Object Oriented Design Standards

The Corelinux Consortium

Revision 1.2
Created on May 8, 2000
Last Revised on July 14, 2000

Abstract

This document describes the Object Oriented Design Standards as they are used in the `corelinux++` project. It provides a set of guidelines, rationales and standards for object oriented design.

Contents

Copyright notice

1 Object Oriented Design Standards

1.1 Scope

The Object Oriented Design Standard defines metaphors and paradigms for analysis and design of software components. The purpose is to make explicit sound software engineering design principles: to provide a standard which will be used to judge the merit of various designs as they evolve.

"Because object oriented design is relatively new to the industry, and because of the large variety of software designs that can be expressed using `C++`, there is the potential for gratuitous variations in style, conventions, and philosophy", see [Taligent, 1994]. This document minimizes these variations by providing a set of conventions for designing effective object oriented programs in `C++`.

It is expected that this document will evolve along with the `C++` language. The guidelines in this document are by no means exhaustive. Revisions will be made as ambiguities are discovered during the design and review process. This document covers the design process. For standards and guidelines on `C++` coding style, see the `C++` Coding Standards

1.1.1 Document Overview

This document is organized into ten (10) sections. Section 1 describes the scope and provides a overview of the document structure. The structure of the rest of this document is as follows:

Section 2 outlines the principals that guide the designers when they develop these standards.

Section 3 discusses the defining characteristics of good designs.

Section 4 discusses potential design problems when doing object oriented designs.

Section 5 describes design metaphors. The metaphors are explained, then standards and guidelines applicable to each metaphor, are defined.

Section 6 provides miscellaneous guidelines.

Section 7 discusses performance considerations.

Section 8 outlines the tools and methodology that will be used.

1.2 Guiding Principles

1.2.1 Design Quality in from the Start

Philip Crosby states, "Quality is defined as conformance to requirements", see [Crosby, 1976]. Bertrand Meyer says, "The general direction is clear: since correctness is the conformance of software implementations to their specifications...", see [Meyer, 1988]. Our entire software engineering process is driven by the need to build products that conform to their requirements, and to demonstrate, as early in the development cycle as possible, the quality of each component. **Quality cannot be tested into a product**; it must be built in from the start.

1.2.2 Reuse, Reuse, Reuse.

Everything in nature is constructed from a set of roughly one hundred and fifty reusable components (the elements). Each of these elements is built from a few fundamental building blocks (the sub-atomic particles). This is an extremely powerful design metaphor, and one that we will emulate throughout projects. Our goal is to build a repository of high quality, reusable components, which engineers can combine in various ways to produce new reusable components at higher and higher levels of abstraction.

Reuse is not limited to classes implemented in `C++`. This document describes design metaphors that will be used again and again. We expect to reuse the software engineering process, development tools, designs, and code.

1.2.3 Minimize implicit design decisions

`C++` does not express the full interface to a class. The following important aspects (amongst others) of a class cannot be expressed:

- Class Semantics
- Valid states for member function invocation
- Concurrency
- Storage Management

Part of the design process is to ensure that this information is carried forward from the design documents into the implementation. The implementation should, as much as possible, verify that the assumptions and design constraints are always valid.

1.2.4 Clarity and Simplicity

Booch says, "It is the task of the software development team to engineer to illusion of simplicity", see [Booch, 1994]. This applies to all levels of design, from the user interface, to the expression of client and descendant class interfaces, to code.

1.3 Characteristics of Good Design

Booch states that, "Good software architectures tend to have several attributes in common:

They are constructed in well-defined layers of abstraction, each layer representing a coherent abstraction, provided through a well-defined and controlled interface, and built upon equally well-defined and controlled facilities at lower levels of abstraction.

There is a clear separation of concerns between the interface and implementation of each layer, making it possible to change the implementation of a layer without violating the assumptions made by its clients.

The architecture is simple. Common behavior is achieved through common abstractions and mechanisms.

Good designs also reflect the clients view of their interfaces. Classes which represent the natural abstractions of a domain are the easiest way to achieve this. The interface should reflect precisely that information which is relevant to the client's problem, and no more.", see [Booch, 1994].

