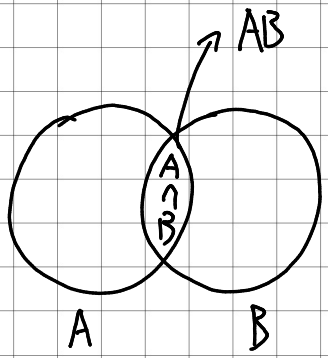
Same method name, different implementations are treated as different functionalities

Assume 1. no overriding (no diamond problem)
2. allow multiple inheritance

inheritance of implementation

textbook scenario starts with multiple entities closely related,
try to generalize a common subset of functionality as superclass
use inherience of implementation to save/reuse code.

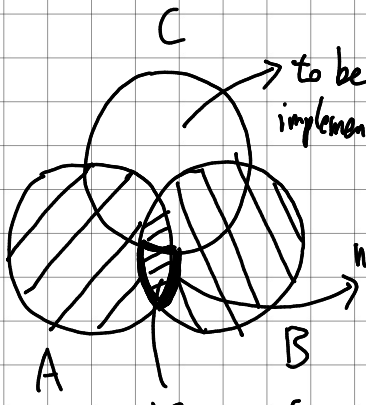


class AB

class A inheries AB

class B inheries AB

① add new entity into the ontology



Inherience

ABC {ABC}

AB{AB|ABC} AC{AC|ABC} BC{BC|ABC}

A{A|AB,AC,ABC} B C

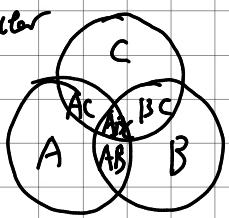
Composition

A{A, AB, AC, ABC}

B{B, AB, BC, ABC}

C{C, AC, BC, ABC}

refactor



composition of implementation

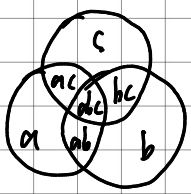
define "unit of functionality"

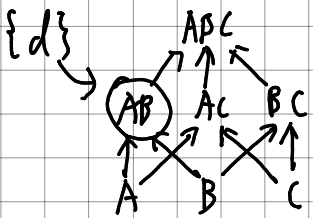
self-contained, independent

To be continue

---

② add new functionality to existential entities.





1. Locate the base class in the hierarchy to append the functionality.

   for example, {d} shared by A and B. ⇒ AB

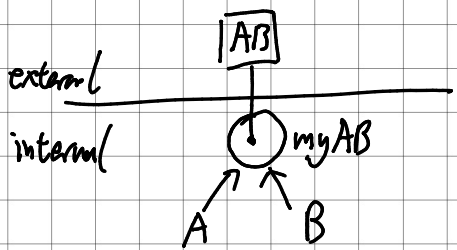
2. add {d} to AB.

   AB {ab, d | abc}

A{a | ab, ac, abc, d} B{b | ab, ac, abc, d}

Problem target base class is out-sourced and doesn't allow direct modification.

Solution delegate base case AB => myAB (regain full controll internally) and modify 'myAB'.



- Inheritence of Interface
- Multiple Dispatch
  1. reflection cons: performance issue, give up static type checking
  2. visiter pattern (double dispatch)
     cons: manual routing, strong coupling
     pro: type guarantee
- Message Protocol Design
  1. point-to-point
  2. mediator
     - a simple protocol shared by all clients (complying to protocol implementation)
     - mediator knows all messages of all clients

cons
- string coordination
- runtime validation

(could be compile-time if you build a tool around it)

compile-time validation

DI approach to factor out object graph dependencies