

Operational Concept Document

Pluggable Code Repository

Jim Fawcett

CSE681-OnLine

Software Modeling and Analysis

June 3, 2017

Contents

Foreword.....	3
Executive Summary.....	4
Introduction	5
Concept	7
Uses.....	9
Structure	12
Tasks and Activities	14
Design Issues	15
Risks	17
Conclusions	18
Appendix – Repository Classes	19

Foreword

This sample document is an Operational Concept Document (OCD) that describes the concept used to design and implement a Pluggable Repository. Explanation of the purposes for development of the Repository and methods used for its design are part of the OCD document and will be explained in the body.

This particular document also has two purposes, to describe the concept of the Sample Projects #1, #2, #3, and #4, but also to provide commentary on how you will build OCDs for your projects. When we describe the project purposes we will present both the purposes for a product that will serve an engineering organization, and also the academic purposes for these projects.

The results of our project efforts in this course are prototypes for a finished product. The prototypes we develop will be essentially functionally complete, but won't have the extensive polishing and testing¹ that a commercial product would have. We stop where the academic merit of additional work is no longer worth its cost in time and focus.

The four sample projects:

1. Operational Concept Document for a Pluggable Repository
2. Local Repository
3. Communication infrastructure and Graphical User Interface
4. Remote Pluggable Repository

Are all focused on creating a final Pluggable Repository prototype. You might think of each of them as an agile-programming sprint, each adding some significant functionality to what will become the final product.

Occasionally, in the document body, we will provide commentary that is intended for instruction, not for directly describing project concept. We place these comments in a callout box like that shown below.

Instructional comments that are not part of the OCD proper will be formatted like this.

¹ It is not unusual for polishing and final testing to take almost as much time and effort as the development of its functionality.

Executive Summary

The Pluggable Repository is intended to be a customizable storage and control mechanism for packages in an evolving software baseline. It provides dependency-based storage and retrieval, and supports multiple versions for each stored package. Furthermore, it provides categorized storage for packages that support effective search and browsing in a large baseline that may contain thousands of packages.

The Repository, at startup, loads a set of policy components that provide most of its functionality. We customize Repository operations by provided one or more customized policy libraries.

Repository clients support baseline browsing through package dependency trees, and can extract an entire dependency tree's contents by naming the root package of the tree. The browsing experience is enhanced by viewing metadata for each of the viewed packages.

The primary development risk is concerned with ease of use. The Repository and its Clients have to be designed to work smoothly together and hide a lot of the internal Repository operations from users.

Its size and complexity are manageable. We estimate that its implementation will need little more than one package for each of: the Repository proper, each of its individual policies, communication infrastructure, and the Client graphical user interface.

Introduction

The acronym OCD stands for Operational Concept Document. Its purpose is to make you think critically about the architecture, design, and implementation of a project before committing to code. It also serves to publish your concept to the development team, which for this course is you.

One focus area for this course is understanding how to structure and implement big software systems. By big we mean systems that may consist of hundreds or even thousands of packages and perhaps several million lines of code.

To successfully implement big systems, we need to partition code into relatively small parts that are easy to understand and easy to test. This is important because we need to thoroughly test each of the parts before inserting them into the project's software baseline². In order to do that effectively we need to understand each of the packages³, their modifications, and dependencies on other packages.

As new parts are added to the baseline and as we make changes to fix latent errors or performance problems in existing packages, we will be creating new versions of existing packages as well as creating new packages. When we make changes we will run test sequences for those parts, all the parts that depend on the changed parts and, occasionally, for the entire baseline.

Because there are so many packages, some with several versions, we need a Repository to store the baseline and some semi-automated processes for checking in packages, for versioning, and for making queries on the Repository contents. These sample projects all focus on the structure, design, and implementation of a Repository for storing code baselines.

One obvious question is why would we do this since there are many well-established code control systems, e.g., git, Subversion, etc.?

To implement a system for continuously integrating new code into complex systems, our goals are to detect breakage as soon as feasible after submitting a change to the baseline. That means that we want to test, not only the changes, but also all packages that directly depend on the changes. The most commonly used code control systems do not make the needed dependency information available, and sometimes make it awkward to extract just the packages needed for an integration test. Our Repository will be designed to provide fine-grained, dependency-aware, package management, to support continuous integration testing.

