Mercurial

Software Patterns
Assignment 2
Group 4
Version 3

Divya Avalur
Assel Bekbatyrova
Eamon Nerbonne
Sunna Björg Sigurjónsdóttir
Edy Suharto

Contents

Revision history

Date	Version	Description
19/12/2011	1	**Assel** Stakeholder concerns, key drivers, patterns **Divya** Introduction of the system and overview of contents **Eamon** System context text, SR diagram **Edy** System context diagram **Sunna** Document creation, stakeholders and concerns, patterns
9/1/2012	2	**Sunna, Eamon** Conversion from google docs into HTML on Mercurial; minor updates.
16/01/2012	3	**Assel** Pattern descriptions **Divya** Architectural View: logical, development **Edy** Architectural View: process, physical **Sunna** Chapter 3 **Eamon** Pattern candidates, merge document, ToC

		generator
23/01/2012	4	**Assel** Client-Server pattern **Divya** Interceptor Pattern **Edy** Add architectural diagram, refine views and reference **Sunna** Chapter 5, Proxy pattern **Eamon** Pattern candidates, merge document, general editing
12/02/2012	5	**Assel** Proxy pattern, evaluation and recomendation sections **Divya** Views, Interceptor pattern diagram, recommendations for Reverse Proxy pattern **Edy** Refine architectural views and diagrams, relaxed layer pattern evaluation and recommendation **Sunna** Chapter 5, Client-Server & Peer-to-peer patterns, diagrams: CS, P2P and system overview, Glossary, final additions chapter 2,3,4,5 **Eamon** merge document, layout, bibliography, HTM validation, Sections 2+3+4; shared datastore, proxy, publish-subscribe, figure numbering

1 Introduction

The document presents the pattern-based recovery and evaluation of the architecture of an open-source system, Mercurial [1]. It is a distributed version control system. This document gives an overview of the patterns that represent the core of the Mercurial system. We used the IDAPO [19] process for identifying the architectural patterns in Mercurial. We also discuss the specific variants of the patterns which we found, and evaluate the patterns using Pattern-Based Architectural Review (PBAR) [20] method. The evaluation of the patterns is based on the system's quality attributes.

The structure of the document is as follows:

Section 2 describes the system context of the Mercurial version control system. Section 3 deals with identifying the stakeholders and their concerns as well as the key drivers which should drive the development of the system.

Section 4 describes several architectural views of the system and the architectural patterns identified for the Mercurial system using IDAPO. The specifics of each pattern's usage, the problem(s) it addresses, its consequences on the system, and other details are also presented. Finally, we describe the architectural decomposition and the responsibilities of each component.

Section 5 is focused on the architectural evaluation of the Mercurial system using the PBAR approach. It also shows how key drivers are addressed.

2 System context

Mercurial [2] is a distributed revision control system. It aims to help developers manage changes (revisions) made to projects. Managing multiple versions of even a single document by hand is an error-prone, time-consuming task. Mercurial allows a team of developers to manage large numbers of documents with many versions. Although Mercurial can maintain a version history for arbitrary files and directories, it is designed in particular to deal well with large numbers of complex text documents such as source code. There are several ways in which Mercurial helps address the problems arising from collaborating on such code.

2.1 Mercurial's functionality

Firstly, Mercurial maintains a log: an annotated history of changes. In addition to keeping track of the documents themselves, Mercurial allows developers to keep track of which lines were changed, the date the changes were made, the identity of the author, and, importantly, the motivation for these changes. This is achieved by splitting changes into cohesive chunks called changesets. When a developer commits a changeset, he adds it to Mercurial's repository of changesets, and includes a commit message describing what was changed and why. Changesets with commit messages are common in revision control systems; they allow developers to maintain systems that are too large or too old by reducing the need to grasp the entirety of the system at once.