1.4 Design Pitfalls

1.4.1 Abuse of Member Data

Classes should access their own member data via accessor and modifier functions, just like clients. If a class provides overloadable modifier functions, but does not use them internally, then descendant implementations may not work correctly.

1.4.2 Implementation Leakage

This occurs when details of the implementation become part of the class interface. It is very easy to have this happen. Each item in the class interface should represent some valid operation on the abstraction.

1.4.3 Forcing Inheritance

Engineers who are new to object oriented design tend to overuse inheritance. The client/supplier relationship is preferred because it binds classes less tightly. A client is only coupled to the supplier via the public interface of the class. Inheritance binds classes together via the public, and protected interfaces. This is a much tighter coupling because the protected interface typically exposes part of the implementation.

1.4.4 Overuse of Multiple Inheritance

Multiple inheritance is a powerful mechanism. It is often the most elegant design solution for a given abstraction. On the other hand, multiple inheritance significantly complicates the inheritance graph. Overuse tends to make designs much harder to understand.

1.4.5 Loss of Abstraction

Classes should be abstractions of a single entity in the problem, or solution, domains. It should be easy to state what a class represents in a single sentence, without a lot of qualifying statements. All of the elements of the class interface should fit the abstraction. As class definitions evolve, the engineer must ensure that each change fits the abstraction.

1.5 Design Metaphors

1.5.1 Programming by Contract

A class must have semantics and it must have state. Member functions have constraints. The class semantics, states, and constraints are explicit during design, but tend to become implicit during implementation. *Programming by contract* attempts to carry these design-time concepts into the implementation. Special code is automatically generated to check, at run-time, that these conditions always hold. The following mechanisms are supported:

Invariant: The set of conditions that must hold whenever a class is in a valid, stable state.

Preconditions: The set of conditions that must be satisfied before a member function can be called.

Postconditions: The set of conditions that must be satisfied if a member function has completed successfully.

Never Execute: A code path that should never be executed. Used to verify that conditional logic handles all conditions.

The following standards and guidelines apply to the *programming by contract* design metaphor:

Standard 1 *Designs must explicitly state the class semantics. This includes the class invariant, member function pre and post conditions, and class state transitions.*

Rationale: Minimize implicit design decisions and assumptions.

Guideline 1 *Public and protected member functions should explicitly state preconditions and post-conditions.*

Rationale: Minimize implicit design decisions and assumptions.

Standard 2 *Derived classes must preserve the invariant state of every base class.*

Rationale: INVARIANT ... think about it.

Guideline 2 *Class implementations should have an INVARIANT clause.*

Guideline 3 *Public and protected member functions should implement pre and post conditions using REQUIRE and ENSURE.*

Rationale: Carry design decisions and assumptions into the code.

Guideline 4 *Derived classes should call BASE_INVARIANT in their INVARIANT clause.*

Rationale: Derived classes must preserve base class invariants.

1.5.2 Command/Query Interface

This metaphor enforces a distinct separation between functions and procedures. *Functions* return information derived from the state of an object, but do not change the state. A *Procedure* performs an operation that changes the object state, but returns no information.

Standard 3 *Procedures shall return nothing. If the procedure fails, exit with an exception.*

Rationale: If the precondition is satisfied, the procedure can only succeed. Any failure is, by definition, an exception.

Standard 4 *Procedures that fail shall restore the object to the state that it was in upon entry.*

Rationale: Minimize implicit design decisions. Engineers know that an object is unchanged if an exception is thrown.

Standard 5 *Multiple sequential calls to a function, with the same parameters, shall return the same result. This is not applicable to a function that returns dynamic results, such as a iterator or cursor.*

Rationale: A function should not change the state of an object, therefore the result should be reproducible.

Guideline 5 *Functions should be* const

Rationale: A function should not change the state of the object. There are cases where the function changes concrete class state, but does not affect the abstract state of a class. Bertrand Meyer has a discussion of concrete and abstract object states, see [Meyer, 1988].

