Robot Vision Systems

Lecture 7: Good Design Principles

Michael Felsberg

michael.felsberg@liu.se

Software Construction

- software engineering discipline
- detailed creation of working, meaningful software through a combination of
  - coding
  - verification
  - unit testing
  - integration testing
  - debugging

Why is software construction important?

- large part of software development
- central activity in software development
- focus on construction: individual programmer's productivity improves
- product, the source code, is often the only accurate description of the software
- the only activity that's guaranteed to be done

Key Construction Decisions

- programming language's strengths and weaknesses: be aware of!
- establish programming conventions **before** you begin programming
- more practices exist than you can use: consciously choose the best suited one
- are practices a response to the programming language or controlled by it?
- program *into* the language, rather than *in* it

Design in construction

- primary: *managing complexity* - greatly aided by *simplicity:*
  - minimize the amount of essential complexity to deal with
  - keeping accidental complexity from growing
- heuristic; Dogmatic adherence hurts creativity and programs
- iterative; try design possibilities
- information hiding: “What should I hide?”

Desirable characteristics of a design

- *Minimal complexity*
- *Ease of maintenance*
- *Loose coupling*

Desirable characteristics of a design

- *Extensibility*
- *Reusability*
- *High fan-in*
- *Low-to-medium fan-out*

Desirable characteristics of a design

- *Portability*
- *Leanness*
- *Stratification*
- *Standard techniques*

Working classes

- primary tool for managing complexity
- interfaces should
  - provide a consistent abstraction
  - hide something
- containment is usually preferable to inheritance
  - unless modeling an “is a” relationship

Working classes

"*If inheritance is a chain saw, multiple inheritance is a 1950s-era chain saw with no blade guard, no automatic shutoff, and a finicky engine. There are times when such a tool is valuable; mostly, however, you're better off leaving the tool in the garage where it can't do any damage*."

High-Quality Routines

- creating a routine is to improve the intellectual manageability of a program
- put simple operations into a routine of its own
- name of a routine: indication of its quality

High-Quality Routines

- primary purpose of a function is to return the specific value described by its name
- use macro routines only as a last resort

Defensive Programming

- a routine is passed bad data, it won't be hurt, even if it is another routine's fault
- programs have problems and modifications, the programmer develops code accordingly
- parts with dirty data and parts with clean data: relieve majority for checking data
- more sophisticated way than "garbage in, garbage out"

Defensive Programming

- errors easier to find, to fix, and less damaging
- assertions detect errors early, in large + high-reliability systems; fast-changing code
- how to handle bad inputs is a key decision: error-handling and high-level design
- exceptions: handling errors in a different dimension from the normal flow of the code

General issues in using variables

- initialize variables when declared to avoid unexpected initial values
- minimize the scope variables and keep it local to a routine or a class
- cluster statements with the same variables
- early binding: reduce flexibility & complexity
  late binding: increase flexibility & complexity
- each variable for one and only one purpose

The power of variable names

- key element of readability; specific kinds require specific considerations
- as specific as possible - vague / general names for multi-purpose = bad names
- conventions: local, class, global data; distinguish type names, named constants, enumerated types, and variables
- adopt convention, depending on size of program and the number of programmers

The power of variable names

- abbreviations rarely needed
- use project dictionary or standardized prefixes approach
- favor read-time convenience over write-time convenience

Guidelines for making use of numbers less error-prone

- use named constants instead of “magic numbers”
- the only literals that should occur in the body of a program are 0 and 1
- prevent divide-by-zero error
- make type conversions obvious

Guidelines for making use of numbers less error-prone

- avoid mixed-type comparisons and do the conversion manually
- heed compiler's warnings
- eliminate all compiler warnings

Creating types

- own types make programs easier to modify and self-documenting
- refer to represented problem part
- consider new class instead of *typedef*
- avoid predefined types
- don’t redefine predefined types

Unusual data types

- structures make programs less complicated, easier to understand and to maintain
- consider class instead of structure
- pointers are error-prone, protect yourself!
- avoid global variables
- use access routines for global variables

Organizing straight-line code

- strongest principle: ordering dependencies
- make dependencies obvious through routine names, parameter lists, comments, and housekeeping variables
- in absence of order dependencies, keep related statements close together

Using conditionals

- *if-else* statements: pay attention to the order; make sure the nominal case is clear
- *if-then-else* chains and *case* statements: choose an order that maximizes readability
- trap errors: default clause (*case*) or last *else* (*if-then-else*)
- choose control construct that's most appropriate for each section of code

Controlling loops

- keep loops simple for readability
  - avoid exotic kinds of loops
  - minimizing nesting
  - clear entries and exits
  - keep housekeeping code in one place
- name indexes clearly; only one purpose
- verify normal operation under each case and termination under all conditions

Unusual control structures

- use multiple *returns* carefully for
  - enhancing readability and maintainability
  - preventing deeply nested logic
- use recursion carefully
- use *gotos* only as last resort
  - enhancing readability and maintainability

Table-driven methods

- alternative to complicated logic and inheritance structures
- key1: access
  - direct access
  - indexed access
  - stair-step access
- key2: contents

Debugging

1. understand the problem
2. fix it; avoid random guesses & corrections

- use compiler at pickiest level and fix the reported errors
- use
  - –debugging tools
  - –your brain

Self-documenting code

- poor commenting is a waste of time
- source code
  - contains most of the critical information
  - most likely to be kept current
- improve the code so that it does not need extensive comments
- comments at summary or intent level; things that the code cannot say about itself
- commenting style that is easy to maintain

Layout and style

- illuminate the logical organization
  - accuracy
  - consistency
  - readability
  - maintainability
- looking good is secondary
- follow some (any) convention consistently
- objective vs subjective preferences

Additional resources

- Code Complete, Steve McConnel, Microsoft press
- **Google C++ Style Guide**, http://code.google.com/p/google-styleguide/
- **The OpenCV Coding Style Guide,** http://code.opencv.org/projects/opencv/wiki/CodingStyleGuide