CoreLinux++ Development Process

The Corelinux Consortium

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

Abstract

This document is the `Corelinux++` development process details.

Contents

Copyright notice

1 Overview

Here is a short overview if this document:

- Overview describes the purpose of this document.
- Requirement Process describes how requirements are received, evaluated, and put into play.
- From Requirements to Analysis describes the process by which requirements are analyzed using a Object Oriented Analysis methodology.
- From Analysis to Design describes the process by which the resulting requirement analysis is moved into the Object Oriented Design phase.
- From Design to Implementation describes the process by which the resulting design is codified.

Class libraries and frameworks are often the hardest things to implement in any language (C++, Java, Eiffel) as they need to be useful across a broad range of applications.

The following document will ultimately define, in detail, the `Corelinux++` process of bringing a requirement to implementation. While it is quite typical to ”jump into the code” we believe that a thorough process, which include analysis and design, prior to coding is more satisfying to the class library developer, and usage by the application.

We believe that skipping the steps we have chosen as our process results in poorly defined behavior, difficult adjustment to change, and users who look for better solutions.

The granularity of ownership follows the granularity of this document. A team member owns a requirement to the point the System Requirement Specification (SRS) document is checked in. At this point an Analysis, Design, and Implementation task will be created in the respective Task Groups from our CoreLinux Project page in Source Forge. Any one can then decide to assign themselves the task of Analysis, then Design when Analysis is complete, and so on for Implementation.

2 Notes

Here are a few settings to help dealing with Sourceforge in Bourne shell style:

```
sf_home=userid@corelinux.sourceforge.net:/home/groups/corelinux/
export sf_home
sf_htdocs=userid@corelinux.sourceforge.net:/home/groups/corelinux/htdocs
export sf_htdocs
```

Read the Consortium [2000b] and Consortium [2000a] to help you through the development process.

3 Requirement Process

1. All things start with a Requirement request statement (e.g. `Corelinux++` should have a Thread class) regardless of the origin. If it was not originally entered into the Source Forge `Corelinux++` Requirements Forum, it will be placed there and assigned a Requirement ID which is the Requirement Forum message identifier.

2. A `Corelinux++` project member then takes ownership of the Requirement and opens it for discussion in the mailing list.

3. All Requirements will be reviewed for acceptance and issues in the CoreLinux-public mailing list.

4. If it is decided that it is not a valid requirement it will be noted with the following:

   INVALID This does not represent a requirement that fits into any current, near, or long term objectives.

   DEFERRED For whatever reason, this requirement is being deferred to a later time. The reason will be noted.

   REDIRECTED If a requirement is not correctly categorized into `libcorelinux++`, libcoreframework++, or whatever, it will be redirected to the appropriate module.

5. Upon acceptance of a requirement, the owner will need to create a Feature in the `Corelinux++` Bug Tracking database. The owner should then assign themselves to the feature , or have someone do it for them if they do not have the authority. Once the feature has an identifier, it will be noted in all related work in the CVS message. This includes Requirements, Analysis, Design, and Implementation. Once the completed implementation is checked-in, the feature can be closed.

6. Requirements that are accepted will be published to the web-site as follows:

   - A `Corelinux++` requirement document will be created in the doc/requirements directory with a file name of `reqXXXX.html` (where `XXXX` is the Requirement ID). Here is the template for this document. The following command will get you going (assuming you are in the /corelinux/doc directory):
     ```
     > cp funcreq.html requirements/reqXXXX.html
     ```
   - Using CVS, the owner will check out the doc/reqs.html requirement index file and add the entry link. From the `/corelinux/doc` directory execute:
     ```
     > cvs update reqs.html
     ```
   - After testing that the link works, the owner will use SCP to put the new and modified pages to the web site. From the `/corelinux/doc` directory execute:
     ```
     scp reqs.html $sf_htdocs/doc
     scp requirements/reqXXXX.html $sf_htdocs/doc/requirements
     ```
     where userid is your sourceforge user id and XXXX is the Requirement ID.
   - Finally, using CVS, the owner adds and commits all necessary files:
     ```
     > cd requirements > cvs add -m"YYYY Initial add of Requirement XXXX" reqXXXX.html
     > cvs commit -m"YYYY Completed Requirement Document XXXX" reqXXXX.html
     > cd ..
     > cvs commit -m"YYYY Added index entry for Requirement XXXX" reqs.html
     ```
     where YYYY is the Feature ID and XXXX is the Requirement ID

7. Once a SRS document has been checked in, the owner should notify the mailing list that it is done. This way some one can assign themselves to the next task.