Mercurial assists collaboration by allowing concurrent edits. When several people simultaneously change a set of files, their changes need to be merged into a final version. Revision control systems such as Mercurial assist the user by largely automating this process: if only one author changes a file, his version simply replaces the old one. When multiple authors change a file in a format Mercurial understands (such as plain text), changes to non-overlapping regions are automatically merged. Finally, where changes do conflict, they are reported, and the user can systematically compare the changes to resolve the conflict. Automatic merging and conflict detection helps allow multiple developers work on the same project by avoiding the friction of each developer keeping abreast of all changes.

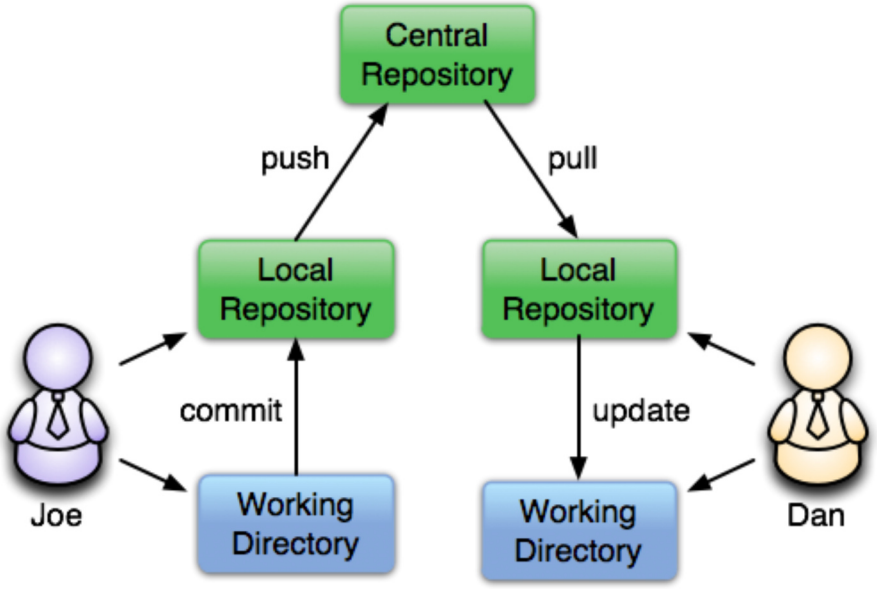

Figure 1: From [12]. A common mercurial workflow; Joe & Dan interact with their own repository which contains the entire project history. Changes are usually exchanged via a central repository.

Mercurial allows distributed development teams by allowing any developer to pull others' changes or push his own to an external repository (see Figure 1). Crucially, there is no technical need for a central repository nor for "one true" history, allowing for truly distributed development. In particular, each committed changeset is uniquely identifiable, allowing Mercurial to avoid applying changes it has already seen, even if those changes depend on each other in non-trivial ways. Mercurial's logged history is conceptually a directed acyclic graph. Thus when two versions are merged that (partially) share the same history, the shared part of the history is not duplicated.

These three features enable many useful scenarios. For instance, in a large system, development mistakes are made. Since it maintains a history of changes and the details of each changes, Mercurial can revert a particular change e.g. to correct a mistake. Unlike a strictly linear process, such a reverted change does not require the user to redo all subsequent changes; instead, the reverted change is itself a new changeset

that can be merged anywhere into the version history.

2.2 Project history and community

Matt Mackall is the creator of Mercurial and has been working on the system as the main developer since 2005 [44]. Mercurial's creation was a response to the Linux kernel's search for an alternative to BitKeeper [5]. Although the kernel ended up using Git, a similarly new SCM by Linus Torvalds, Mercurial was subsequently adopted by several other OSS projects. The origins in the linux kernel development community motivated two of Mercurial's key requirements: the ability to scale up to a large repositories with many revisions, and the need to support a truly distributed development style.