Guideline 6 *Functions that fail should throw an exception.*

Rationale: If the function fails it must be an exception. There may be some scenarios where the standard exception mechanism grossly complicates the design or the code, in which case returning a status is acceptable.

Guideline 7 *Functions that must return error values should separate the error values from the valid return types.*

Rationale: Separating error return values from valid return data adds to the clarity of the code. CARE MUST BE TAKEN NOT TO ABUSE THIS MECHANISM. Functions returning error status should still check pre and post conditions.

1.5.3 Shopping List Class Interfaces

This metaphor emphasizes the fact that each member of the interface presents some necessary attribute, or operation, of an abstraction. A large number of dependencies between member functions are indicative of a procedural design. The goal of designing a class interface is that each public function represents a single, complete, and independent component of the abstraction. This applies to the public, as well as the protected interfaces.

Guideline 8 *Member functions should not have to be called in any particular order.*

Guideline 9 *No initialization should be needed after construction.*

Rationale: Functional dependencies shall be clearly documented in the design, as well as in the class header.

1.5.4 Exception Based Error Handling Model

The traditional method of handling errors, namely passing error codes up the return stack, is a major source of code complexity. It is also a source of undesirable "control coupling" between modules. In `C++` certain operations, such as construction, destruction, and assignment, do not allow for any value to be returned. This results in several different error handling schemes, depending upon what operations are involved.

The exception-based error handling model provides a well defined, robust, and extensible mechanism in which to handle errors. It provides a clean separation between the normal execution path and the error recovery code, resulting in a marked reduction of code complexity.

Standard 6 *All exception must derive from the base class:* `Exception`.

Rationale: The undisciplined use of exceptions leads to code that is less robust, and harder to maintain. By deriving from a common base class, most code needs a single catch block, i.e. `catch(ExceptionRef)`

Standard 7 *If the catch block does not correct the error, it must re-throw the exception.*

Standard 8 *Any procedure that can change the state of the object, must restore the original state in the event of an exception.*

This requires that an internal unwind to the invariant state is the responsibility of the procedure prior to throwing the exception.

Standard 9 *Every thread must have a try/catch block at the outermost level.*

Standard 10 *Assertion exceptions will always terminate the program.*

Standard 11 *Exceptions generated by a class shall be clearly documented in the design, as well as the class header.*

Rationale: Exceptions are part of the interface of a class. They must be clearly documented as the rest of the class interface.

Guideline 10 *Do not use exception handling for flow control..*

Guideline 11 *Catch blocks shall not return if they are unable to correct the problem.*

Rationale: Functions and procedures can only succeed or fail. Failure is signaled by throwing an exception. Catch blocks that do not correct the problem must not return as if the function were successful. This is a guideline because exception handlers at the topmost level (i.e. the user interface) may simply report the exception to the user and allow the program to continue running.

Guideline 12 *Create as few new exceptions as possible.*

Rationale: Each exception class adds complexity to the system. Most exceptional cases are detected by the use of invariants and assertions, or via system exceptions.

1.6 Miscellaneous Design Guidelines

1.6.1 All Interfaces Expressed Through Objects

Guideline 13 *All function calls should be in the context of some class.*

1.6.2 Required Member Functions

All classes shall explicitly code a default constructor, a copy constructor, a destructor, and an assignment operator. Minimize implicit design decisions. By declaring and implementing the automatically generated members, clearly the engineer has taken them into account and thought through their implications. The required members shall be private if the class semantics dictate that they are not needed by the class.

1.6.3 Using Friends

Guideline 14 *The use of friend functions is discouraged.*

Rationale: Friends can usually be replaced by some other mechanism. For example, an iterator can be a friend or it can be an embedded class. If alternate mechanisms that do not excessively complicate the design are available, then use those.

Standard 12 *Friend classes shall not be used.*

Rationale: If every member must have access to the implementation of some other class then the friend class should be embedded. A class that must be a friend to more than once class indicates a design problem.

1.6.4 Virtual Inheritance

Guideline 15 *The use of virtual inheritance is discouraged.*

1.6.5 Name Spaces

Standard 13 *Name spaces are now widely supported by `C++` compilers and should be used.*

Standard 14 *All "global" information shall be partitioned with namespaces.*

1.6.6 Run Time Type Information

Run time type checking are now widely supported by `C++` compilers and should be used.

