Factoring Code

This page is obsolete, with the information incorporated into the updated HotSpot Style Guide.

Suggestions for Factoring and Class Design

[This flat list needs more organization.]

- Keep functions small, a screenful at most.
- Factor away nonessential complexity into local inline helper functions and helper classes.
- Think clearly about internal invariants that apply to each class, and document them in the form of asserts within member functions.
- Make simple, self-evident contracts for class methods. If you cannot communicate a simple contract, redesign the class.
  Implement classes as if expecting rough usage by clients. Rough use that breaks a class must throw an assert.
- When possible, try to design as if for reusability, simply because this forces a clear design of the class's externals, and clean hiding of its internals.
- Initialize all variables and data structures to a known state. If a class has a constructor, initialize it there.
- Do no optimization before its time. Prove the need to optimize.
- When you must defactor to optimize, preserve as much structure as possible. If you must hand-inline some name, label the local copy with the original name.
- If you need to use a hidden detail (e.g., a structure offset), name it (as a constant or function) in the class the owns it.
- Don't use the Copy and Paste keys to replicate more than a couple lines of code. Name what you must repeat.
- Don't give two names to the same thing. Try not to give two names similar things: Derive one from the other or both from a private third.
- When choosing names, avoid categorical nouns like "variable", "field", "parameter", "value", and verbs like "compute", "get". ("`storeValue(int param)`" is bad.)
- Names should be verb phrases that reflect changes of state known to a class's user, or else noun phrases if they cause no change of state visible to the class's user.
- If a class needs a method to change a user-visible attribute, the change should be done with a "setter" accessor matched to the simple "getter".
- Choose names consistently. Do not introduce spurious variations. Abbreviate corresponding terms to a consistent length.
- Be sparing and tasteful with C++ function overloading and optional arguments, because they will easily confuse the reader.
- Take care to avoid fancy C++ features, such as operator overloading, copy constructors, implicit conversions, covariance, etc.
- Absolutely avoid multiple inheritance, exceptions, namespaces. Introduce new templates only by Act of Congress..

(Collected 7/2002.)

Issues:

- code chunk size
- code chunk boundaries
- quality communication regarding class internals
- quality communication regarding interfaces