² A software baseline is the set of all packages that are part of the current state of project development. The baseline does not contain code in the possession of individual developers until they have been submitted to the Repository for control.

³ A package is a source code file that includes initial comments with prologue, a description of its operations, files required to build the package, and a maintenance history as well as code definitions.

In this and following projects we will be creating a Repository - a semi-automated storage mechanism that provides pluggable policies for:

1. File management
2. Version control
3. Package ownership
4. Checkin and Checkout
5. Package browsing

The term "pluggable" means that we can substitute one version of a policy with another without requiring any other changes to the Repository code. We will see that means that policies will need to be implemented as components - a software part that has an interface and object factory⁴.

For this project, we develop, and document here, the concept for creating a Repository that we then implement in Sample Projects #2, #3, and #4.

⁴ An object factory creates instances of concrete classes that derive from some interface. Instances are returned to the caller bound to an interface reference.

Concept

The Pluggable Repository is composed of an application that loads libraries, at runtime, that provide all of the major functionalities required to manage a code baseline. This structure allows a software engineering organization to customize the repository for its own style of product management. Organizations usually have a process they use for building and managing a baseline which may differ from the way others do that. The Repository has a library for each of its major activities, allowing customization by simply using a different existing library or creating a new library for any of its functions.

Another part of the concept is the use of dependency-based storage. That allows a user to extract a package and all the other packages it depends on simply by naming the root package of that dependency subtree. This makes frequent building and testing of parts of the baseline much easier because we extract all of the parts needed for the build with one extraction request, without getting a lot of packages that are not needed.

The Repository represents each package with an XML metadata file. The metadata has a reference to its primary source code file, and also has references to the metadata files for source code on which it depends. Essentially, the repository treats each metadata file as a package and its source code reference is simply the implementation of that package.