4 From Requirements to Analysis

1. Before starting the analysis, assign yourself to the respective Analysis task in the project.

2. Analysis will proceed starting with Use-Case Modeling. In use-case modeling, the system is looked upon as a "black box" that provides use cases. How the system does this, how the use cases are implemented, and how they work internally is not important. In fact, when the use-case modeling is done early in the project, the developers have no idea how the use cases will be implemented

3. The Analysis model will be used when moving into Design. To minimize the amount of maintenance between Analysis and Design, as there is bound to be iterations from requirements, to analysis and the design, we will use the same model throughout.

4. Models are to be managed as follows:

   - A model will be created in the doc/models directory using the MagicDraw UML tool. This model is to be named using the Requirement ID and short description name (XXXX-NNNNN). The model is to be saved using the MagicDraw UML format. This will create two (2) files; XXXX-NNNNN.mdf and XXXX-NNNNN.mdr.
   - We will only use the default settings for the model. This will keep it consistent.
   - Once the model is created it is to be added and committed to CVS from the /core-linux/doc/models directory:

     ```
     cvs add -m"YYYYY Initial UML for XXXX-ZZZZ" xxxx-nnnnn.mdf xxxx-nnnnn.mdr

     cvs commit -m"YYYYY UML for XXXX-ZZZZ" xxxx-nnnnn.mdf xxxx-nnnnn.mdr
     ```

     where YYYYY is the Feature ID, XXXX is the Functional Requirement ID and ZZZZ describes the requirement.

     for example:

     ```
     cvs add -m"100547 Initial UML for 2872-Adapter Pattern" 2872-Adapter.mdf 2872-Adapter
     cvs commit -m"100547 UML for 2872-Adapter Pattern" 2872-Adapter.mdf 2872-Adapter.mdr
     ```

5. Use case objectives are

   - To decide and describe the functional requirements of the system, resulting in an agreement between the team members.
   - To give a clear and consistent description of what the system should do, so that the model is used throughout the development process to communicate to all developers those requirements, and to provide the basis for further design modeling that delivers the requested functionality.
   - To provide a basis for performing system tests that verify the system. For example, asking, does the final system actually perform the functionality initially requested.
   - To provide the ability to trace functional requirements into actual cases and operations in the system. To simplify changes and extensions to the system by altering the use-case model and then tracing the use cases affected into the system design and implementation.

6. The actual work required to create a use-case model involves defining the system, finding the actors and the use cases, describing the use cases, defining the relationship between use cases, and finally validating the model.

7. The use-case model consists of use-case diagrams showing the actors, the use cases, and their relationships. These diagrams give an overview of the model, but the actual descriptions of the use cases are typically textual. Both are important!

8. Class diagrams can be created to capture the classes in the use-case model. At this point, the classes are there to support the sequence or collaboration diagrams only.

9. The use-case model will be used to realize the use case. The UML principles for realizing use cases are:

   - A use case is realized in a collaboration: A collaboration shows an internal implementation-dependent solution of a use case in terms of classes/objects and their relationship (called the context of the collaboration) and their interactions to achieve the desired functionality (called the interaction of the collaboration).

- A collaboration is represented in UML as a number of diagrams showing both the context and the interaction between the participants in their collaboration. Participating in a collaboration are a number of classes (and in a collaboration instance: objects). The diagrams are collaboration, sequence, and activity. The type of diagram to use to give a complete picture of the collaboration depends on the actual case. In some cases, one collaboration diagram may be sufficient, in other cases, a combination of different diagrams may be necessary.
- A scenario is an instance of a use case or a collaboration. The scenario is a specific operation path (a specific flow of events) that represents a specific instantiation of the use case. When a scenario is viewed as a use case, only the external behavior toward the actor is described. When a scenario is viewed as an instance of the collaboration, the internal implementation of the involved classes, their operations, and their communication is described.

