Design Flaws[1]

In order to design well, it helps to understand common sources of flawed designs. The symptoms described below were elaborated in the Martin reference, cited in the footnote, below.

Rigidity

- The design is hard to change:
  "A design is rigid if a single change causes a cascade of subsequent changes in dependent modules." This is often caused by tight coupling between many modules.

Fragility

- The design is easy to break:
  "Fragility is the tendency of a program to break in many places when a single change is made." Fragile designs often result from the use of global data, wide-spread sharing of mutable data, and the use of macros and literal numbers instead of named variables, often called "magic numbers".

Immobility

- The design is hard to reuse:
  "A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great." This is often caused by many concrete bindings[2] between modules, especially when they pass a lot of user defined types that are unique to the embedding system.

Viscosity

- Hard to do the right thing:
  "When design-preserving [changes] are harder to employ than hacks, the viscosity of the design is high." Viscosity of design is caused by complexity, of either interface or implementation.

Needless Complexity

- Over design:
  "A design contains needless complexity when it contains elements that aren't currently useful." Often this is a flaw in properly factoring a design into implementation packages[3].

---

[1] Follows closely the presentation in Agile Software Development, Robert C. Martin, Prentice-Hall, 2003

[2] A lot of the focus of this course is on ways to bind to abstractions rather than concrete classes.

[3] We embellish a class with all the functionality we think it may ever need rather than factoring out only those parts needed immediately.

Needless Repetition

- Cut and Paste:
  ”Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations.” If we duplicate code rather than factoring into a component for reuse, fixing errors and performance problems are much harder to effect.

Opacity

- Disorganized expression:
  ”Opacity is the tendency of a module to be difficult to understand.” “… developers need to put themselves in their readers’ shoes and make a concerted effort to refactor their code so that their readers can understand it.”

Design Issues and Principles[4]

Basic:
- KISS Principle: Correctness, simplicity, and clarity come first
- Cohesion: give each entity one responsibility
- Rely on Types not on representations
- Understand C++ scope and memory model (see RAII)

Basic Structure:
- No globals, minimize shared data, hide information
- Parameters: when to pass by value, pointer, smart pointer, or reference
- Inheritable classes: virtual destructor, semantics of virtual versus non-virtual

Module Structure:
- At most two files
- Make headers self sufficient
- Proper header and implementation structure
- No definitions with linkage in header file
- Consider non-member non-friend functions (part of module interface)
- Keep type and nonmember interface in same module
- Minimize definition dependencies, avoid cyclic dependencies
- Use namespaces and proper using declarations
- Prefer portable types in a module's interface

Resource Handling:
- RAII Principle: resources owned by objects
- use smart pointers, especially in STL containers

Polymorphism:
- Liskov Substitution Principle (strategy pattern)
- Open and Closed Principle (use hooks)
- Prefer abstract interfaces
- Overloading, overriding, generics
- Prefer explicit conversions
- Consider cloning

Binding Structure:
- Dependency Inversion Principle (interfaces and factories)
- Interface Segregation Principle (mixins)
- Prefer non-public virtual functions (Template Pattern)

Exception Safety
- Pimpls
- Destructors, de-allocation, and swap never fail
- Distinguish between errors and non-errors
- Throw by value, catch by reference

---

[4] Makes liberal use of the materials in C++ Coding Standards, Sutter and Alexandrescu, Addison-Wesley, 2005

Design Gotchas[5]

Structure:
- Must use constructor initializers
- Don't make code depend on order of function arguments
- Don’t depend on the order of initialization of modules
- Avoid allocating and de-allocating in different modules

Resource Handling:
- Avoid calling virtual functions in constructors and destructors

Polymorphism:
- Don't overload or use default parameters in virtual functions
- Don't treat arrays polymorphically

Standard Template Library (STL):
- Beware of iterator invalidation, especially with vectors

[5] Makes liberal use of the materials in C++ Gotchas, Dewhurst, Addison-Wesley, 2003

Basic Objects

FileInfo
SmartPointer
Thread
BlockingQueue
Task
Graph