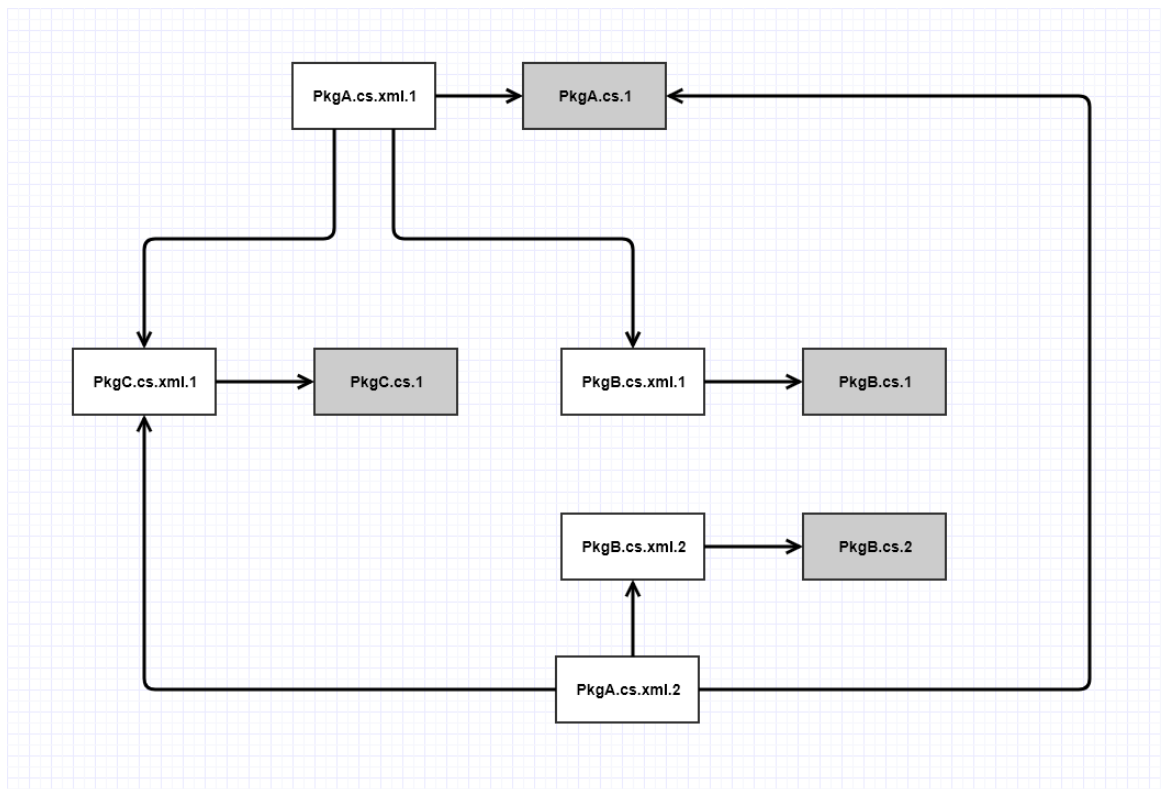


Figure 1. – Metadata-based dependency relationships

Each metadata file is a node in a virtual dependency forest. Each tree in the forest is a set of packages that are related through dependencies. Usually, each dependency tree contains the code to build a single process, e.g., a client or a server. The entire project⁵ may consist of several trees, e.g., client and server. The Remote Pluggable Repository is a project that has both repository server process and repository client process, so, the project's code has a forest of two trees, one for each process.

Note that there is no runtime data structure holding dependency tree information. The repository navigates a tree by loading and analyzing each metadata file to find dependency references in the metadata. Should that turn out to be a performance issue, we could always build an in-memory structure to hold this information, probably in a Dictionary, where each key is a package and the associated value is a list of child packages. Until navigation is shown to be a performance issue, we elect to avoid adding that bit of additional functionality, just to keep the implementation as simple as we can.

⁵ We use the word “project” in two ways. Here, the word “project” refers to all the code to build an application, like this Remote Pluggable Repository project, that contains several processes. Each process will be built as a Visual Studio project, embedded in a Visual Studio solution.

Uses

There are two kinds of uses we need to address: uses of a finished version of the final project, and instructional uses for each of our four projects.

Uses of the finished product are defined by its use in a software development organization:

1. Developer's daily activities:

Developers use the repository more frequently than any other user group. The repository will be part of their daily work-flow and it is critically important that the repository operations make their work more efficient, not less. It must be easy to check in packages, extract package dependency graphs, and view repository contents. Developers spend a lot of time browsing through their own code and that written by others on which they depend or which they need to support.

Impact on design:

Navigation through large sets of packages should be relatively painless, and at any point in the browsing process the user should be able to examine the code and package details. Because the baseline may contain thousands of packages, it will be important to be able to group packages into sensible categories and allow navigation within the packages of each group. Dependency relationships between packages are an important aspect of a system's design, and it is important that developers can easily navigate through a dependency graph. For that we will need to construct metadata for each package that describes its dependencies on other packages⁶.

2. Quality Assurance work flow:

QA personnel extract large parts of a baseline to create builds for regression testing. They also need to run tools, across the entire baseline, that analyze conformance to code standards and look for structural defects.

Impact on design:

Because of the size of baselines we expect to manage in the Repository, it is important that scanning the baseline and extracting packages for builds be made as efficient as we can, within the constraints of cost and complexity. It would also be very useful to provide a tool interface that allows QA to automatically schedule and report the results of code quality scans.

3. Manager's need for progress information:

A Program Manager is charged with delivering a large complex system that meets its customer obligations, satisfies code quality standards expected by the developing organization, and meets the allocated program schedule. To do that a manager reviews QA reports, and looks at code commit and testing activity. For example, just before a customer review, a manager would expect to see commit activity at a low level (very little new code entering the baseline) and a moderate amount of regression testing (QA ensuring a stable demonstration build). However, any hot-spots of developer

⁶ Metadata could be encoded, for each package, in a database, perhaps using MongoDB. Alternately, metadata can be encoded into XML files, one for each package. To make deployment simple, we will elect to use separate XML files.

commit and testing activity may indicate some part of the baseline hasn't reached the level of maturity needed for a customer review.

Impact on design:

It will be important to provide logging of commit activity and tools to extract summaries of that information on a scheduled basis. Engineering organizations that develop large code baselines will need to support continuous integration testing in some form of test harness. The test harness would also provide logging information for managerial consumption. Our focus here is just on the Repository, but we should attempt to implement a design that will easily integrate with a separate Test Harness facility. That implies a programmatic extraction process, acceptance and storage of testing logs, and use of common communication protocols.