Mercurial has an active community[45], that can be reached via various mailing lists, depending on your needs. [46] For a fast response to your questions you can reach Mercurial developers on IRC, where they even provide a timetable when various developers are likely to be using the IRC. [47]

3 Stakeholders and concerns

3.1 Stakeholders and concerns

Stakeholders and their concerns have been identified by reading the documentation and analyzing the functionality of the system. Stakeholders are not specifically defined in Mercurial documentation, but the benefits for end-users are listed [10].

End user The end user (software developer) can be a large team of people as well as one person.

Mercurial project leader Matt Mackall is the sole full-time developer working on Mercurial and is also the project leader, serving as a gatekeeper, validating third-party contributions [11]. His stated aims in creating Mercurial were to create a simple, scalable, efficient, and distributed SCM [48].

Core Mercurial software developers Several other software developers regularly contribute to various Mercurial components.

Patch/extension developer Mercurial welcomes patches and extensions from any software developer.

Stakeholder	Concerns
End user	**Reliability:** The system should be stable and available even in the face of an unreliable network connection. **Durability:** The system should not lose data even in the face of a crash **Extensibility:** The system should be flexible and adaptable to users' needs and workflows **Scalability:** They system should be able to deal with large projects having many files and many changesets. **Usability:** The system should be easy to use and easy to learn **Performance:** The system should work quickly on projects of any size. **Portability:** The system should work interoperably on common platforms
Mercurial project leader	**Modifiability:** The system should be simple enough to be easily changed or extended. **Scalability:** The system should scale to large projects such as the Linux Kernel or Mozilla Firefox.

	Efficiency: The system should use minimal memory, disk I/O and bandwidth **Distributed:** The system should allow a distributed development style without requiring centralized control.
Mercurial software developer	**Modifiability:** The code should be well documented and easy to change and extend. **Efficiency:** For effective work of the system WAN bandwidth and disk seek rate should be optimized for the system.
Patch/ extension developers	**Extensibility:** The system should be flexible enough to accommodate diverse needs as easily as possible.

3.2 Key drivers

The key drivers are identified in Mercurial's documentation [13]. These key drivers are talked about extensively in several Mercurial sources. They not prioritized in the documents and therefore are listed in arbitrary order below.

Usability: The basic interface of Mercurial is easy to use, easy to learn and hard to break. Mercurial sports a consistent command set and potentially dangerous actions are available via extensions which need to be enabled.

Extensibility: The functionality of Mercurial can be increased with extensions, either by activating bundled extensions, downloading third party extensions, or by easily writing a new one.

Portability: Mercurial is written with platform independence in mind. Binary releases are available on all major platforms.

Performance: Mercurial is designed to be fast. One can generate diffs between revisions, or jump back in time within seconds.

Reliability: Mercurial is truly a distributed system, which gives each developer a local copy of the entire development history. This way it works independent of network access or a central server.

4 Architecture

According to its available document on website [7], the general overview of Mercurial architecture can be depicted below.

