Core OOSD: SOLID

Object Oriented System Design

Otago Polytechnic
Dunedin, New Zealand

Background

- OO Programming emerged in the 1980’s qnd 1990’s.
- By the early 2000’s programmers were building larger programs and started to recognise design problems.
- Robert Martin (“Uncle Bob”) identified five key principles that have become known by the acronym SOLID.

Guidlelines

- It’s a bit of a cliché that almost any question in an advanced programming class can be answered, “It depends.”
- How closely should you follow the guidlelines we will discuss this morning? It depends.
- You should try to follow them. When you do find yourself breaking them, it should be because you’ve made a deliberate choice to do so.

SOLID

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface substitution principle
- **D**ependency inversion principle

Single responsibility principle

- A class should do only one thing.
- You should be able describe a class' purpose with one *concise* sentence.
- When specifications change, a class should have only one reason to change.

Example: Card class

- What is the job of the playing card?
- Should it report its numeric score value for blackjack?
- This would mean that the cards would need to know about blackjack rules.

Open/closed principle

- Classes should be *open* to extension, but *closed* to modification.
- This means that consumers of a class can rely on its methods remaining available, i.e., changes to the class won't hurt current uses of the class.
- We can extend a class so that it can do more, but we never take away or modify exisiting functionality.

Example: Deck class

- I wrote my `Deck` class to use one 52 card deck.
- Casinos usually use multiple decks to make card counting harder.
- But if I *modify* the deck class to use more decks, I may break exisiting uses of the class.
- However, I could *extend* the class, for example by supplying an alternate constructor, without breaking preexisiting uses.

Liskov substitution principle

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface segregation principle

- Many client-specific interfaces are better than one general-purpose interface.
- Clients should not be forced to depend upon methods they do not use.

Dependency inversion principle

- High level modules should not depend on low-level modules.
- Both should depend upon abstractions.