4. Customer Review and Code Maintenance:

A Program Manager needs to supply to his customers, summary level progress information at each review. This is likely to be a subset of the information the manager uses to access weekly progress. When the product is delivered, it needs to be packaged for deployment. One very effective way to deploy a project is to provide the customer with a Repository with a subset of functionality suitable for code maintenance, assuming the customer's engineering staff will be maintaining the product.

Impact on design:

There should be no additional impact on design to satisfy the need for progress information for customer reviews. That is already provided for managerial use. However, if we deploy the repository on product delivery, we will want to ensure that proprietary functionality is not part of that delivery. Since the concept already supports Repository use of Pluggins, we can simply configure any proprietary functionality as pluggins that are not part of the deployment package.

Instructional uses have different actors, the developing student and instructor, with different needs. Each student needs to thoroughly test each part of the developing project and, at the end, demonstrate each requirement to the instructor. The instructor will run each project and look at its code implementation for evaluation.

1. Student project development:

The student developer needs to partition the repository functionality into relatively small, simple parts, each of which have built-in testing functionality to demonstrate successful development.

Impact on design:

Each repository part will need to provide test functions that can be run as part of stand-alone testing of the part, and later as one link in a chain of test processing.

2. Instructor evaluation:

Students are responsible for demonstrating that they meet each of the requirements in the Project Statement, e.g., Sample Project #2. They will need to log test results to the console, being careful to provide information, not just raw data. That is, the outputs should respond to each Project Statement requirement, using language that is easy to understand, and with data that is as brief as possible, while still demonstrating all parts of the requirement.

Impact on design:

Logging facilities should be configured to allow turning on or off demonstration outputs. Ideally, this logging will be a part of those facilities used by managers in the final product.

Structure

The Pluggable Repository system includes a Repository process with multiple Client processes to support user actions on the Repository contents. Client processing and message-passing communication are parts of Sample Project #3 and won't be addressed in this document. In Sample Project #4, this document will be revised to add additional concepts for Client and communication facilities.

The Repository structure for Sample Project #2 is illustrated in the package diagram, below.