10. With the exception of the Class Diagram noted above, the Sequence, Collaboration, Activity, and nested Use Case diagrams should be created in the context of the Use Case they are intended for. To do this in MagicDraw UML you open the specification of the use case in the main diagram, click the Diagram tab and create from there.

11. All actors, classes, and diagram specifications are to be documented.

12. MagicDraw UML provides the ability to generate various reports and images. For analysis our primary concern is the Use-Case report and any use-case, collaboration, sequence, activity, and class diagrams.

13. To create the Use-Case report:

    - Make the Use Case diagram the active window
    - Select `File->Report`
    - In the General Options tab (using the Adapter example):

      ```
      Report File Name : /corelinux/doc/analysis/2872uc.html
      Report Title     : Adapter Use Case Report
      ```

    - In the Report Options tab :

      ```
      General Report Options (all checked)
      Use case reports (all checked)
      Class reports (just Include class report)
      Model dictionary (all checked)
      ```

14. To create the associated diagram images:

    - Make the diagram the active window but do not have any specific object selected
    - Select `File->Save` Diagram As Image
    - In the Save Dialog (using the Adapter Use Case diagram as an example):

      ```
      Look in         : analysis
      Files of Type   : *.png
      ```

File Name : 2872uc-AdapterMain
This will generate /corelinux/doc/analysis/2872uc-AdapterMain.png

- Using Adapter Instantiate Sequence as example:

  ```
  Look in         : analysis
  Files of Type   : *.png
  File Name       : 2872sq-InstantiateAdapter
  ```

  This will generate /corelinux/doc/analysis/2872sq-InstantiateAdapter.png
- For other diagrams types use (where XXXX is Requirement ID and NNNNN is brief description):

XXXXuc-NNNNN	Use Case Diagram
XXXXsq-NNNNN	Sequence Diagram
XXXXst-NNNNN	State Diagram
XXXXco-NNNNN	Collaboration Diagram
XXXXac-NNNNN	Activity Diagram
XXXXcl-NNNNN	Class Diagram

15. Because the MagicDraw UML tool does not generate links to the diagrams, you will need to add a html page in the analysis directory which references the diagrams generated. If need be, decorate the page with informative text. The new page should be named XXXX.html (where XXXX is the Requirement ID), added to CVS and committed.

    ```
    cvs add -m"YYYYY Initial Analysis Report and Diagrams for XXXX-ZZZZ" xxxx*.*

    cvs commit -m"YYYYY Analysis Report and Diagrams for XXXX-ZZZZ" xxxx*.*
    ```

    for example:

    ```
    cvs add -m"100547 Initial Analysis Report and Diagrams for 2872-Adapter Pattern"
    2872*.*

    cvs commit -m"100547 Analysis Report and Diagrams for 2872-Adapter Pattern" 2872*.*
    ```

16. The Functional Requirement Document (in the requirements directory) needs to be updated with links to the report and image pages in the Analysis References line. If need be, references to other specifications are added to the Cross References line. Don't forget to commit the change to CVS as well

    ```
    cvs commit -m"YYYYY Updated for Analysis links" reqxxxx.html
    ```

17. Finally, all of the documentation is to be loaded up to the web page:

    ```
    [/corelinux/doc]$ cd analysis
    ```

```
[/corelinux/doc/analysis]$ scp XXXX*.*  $sf_htdocs/doc/analysis

[/corelinux/doc/analysis]$ cd ../requirements

[/corelinux/doc/requirements]$ scp reqXXXX.html $sf_htdocs/doc/requirements
```

5 From Analysis to Design

1. Once an Analysis has been checked in it is available for design. Before starting design, assign yourself as the Design Task owner.
2. If not already done in the Analysis phase, analysis classes will be divided into functional packages.
3. Additional technical classes are added.
4. Concurrency needs are identified and modeled through active classes, asynchronous messages, and synchronization techniques for handling shared (if applicable) resources.
5. As of the current version of MagicDraw UML (3.5) **exceptions** are specified in the Language Settings of the function specification dialog. A **strong** emphasis is placed on exceptions and faults in the design. This includes both normal and abnormal where:

   Normal Those that can be anticipated in the course of performing the functions.