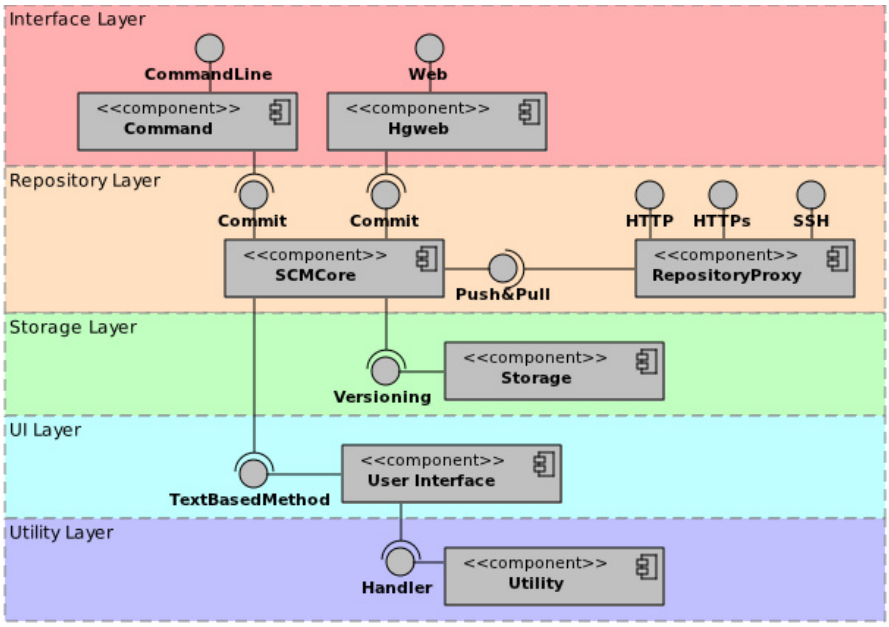

Figure 2: Mercurial Architectural Diagram based on [7]

Since Mercurial can be seen as peer-to-peer application, each peer has the same structure as the other. Each peer represents a node which contains five layers basically. Seen from bottom-up view, the layers for each peer are Utility, UI, Storage, Repository, and Interface. In general, no layer should know the details of any layer above it, and no layer should abuse the interface of layers below it [7]. However, the layers are not truly strict since a certain layer can access the bottom layer directly.

One interesting point is there are two layers that seem similar, Interface layer and UI (User Interface) layer. Based on investigation in source code, the interface layer deals with commands which the end-user could execute e.g. add, commit and update, while UI layer deals with the message which is delivered to end-user regarding to the command execution.

4.1 Architectural views

4.1.1 Logical view

The logical view for the Mercurial describes the functionality of the Mercurial system to the end-users. Mercurial is a modern distributed version control system (DVCS) which is written in Python. The architecture of Mercurial is decomposed logically into five layers. It is basically follows a Relaxed Layer pattern. The bottommost layer is the utility layer which implements handler. It includes generic functionality and platform abstraction. The next layer is the User Interface Layer. It implements a TextBasedMethod and provides generic methods for communicating with the user and managing configuration info. The storage layer consists of a storage component which is used to implement versioning of the system. It contains the basis for version storage, revlog. The repository layer consists of two components SCMcore and Repository Proxy. The SCMCore implements core primitives such as commands like pull, update, add, commit and push. The Hg update is used to update the file in the given repository. The Hg add is used to add any file. The Hg commit is used to commit any changes in the file. The Hg push is used to push the file into the repository. The remoting interface (push or pull) is used between SCMcore and RepositoryProxy. There are proxy objects for remote repositories (remoterepository, sshrepository, httprepository, httpsrepository, statichttprepository) inside RepositoryProxy. The interface layer consists of two components, Hgweb and Command. This layer is basically the layer in which users interact with the system. Hgweb has web interface and Command component has command-line interface [7].

Regarding data model, Mercurial uses a generic structure of historic data, called

revlog. There are three types of revlogs: the changelog, manifests, and filelogs. The changelog contains metadata for each revision, with a pointer into the manifest revlog (that is, a node id for one revision in the manifest revlog). In turn, the manifest is a file that has a list of filenames plus the node id for each file, pointing to a revision in that file's filelog. In the code, there are classes for changelog, manifest, and filelog that are subclasses of the generic revlog class, providing a clean layering of both concepts [5].

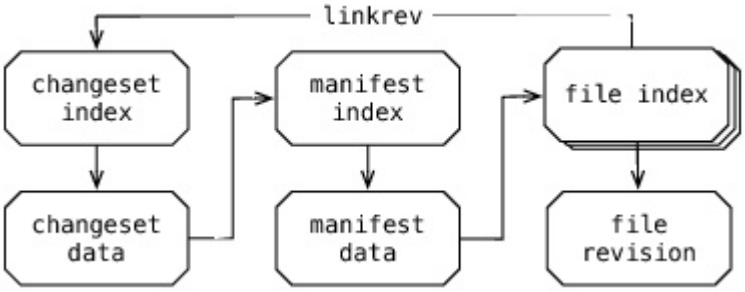

Figure 3: Mercurial Generic Data Model, Revlog [5]