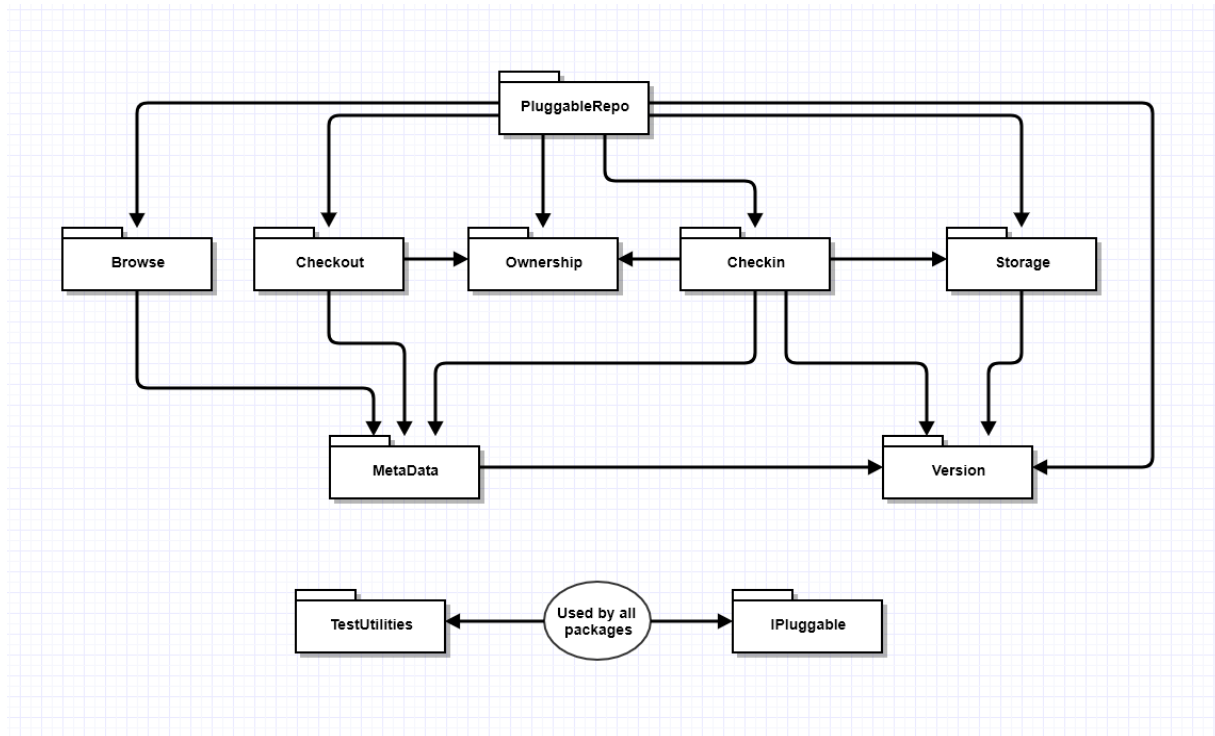


Figure 2. – Pluggable Repository Structure for Project #2

This diagram contains the Pluggable Repository top-level packages and illustrates their calling dependencies⁷.

1. PluggableRepo package:

This is the executive package, responsible for loading libraries for all of the pluggins functionality, and establishing run-time sequencing of Repository operations⁸.

⁷ For Sample Project #4 we will add packages needed to implement communication, message-building and dispatching, and to build the Client Graphical User Interface.

⁸ In Sample Project #4, the Repository operations are determined by user actions in the Client. However, for Sample Project #2, the operations simply support demonstration of the Repository functionality, so the executive will sequence through self-tests for each of its loaded libraries.

2. Browse package:

The operations of this package provide support for Client access using a Graphical User Interface, mediated by message-passing communication between Client and Repository. In this Project #2 we will simply provide a loadable shell that does almost nothing, deferring development of its operations to Project #4.

3. Checkout Package:

Checkout provides the ability to extract a dependency tree by simply naming the root package of the tree. We will implement a single-owner policy for each package, so there is no additional functionality required for checkout. It is simply a dependency tree extractor.

4. Ownership Package:

The ownership policy determines which users are allowed to commit new versions of a package to the Repository contents. For this implementation we use a simple single ownership policy: each package has a single owner and only that owner can commit new versions. Note that, under this policy any user may checkout a package. They simply cannot modify it and commit the changed code, unless they are the package owner.

5. Checkin Package:

Checkin is responsible for accepting for storage a user supplied file, building metadata from user supplied information, and moving the file and metadata into a specified Repository category folder. Checkins are either open or closed. Contents of an open checkin may be changed at any time without changing the version number. However, once a checkin is closed, its contents become immutable. Any text changes to a closed checkin file may only be effected by a new checkin. Closed-ness is indicated by a property in the metadata for the checkin.

6. Storage Package:

Storage handles all of the file copying and moving between a staging area and Repository category folders. It also provides information, on request, about the contents of each category.

7. Version Package:

This package is used by checkin and storage to manage versioning of Repository contents. Our concept for this is to provide integer version numbers for everything in the Repository, including metadata XML files, and source code files. We do that by modifying the name of the files with a trailing version number, e.g., someFile.cs.3 or someMetadata.XML.2.

8. MetaData Package:

The Repository views a metadata file as representing a package. The metadata contains a reference to the primary file, and to each of its package dependencies. It also contains descriptive information about the primary file for use in browsing repository contents.

Tasks and Activities

The tasks the Local Repository of Sample Project #2 will execute are quite different from those of the Remote Repository of Sample Project #4. For Project #2 we won't have a graphical user interface and so the Repository can only execute programmed activities, and those are all focused on demonstrating that all of the major Repository pluggable parts work as expected.

