Class Design

Object-Oriented Programming with C++

Designing classes

- How to write classes in a way that they are easily understandable, maintainable and reusable

Class Design: What to do

- How many types of class do we need?
- When to define a class?
- What interface and data in a class?
- Do we need to construct inheritance to promote interface and code reuse?
- Which function should be virtual to support dynamic binding in run-time?

Contents

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted…
- The work is done by different people over time (often decades).

Change or die

- There are only two options for software:
  - Either it is continuously maintained
  - Or it dies.
- Software that cannot be maintained will be thrown away.

Code quality

- Two important concepts for quality of code:
  - Coupling
  - Cohesion

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.

If X changes→how much code in Y must be changed?

Loose coupling

- Loose coupling makes it possible to:
  - Understand one class without reading others;
  - Change one class without affecting others.
  - Thus: improves maintainability.

Tech. to loose

- call-back
- message mech.

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has high cohesion.
- Cohesion applies to classes and methods.
- We aim for high cohesion.

High cohesion

- High cohesion makes it easier to:
  - Understand what a class or method does;
  - Use descriptive names;
  - Reuse classes or methods.

Cohesion of methods

- A method should be responsible for one and only one well defined task.

Cohesion of classes

- Classes should represent one single, well defined entity.

Code duplication

- Code duplication
  - is an indicator of bad design,
  - makes maintenance harder,
  - can lead to introduction of errors during maintenance.

Responsibility-driven design

- Question: where should we add a new method (which class)?
- Each class should be responsible for manipulating its own data.
- The class that owns the data should be responsible for processing it.
- RDD leads to low coupling.

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.

Thinking ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken.

Design questions

- Common questions:
  - How long should a class be?
  - How long should a method be?

- Can now be answered in terms of cohesion and coupling.

Design guidelines

- A method is too long if it does more then one logical task.
- A class is too complex if it represents more than one logical entity.
- Note: these are *guidelines* – they still leave much open to the designer.

Review

- Programs are continuously changed.
- It is important to make this change possible.
- Quality of code requires much more than just performing correct at one time.
- Code must be understandable and maintainable.

Review

- Good quality code avoids duplication, displays high cohesion, low coupling.
- Coding style (commenting, naming, layout, etc.) is also important.
- There is a big difference in the amount of work required to change poorly structured and well structured code.