Mercurial is a command-line application. This means a simple interface, the user calls the hg script with a command. This command (like log, diff or commit) may take a number of options and arguments. There are also some options that are valid for all commands. Next, there are three different things that can happen to the interface: hg will often output something the user asked for or show status messages, hg can ask for further input through command-line prompts, hg may launch an external program (such as an editor for the commit message or a program to help merging code conflicts). The correlations between modules is depicted below.

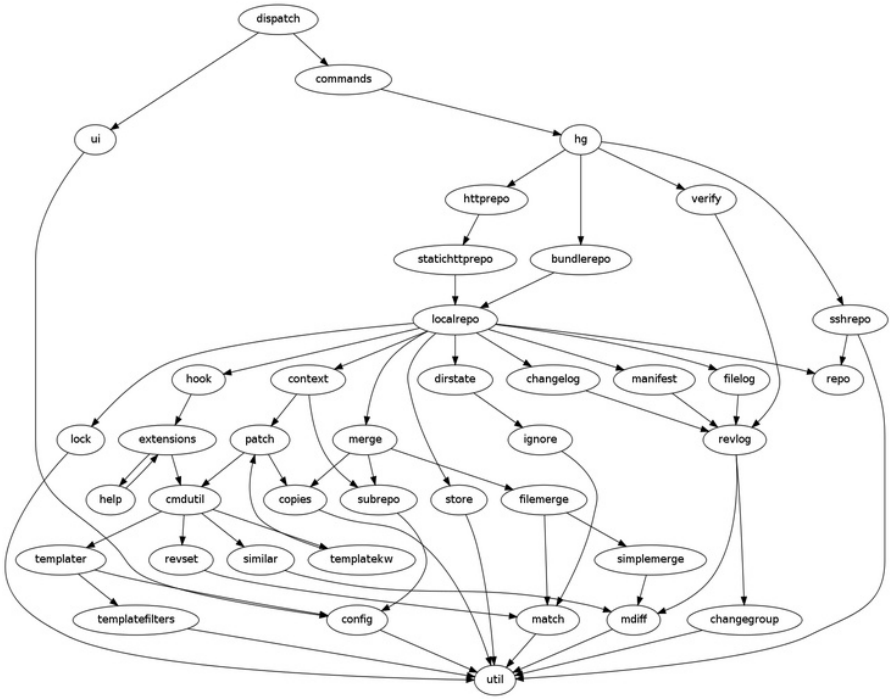

Figure 4: Mercurial Modules Correlation [5]

4.1.2 Process view

The process view of Mercurial depicts the interaction of processes or objects during runtime. When Mercurial tracks modifications to a file, it stores the history of that file in a metadata object called a 'filelog'. Mercurial uses a structure called a 'manifest' to collect together information about the files that it tracks. Each entry in the manifest contains information about the files present in a single changeset. The 'changelog' contains information about each changeset. Within a changelog, a manifest, or a filelog, each revision stores a pointer to its immediate parent (or to its two

parents, if it's a merge revision). For every changeset in a repository, there is exactly one revision stored in the changelog. Each revision of the changelog contains a pointer to a single revision of the manifest. A revision of the manifest stores a pointer to a single revision of each filelog tracked when that changeset was created. The underpinnings of changelogs, manifests, and filelogs are provided by a single structure called the revlog. Mercurial only ever appends data to the end of a revlog file. Mercurial treats every write as part of a transaction that can span a number of files. A transaction is atomic: either the entire transaction succeeds and its effects are all visible to readers in one go, or the whole thing is undone. This guarantee of atomicity means that if user is running two copies of Mercurial, where one is reading data and one is writing it, the reader will never see a partially written result that might confuse it. In the working directory, Mercurial stores a snapshot of the files from the repository as of a particular changeset. When user updates the working directory to contain a particular changeset, Mercurial looks up the appropriate revision of the manifest to find out which files it was tracking at the time that changeset was committed, and which revision of each file was then current. It then recreates a copy of each of those files, with the same contents it had when the changeset was committed [4].