1.6.7 Streams

The use of streams is dependent upon the nature of the project. If the project is a graphical application, the stream metaphor does not fit well, and should not be used. Small applications, tools, and utilities, may use streams. For example, a pretty printer or code formatter should use streams. If the choice is between streams or `printf()` functions, streams should always be chosen.

The use of streams in graphical applications shall be limited to output of error log and debug information. Streams shall always be used in place of the older 'C' stdio libraries. Streams are type safe. They can also be extended to support new types as they are added.

1.6.8 Concurrency

TO BE DONE

1.6.9 Persistent Data

TO BE DONE

1.6.10 Order of Initialization

Guideline 16 *Do not use static objects.*

Rationale: There are two well known problems with `C++` static objects. The first is with the order of execution of static object constructors, and initialization dependencies between static objects. The second has to do with throwing exceptions, where does the enclosing catch block go?

Standard 15 *Classes shall be self initializing*

Rationale: Initialization is a detail of the implementation. Should an alternative implementation be found, that doesn't require initialization, clients will have to change. If the client is responsible for initialization, then the client must be aware of order if initialization issues. If the class initializes itself, then there is no problem with order of initialization.

1.6.11 Portability

This consortium is only concerned with portability between ports of the Linux system.

Standard 16 *To enhance portability the CoreLinux++ library includes type wrappers that may need conditional compilation directives. The CoreLinux++ libraries should used the types defined in `Types.hpp`.*

Guideline 17 *Native `C++` classes (`streams`, `stl`, `string`) should be used but it is recommended that we at least wrap them. For example:*

```
class String : public string
```

Rationale: We recognize that the standards do not provide all the real world abstractions that are generally useful in most development scenarios. In our goal to provide this extended functionality it may be that we want to exploit what the standard provides and still have the means to extend it to fit the real world needs.

1.6.12 Templates

Guideline 18 *Generic types should be implemented using templates.*

Rationale: Keep the benefits of strong static type checking.

Guideline 19 *Consider breaking a template into a non-template base class for non-type specific functions and a template for the type specific functions, especially if the template will be instantiated many times, or if there are a large number of non-type dependent member functions.*

Rationale: Reduce the amount of redundant code that is generated.

2 Performance

Design for clarity and ease of understanding instead of focusing upon performance. Donald Knuth advises use to "first create solutions using excellent software engineering techniques, then, only if necessary, introduce small violations of good software engineering for efficiency", see [Knuth, 1974].

The engineer must know what tools and components are available, and when they should be used. The selection of correct data structures and algorithms will boost performance more than attempting to optimize the implementation of inappropriate ones.

Guideline 20 *Coding in assembler is the optimization technique of last resort.*

Standard 17 *If there is not empirical proof that a piece of code is a bottleneck it does not need to be optimized.*

Rationale: In most situations optimizing small portions of code result in huge performance gains. It is very difficult for the engineer to determine which sections of code are the best candidates for optimization. Use of a profiler on the other hand tells the engineer exactly where performance bottlenecks are.

3 Tools and Methodologies

We should be using Rational Rose for analysis and design. This tool implements the Booch design methodology, and accompanying notation. Engineers are expected to be completely familiar with the methodology and the notation. All aspects of the methodology will be used, with the exception of module and process diagrams. A full treatment of the methodology can be found in Booch [1994]. Dated, although RR is useful and includes UML support, it is not available on the Linux platform. Until we agree on a tool for Linux we will hand code UML.

References

Grady Booch. *Object Oriented Analysis and Design With Applications.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

Phillip B. Crosby. *Quality Is Free.* McGraw-Hill, New York, NY., 10020, 1976.

D.E. Knuth. Structured programming with goto's. *ACM Computing Surveys*, Vol 6(No. 4), December 1974.

Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

Inc. Taligent. *The Taligent Guide to Well-Mannered Object-Oriented Programs.* Taligent Inc., Cupertino, CA., 1994.