   Abnormal Those that can't be anticipated and must be handled by some generic exception mechanisms.
6. A **strong** emphasis is placed on constraints. A constraint is a restriction on an element that limits the usage of the element or the semantics (meaning) of the element. Constraints are one way of enforcing the "contract" by which the objects in a live system may or may not interact. For example, if class A is constrained to having at most five (5) elements in a collection data member, a pre-condition constraint would be put on the *addElement(ElementCref)* method:

   ```
       pre: = theCollection.size <= 5
   ```
7. The dynamic behavior of the design is emphasized. This is done through class, collaboration, activity, sequence, and state diagrams. The benefit is that it will reduce the implementation time by having a clear understanding of what the code should be doing.
8. All class methods and members are to be ordered in the diagrams following our standard, basically:

   ```
   For members:
     Should be protected or private only.

     For association, aggregates, containment:
       At the very least the cardinality should be specified
   ```

```
as well as the role names ( just one if it is not bi-directional ).

 For comprehension, the relationship itself
 should have a name ( typically a verb ).

 For operations:

   Public
     constructors,
     virtual destructor,
     operator overloads,
     accessors,
     mutators.

  Protected
     same order

  Private
    ditto
```

Indicate virtual or constant where applicable.

9. Return values from methods and arguments will use the `Corelinux++` define types (`xxRef, xxPtr, xxCref, xxCptr, etc.`).
10. Methods taking no arguments will have a void type with no name assigned.
11. Models and documents (Class Report and Diagrams for Web) are to be managed as follows:
    - The model that was created in the `corelinux/doc/models` directory during analysis will be used for design. This model is named using the Requirement ID and short description name (XXXX-NNNNN).
    - Before beginning the design work in the model, make sure that the project settings are changed.

      ```
      Load the model
       Click Options
       Click Project...
       In the Project Options dialog, select the Code engineering folder
       Set Default Language to C++
       Now select the Styles Folder and fully expand the default style
       Select the ClassView entry
       Change Attribute and Operation sorting to No Sorting
       Click OK
       Save the project
      ```
    - When the design is complete, commit to CVS from the /corelinux/doc/models directory:
      `cvs commit -m"YYYYY UML Design for XXXX-ZZZZ" xxxx-nnnnn.mdf xxxx-nnnnn.mdr`
      where YYYYY is the Feature ID, XXXX is the Functional Requirement ID and ZZZZ describes the requirement.

for example:

```
cvs commit -m"100547 UML Design for 2872-Adapter Pattern" 2872-Adapter.mdf
2872-Adapter.mdr
```

12. During this phase our primary concern for web documentation is the Class report and any class, collaboration, state, sequence, and activity diagrams that support the design.

13. To create the Class report:

    - Make the Class diagram the active window
    - Select `File->Report`
    - In the General Options tab (using the Adapter example):

      ```
      Report File Name : /corelinux/doc/design/2872uc.html
      Report Title     : Adapter Class Report
      ```

    - `In the Report Options tab :`

      ```
      General Report Options (all checked)
      Use case reports (NOT checked)
      Class reports (all checked except Public Members Only)
      Model dictionary (all checked)
      ```

14. To create the associated diagram images:

    - Make the diagram the active window but do not have any specific object selected
    - Select `File->Save Diagram As Image`
    - In the Save Dialog (using the Adapter Classes diagram as an example):

      ```
      Look in         : design
      Files of Type   : *.png
      File Name       : 2872cl-AdapterClasses
      ```

      This will generate `/corelinux/doc/design/2872cl-AdapterClasses.png`

15. Use these naming conventions for other diagrams.

16. Because the MagicDraw UML tool does not generate links to the diagrams, you will need to add a html page in the design directory which references the diagrams generated. If need be, decorate the page with informative text. The new page should be named XXXX.html (where XXXX is the Requirement ID), added to CVS and committed.

    ```
    cvs add -m"YYYYY Initial Design Report and Diagrams for XXXX-ZZZZ" xxxx*.*

    cvs commit -m"YYYYY Design Report and Diagrams for XXXX-ZZZZ" xxxx*.*
    ```

    for example:

    ```
    cvs add -m"100547 Initial Design Report and Diagrams for 2872-Adapter Pattern"
    2872*.*

    cvs commit -m"100547 Design Report and Diagrams for 2872-Adapter Pattern" 2872*.*
    ```

17. The Functional Requirement Document needs to be updated with links to the report and image pages in the Design References line. If need be, references to other specifications are added to the Cross References line. Don't forget to commit the change to CVS as well.