According to the process view explained above, Mercurial process view is depicted in sequence diagram below.

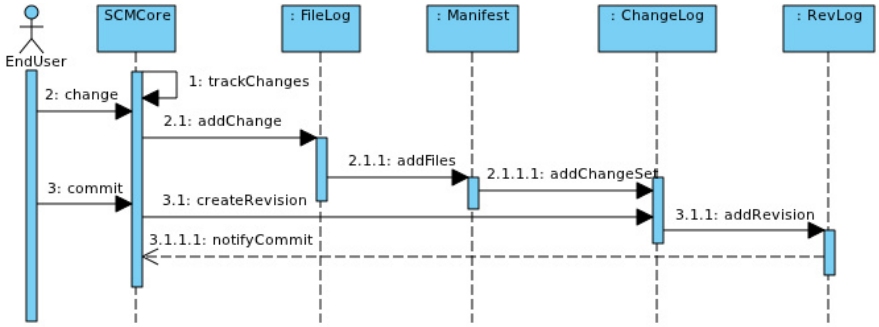

Figure 5: Mercurial Sequence Diagram for Modify and Commit

4.2 Candidate Architectural Patterns

The IDAPO process [19] was followed in the making of this report. In step 4 candidate patterns are identified and listed. In steps 5-7 the documentation and source code is studied in detail to assert that the candidate patterns are actually present. Below is the list of candidate patterns and our conclusions on their presence in Mercurial.

Remote Proxy For accessing remote repositories from a local client. *Accepted*

Layers The levels of abstraction. *Rejected, as some layers are bypassed*

Relaxed Layers The levels of abstraction are more suitable as relaxed layers. *Accepted*

Peer-to-peer Mercurial itself implements such a system between repositories - it's arguable that it's a property of the system not within the system; but it's important to the architecture regardless. *Accepted*

Shared Datastore aka Shared Repository; renamed to avoid confusion with the term repository to meaning a Mercurial SCM repository. The disk revlogs are shared between various other components and could be seen as a shared repository. *Accepted*

Indirection layer The revlog storage can be seen as an indirection layer over the filesystem: direct access is problematic due to performance, corruption and locking. *Rejected; shared repository is a better match*

Active repository Active repository: the Mercurial VCS itself can be seen as an active repository in that it provides hooks that can actively notify others of changes. Is this more like a feature than a pattern? *Rejected; implementation looks more like the Interceptor pattern.*

Interceptor Mercurial allows for configurable external code to be executed at predefined points to allow for repository hooks and extension functionality. *Accepted*

