Software Design

“There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies.”

-- C.A.R. Hoare (1985)
Readings

- “Code Complete (2nd ed.),” chapter 5, pg. 87-105 by Steve McConnell (link on Calendar)

Summary is only on a portion of the chapter, but please read the whole chapter. It’s good!
Where does design fit?

Requirements

Architecture – high level design

Low level design

Code
Architecture is design

- Architecture, high level design, focuses on system components and their connections.
- Low level design is a step closer to code. It looks at objects, classes, algorithms, variables, …
- The boundary between high and low level design is blurry. Many of the same principles apply!
Good design quotes

The driving force behind design is managing complexity

The goal of all software design techniques is to break a complicated problem into simple pieces

The key to performance is elegance, not battalions of special cases.

Pragmatic Programmer
What do we want in a design?

Top 7 desirable characteristics:

1. Minimal complexity
2. **Loose coupling**
3. **Strong cohesion**
4. Extensibility
5. Reusability
6. Maintainability
7. Leanness
Loose coupling

- **coupling** assesses the kind and quantity of interconnections among modules.

- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled.

- The more tightly coupled are two modules, the harder it is to work with them separately, and thus the benefits become more limited.
Eliminate Effects Between Unrelated Things –
design components that are self-contained,
independent, and have a single, well-defined purpose

Pragmatic Programmer
Tightly or loosely coupled?
Tightly or loosely coupled?

[Diagram showing relationships between User Interface, Graphics, Data Storage, Application Level Classes, Business Rules, and Enterprise Level Tools with various associations labeled.]
Tightly or loosely coupled?
Strong cohesion

- *cohesion* refers to how closely the operations in a module are related

- Tight relationships improve clarity and understanding

- Classes with good abstraction usually have strong cohesion

- No schizophrenic classes!

CSE 403, Spring 2008, Alverson
Strong or weak cohesion?

class Employee {
public:
    …
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;
    …
    bool IsJobClassificationValid(JobClassification jobClass);
    bool IsZipCodeValid(Address address);
    bool IsPhoneNumberValid(PhoneNumber phoneNumber);
    …
    SqlQuery GetQueryToCreateNewEmployee() const;
    SqlQuery GetQueryToModifyEmployee() const;
    SqlQuery GetQueryToRetrieveEmployee() const;
    …
};
How do we attack the design process?

Treat design as a wicked, sloppy, heuristic process. Don’t settle for the first design that occurs to you.

Collaborate.
Strive for simplicity.
Prototype when you need to.
Iterate, iterate, and iterate again.

-- Steve McConnell, Code Complete
Heuristics help provide ideas

- There is no “right answer” with design
- Applying effective heuristics can provide insights and lead to a good design

- Code Complete Chapter 5 looks at 9 heuristics:
  1. Identify objects
  2. Form consistent abstractions
  3. Encapsulate implementation details
  4. Favor composition over inheritance
  5. Hide information
  6. Keep coupling loose and cohesion strong
  7. Identify areas likely to change * in your project
  8. Use design patterns
  9. Consider testability
  10. Other common principles
Inheritance

Inheritance promotes abstraction and reuse but breaks encapsulation.

The main guiding philosophy is that inheritance should be used only when a subclass is-a superclass.

```
Person

Kid
```
**Composition**

*Compose* by making a class an *instance* of another class (instead of a subclass)

```cpp
Class Kid {
    private:
        EatingPattern *ep;
    }

Kid::Kid()
{
    ep = new PickyEatingPattern;
}
```
Favor composition over inheritance

```
kid = new PickyEatingPatternKid;
```
Composition

- Fosters reuse
- Keeps class libraries shallow (easier to understand)
- Keeps encapsulation strong
- Decouples concepts
- Enables runtime binding
Design patterns

A design pattern is a repeatable solution to a common software design problem

- example: **Iterator** pattern
  
  Defines an interface that declares methods for sequentially accessing the objects in a collection.

- Example: **Factory** pattern
  
  Defines an interface that declares methods for letting a class defer instantiation to its subclass.
Recent history of patterns

- In 1990 a group called the Gang of Four (Gamma, Helm, Johnson, Vlissides) compiled a catalog of design patterns

- 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* is a classic of the field
A pattern describes a recurring software structure:
- is abstract from concrete design elements
- identifies classes, their collaborations and responsibilities
- lists implementation trade-offs
- are not code or designs; are “templates” to be instantiated

The software engineer is required to:
- evaluate trade-offs of using a pattern
- make design and implementation decision how best to apply the pattern, perhaps modify it slightly
- implement the pattern in code
Can you think of benefits?

- Benefits of using patterns (from Sp08):
  - common vocabulary
  - allows new developers to come up to speed faster
  - use the results of experience in the field
  - provides a starting point for coding
  - capture good design principles, allowing extensibility, and all that good stuff
Great website on patterns

http://sourcemaking.com/design_patterns

• complete with code samples
Gang of Four (GoF) patterns

- **Creational Patterns** (abstract object instantiation)
  Abstract Factory, Factory, Builder, Prototype, Singleton

- **Structural Patterns** (combine objects)
  Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

- **Behavioral Patterns** (communication between objects)
  Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
403 Mid-Quarter Assessment

- Lead by

Jim Borgford-Parnell, Ph.D.
Assistant Director
Center for Engineering Learning and Teaching