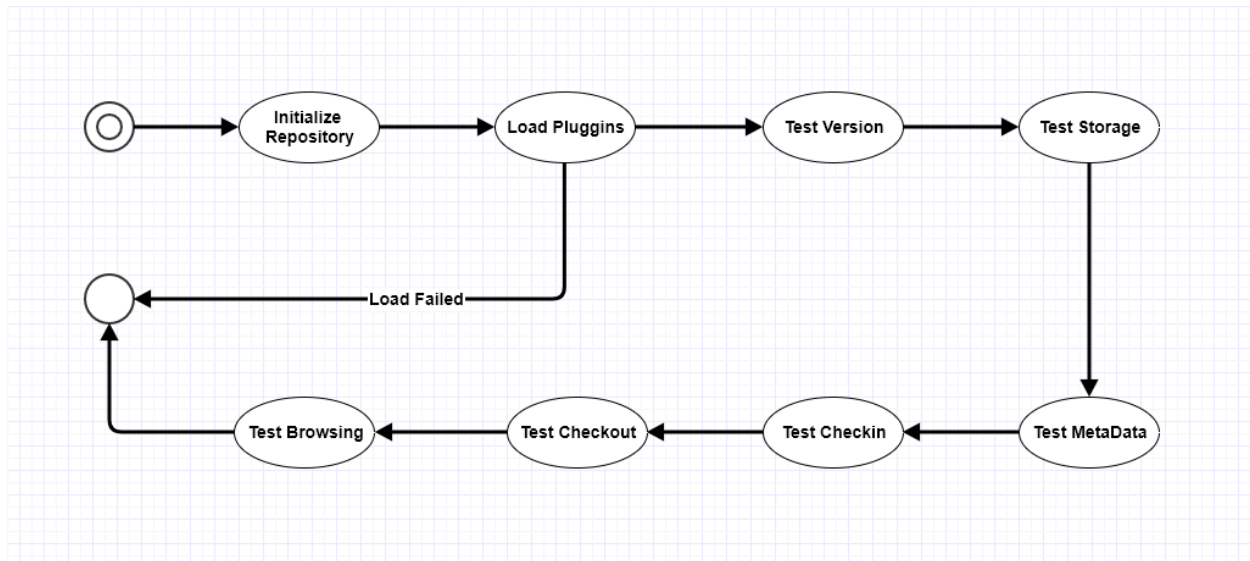


Figure 3. – Pluggable Repository Activities

For Project #2, the Repository attempts to load its required pluggins, and, if that is successful, executes a series of fairly detailed tests on each part. Each of these test will become component self-tests in the final Sample Project #4⁹.

If you think of the implementation of each of our four Sample Projects as an agile-programming sprint, then this Project #2 is the first sprint that builds the core functionality of the Repository, with later sprints focused on implementing communication and user interface facilities, followed by a concluding sprint to finish a solid working Remote Repository. For a commercial product, we would follow with one final sprint to polish and thoroughly test the product for release. We omit that final phase because of its diminishing academic value.

⁹ The activities of the Remote Repository will be quite different. There, the activities will be driven by actions taken in a user's Repository Client application.

Design Issues

The Pluggable Repository has a simple operational model, e.g., store, and make accessible, source code packages and their dependencies. However, its implementation is relatively complex and there are a number of design decisions we need to think critically about before committing to code:

1. Versioning:

Within the Repository, we've elected to represent versions by appending a version number to each file name, and use those versioned names in all metadata references. However, when building binary versions of the code we must strip off the version numbers. That is easy enough to do. However, for users to scan dependency trees, version numbers are needed.

Proposed solution:

Support browsing in the repository and download files on demand for viewing, or build a cache of versioned files on the client for browsing there. Both options make sense and we will make a choice based on performance and complexity.

2. Performance:

There are two activities that may have a significant impact on performance: browsing repository contents, and the related scanning of dependency trees for extraction. Both relate to the use of metadata to represent packages and dependencies.

Proposed solution:

We will elect to use on-demand loading of metadata files but monitor the resulting performance. If that turns out to be an issue, we can build a run-time Dictionary of packages and dependency information, or even to use an existing storage mechanism like MongoDB¹⁰.

3. Ease of use:

If the Repository isn't easy to use, developer productivity will suffer, and we wasted a lot of time and effort to build it.

Proposed Solution:

To make the Repository easy to use we need to make its operations reliable and as simple as is reasonably possible. We will do that by making the Repository parts as simple as we can, and by extensively testing them as we build. Also, the client interface will act as a mediator, hiding some of the complexity of interacting with the Repository. Here, in Figure 4., below, is a mock-up of two Client views that show how we intend to support productivity. On the left is the home view, with four tabs that support navigation, code views, checkin, and Message flow. Users will navigate by double clicking on a package reference in the top view, which brings up a view of the package primary code and its metadata. We move up or down the dependency tree using the buttons and view in the Dependencies list.

¹⁰ We would probably elect to build our own Dictionary rather than use an existing database. That makes deployment simpler and allows us to build only that functionality explicitly needed for our purposes.