    ```
    cvs commit -m"YYYYY Updated for Design links" reqxxxx.html
    ```

18. Finally, all of the documentation is to be loaded up to the web page:

    ```
    [/corelinux/doc]$ cd /corelinux/doc/design

    [/corelinux/doc]$ scp XXXX*.* $sf_htdocs/doc/design

    [/corelinux/doc]$ cd ../requirements

    [/corelinux/doc]$ scp reqXXXX.html $sf_htdocs/doc/requirements
    ```

6 From Design To Implementation

1. Review the Code Standards or any of the `Corelinux++` libcorelinux class declarations for clarification. This document has been updated to include details on the declaration comments that are formatted for HTML generation.

2. Before starting implementation, assign yourself as the task owner in the Implementation Task list.

3. **DO NOT CHECK IN PARTIAL IMPLEMENTATIONS OR BROKEN CODE!** Unless what you are working on is part of a joint effort (class dependencies, etc.), there is no reason to do so. Of course there may be bugs, but the general idea is that it works when it is committed.

4. If you **must** check in exemption code (see above) let others know why, if it isn't already known, you are about to do this (mailing list is best bet).

5. For the libcorelinux++ class library, the following steps detail one approach (yours may be different but the results are the same):

    - Create class declarations (`ClassName.hpp`) in **/corelinux/include**
    - Make sure that the `DEFINE_CLASS` and class declaration are wrapped in a **namespace corelinux** block
    - If class is to be included all the time (like Exception), add forward declaration and include to **/corelinux/include/Common.hpp**
    - Create class definition (`ClassName.cpp`) in **/corelinux/src/classlibs/corelinux**
    - Make sure that the implementation is wrapped in a **namespace corelinux** block
    - Add class object macro and dependencies to **/corelinux/src/classlibs/corelinux/makefile**
    - Create a directory for example code off of **/corelinux/src/testdrivers**
    - Create the **dirs.inc**, **project.inc**, and **makefile** (support files).

- If needed, add include directory off of new example code directory
- Create source and optional header files
- Compile, test, stress, make it robust!
- Add all and clean targets for new sample in **/corelinux/src/testdrivers/makefile** and test
- From `/corelinux` run `make clean` and `make all` to test
- Add any new directories, support, and source files (hpp and cpp) to CVS and commit
- Commit any updated source or support files to CVS
- Close the **Feature** in the `Corelinux++` project
- Post a message to the mailing list, knowledge of availability is valuable

6. If there is to be a segregation of header files for the main libcorelinux++ shared library (`libcorelinux++.so`):

   - Add a new directory under **/corelinux/include**
   - Add declaration header new directory
   - Add path to new directory in **/corelinux/src/classlibs/corelinux/dirs.inc**
   - Build and run all testdrivers, correct as necessary
   - Create and commit in CVS as needed
   - Close the **Feature** in the `Corelinux++` project
   - Post a message to the mailing list

7. If there is to be a modularization of shared libraries but still part of `libcorelinux++` package

   - Follow steps for declaration headers
   - Add a new directory under **/corelinux/src/classlibs**
   - Add support and implementation files to new directory
   - Add all and clean targets for new library in **/corelinux/src/classlibs/makefile** and test
   - Build and run all testdrivers, correct as necessary
   - Create and commit in CVS as needed
   - Close the **Feature** in the `Corelinux++` project
   - Post a message to the mailing list

8. For the libcoreframework++ :

   - Base header files for framework in **/corelinux/include/frameworks**
   - Implementation and support files in **/corelinux/src/classlibs/frameworks**
   - Segregation and modularization resolved as in steps above, relative pathing considered

9. When in doubt, post an e-mail to the mailing list.

References

The Corelinux Consortium. *The Corelinux C++ Coding Standards.* The Corelinux Consortium, 1.3 edition, May 2000a. http://corelinux.sourceforge.net/cppstnd.php.

The Corelinux Consortium. *The Corelinux Object Oriented Design Standards.* The Corelinux Consortium, 1.3 edition, May 2000b.