete*** *class is immediately usable because its class designer provides a fully functional, complete interface*
  + An **abstract** class is a placeholder for extensibility and is not immediately usable. Inheritance is anticipated; Abstract classes are designed for generality, for heterogeneous collections.
  + *A* ***wrap­per*** *class layers an interface over an existing class (or set of classes) to iso­late users* from an unstable class or interface. A wrapper defines and *controls* a consistent interface that may be layered over one of multiple implementations. Wrappers promote the reuse of existing classes for new or modified applications
  + A **delegate** class provides functionality and promotes flexibility. Delegates are often held as subobjects inside another class, and may be instantiated internal or external to the containing class. Delegates provide functionality to an enclosing class, thus isolating the application code from change: the enclosing class can easily modify or replace its delegate
* Specification differs from specialization in that the child class is not a refinement of existing usable type but a realization of incomplete abstract specification.
* What inheritance provides that composition does not? Type exten­sion, substitutability, and support for heterogeneous collections
* In contrast to inheritance, composition affords design variety in terms of cardinality, association, ownership, and subobject instantiation
* Through composition, a class design may:
* Wrap up existing code
* Providing isolation from change
* Alter cardinality
* postpone instantiation
* Support replacement
* Transfer or share ownership