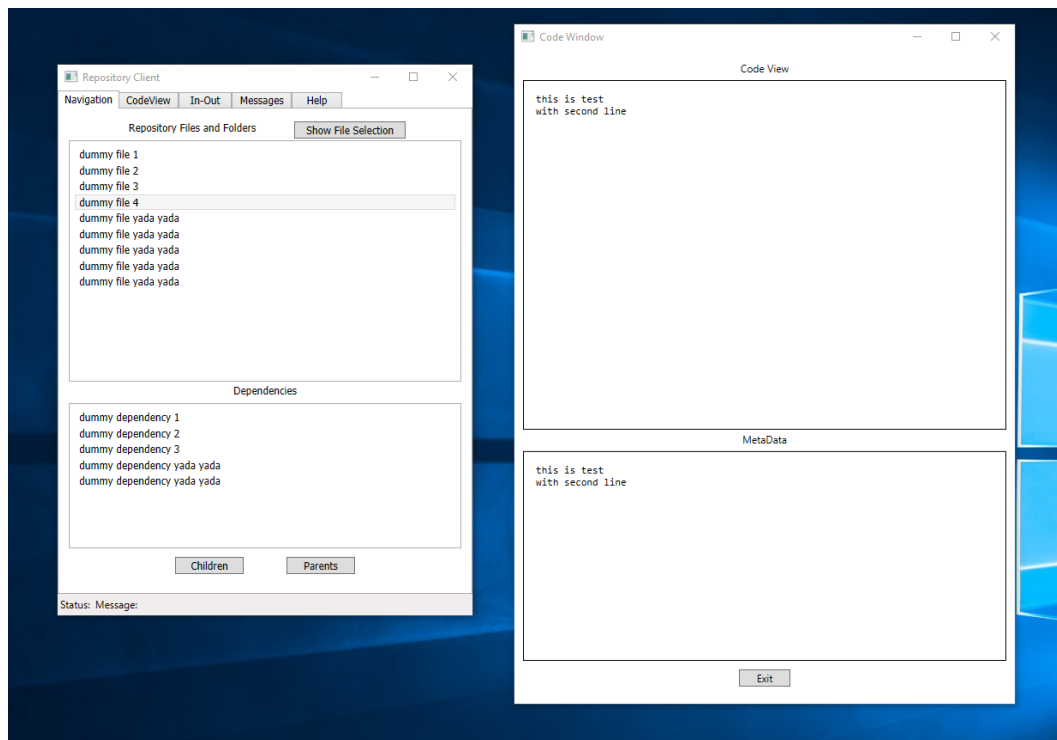


Figure 4. – Mock-up of two Client views.

4. Traceability:

Managers need the ability to view commit histories and timelines, to assess progress of the current project.

Proposed solution:

If we elected to save metadata information in an auxiliary database, then it would be easy to capture commit events and query for commit histories. We will elect, instead, to extract that information, on demand, from the metadata itself. Each metadata file contains the time of creation, and it will be relatively easy to build a history scanner to extract views into that information. Note that gathering this information doesn't happen very often, perhaps once a day, so performance isn't really an issue for this operation.

5. Size of the Repository contents:

A production Repository will need to hold, perhaps, thousands of packages. How will a user find a particular package?

Solution:

We will divide the Repository storage into a number of categories, each of which holds only a modest number of packages, supporting search and retrieval operations. Each category will be implemented with a sub-folder within the storage.

Risks

We want to identify risks associated with the building and use of our Pluggable Repository. We do that to minimize risk by design, and to consider cost-benefit of building the Repository.

1. Productivity:

The impact on productivity of using the Repository is the dominant risk factor. If we don't design well and implement carefully, the Repository could be clumsy to use and might reduce productivity rather than improve it.

Mitigation:

Use of pluggable components makes it easy to replace any part of its functionality should it prove to make operations too complex and difficult to use. We should allocate schedule time to polish the Client/Repository operations to improve their ease of use.

2. Security:

The Repository will hold contents with proprietary value. We want to protect that from unauthorized access, to prevent theft and malicious changes by competitors.

Mitigation:

We will elect to defer security to the perimeter defenses employed by using organizations. This decision is based on academic use. Since this course does not focus on security concerns, implementing secure operations in our Repository would use time and effort better spent on the other activities our course focuses on. A commercial product would surely need some effective access control.

Conclusions

The Pluggable Repository concept can be implemented with reasonable cost and schedule resources. It is likely to improve developer and quality assurance productivity, and will make progress information available to managers and customers. An initial feasibility prototype¹¹ can be implemented quickly and will provide a very good vehicle for assessing impact on software development productivity.

¹¹ The results of Sample Project #4.

Appendix – Repository Classes

Class diagrams are seldom included in Operational Concept Documents, as they are concerned with implementation, not concept. However, this class structure is important for our concept because it shows the component interfaces used for each of the pluggable policies. Each of the concrete classes is allocated to its own package, and has the same responsibilities as its package.

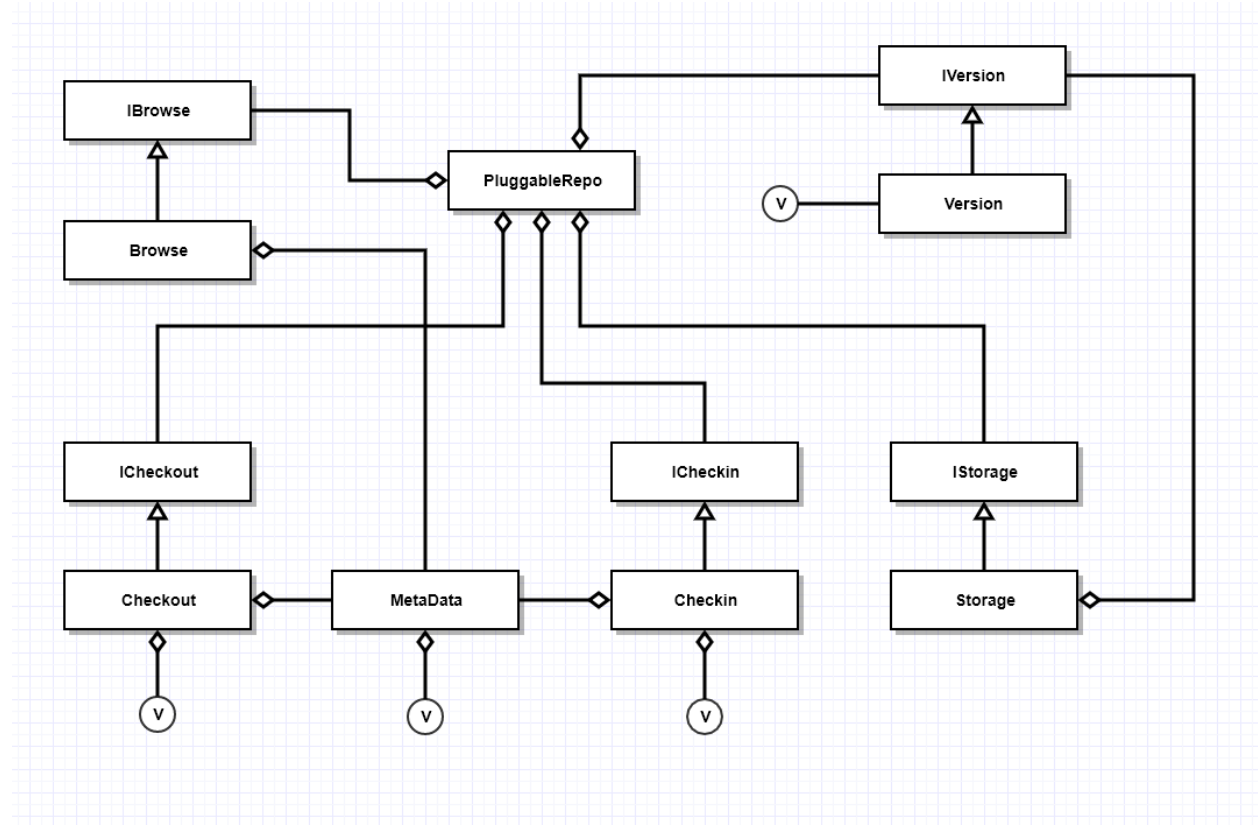


Figure 5 – Pluggable Repository Class Diagram – Second Project

Note that the interfaces are defined by the Pluggable Repository, not each individual component. The Repository defines the language it will use to communicate with each of the policy components, and the components are required to implement that language.