Client-Server A Mercurial repository communicates via a client-server protocol

when over the network e.g. when pushing or pulling. *Accepted*

4.3 Architectural patterns

4.3.1 Peer-to-peer

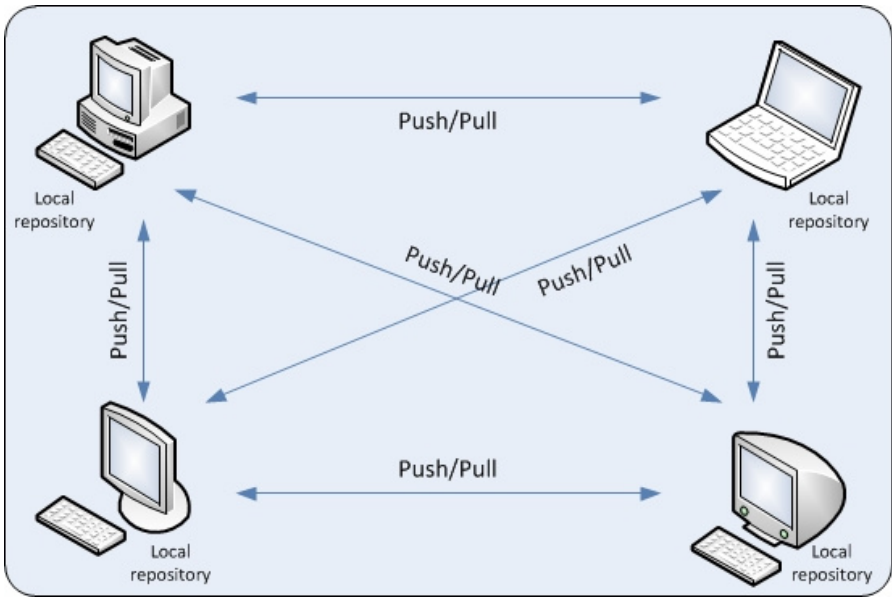

Figure 6: Mercurials Peer-to-peer diagram

Pattern Section	Comments
Name	Peer-to-peer
Problem Context	Mercurial is designed to allow for a truly distributed workflow, hence it is truly distributed.
Solution Variant	In Mercurial each peer can act as both client and server, depending on the setup chosen. In Mercurial there is no need for a central repository. Each client clones (makes a full copy) the whole repository he is working on from another client, and is able to pull from that client and push to other clients at any time. While working, you only make changes to your local repository until you push the changes out.
Rationale	The peer-to-peer pattern allows for excellent reliability, since Mercurial is truly distributive and each user stores the whole repository on their local machine and users are not limited by network problems. Performance and usability is improved since local commits can be done without pushing remotely to a server ([14], [15], [2]).
Consequences	This pattern eliminates the problem of SOP (Single Point of Failure). As each componenent acts as a peer, time of request and time of response is reduced. This inturn will increase the communication speed and also increases work productivity. Peer-to-peer also increases scalability, since it does not limit the amount of users sharing one project.

Related patterns	Client-Server

4.3.2 Shared Datastore

The Shared Datastore pattern is usually called *Shared Repository*: to avoid confusion with the term “repository” which is used differently in the context of Mercurial the pattern is renamed here. Figure 3 (section Section 4.1.1) shows the setup of the back-end mercurial datastore.

Pattern Section	**Comments**
Name	Shared datastore
Problem Context	Mercurial needs to store versioned data efficiently and robustly. Versioned data includes not just user-data (e.g. source code), but also metadata (e.g. commit messages), and the repository tree structure. All data stored in Mercurial must be stored efficiently, i.e. with low space and time overhead. Taking into account the performance characteristics of hard drives and the typical data storage patterns to optimize access while avoiding data corruption in the event of a crash or software bug is complex. In particular, SCM systems generally add new data but rarely (if ever) change existing data; stale data is read less frequently than the newest version; disk seeks are slow yet repositories may contain very many revisions with very many files so that the complexity of retrieving historical data should not depend on the size of the repository.
Solution Variant	Mercurial stores most of its data in *revlogs* [39]. Each Mercurial repository has many such revlogs; most persistant data (whether Mercurial-internal or user data) is stored and managed centrally in these datastructures. Each revlog consists of an append-only data file and an index file with fixed-size entries (to enable O(1) seeking [49]) storing where a particular version is to be found in the data file. There are three classes of revlogs. Firstly, user data is stored in a per-file revlog. Secondly, a directory listing with revision numbers of each file forming a particular version of the directory tree is stored in a manifest revlog. Finally, changeset metadata such as commit messages and the corresponding manifest revision number is stored in a changeset revlog.
Rationale	Standardizing the data storage method using revlogs primarily supports the system's performance by ensuring versioned information can be accessed quickly regardless of the amount of revisions stored. Additionally, the shared datastore simplifies extensibility partially by providing a uniform abstraction for many different types of data stored by mercurial. Also, the shared datastore accounts for platform differences such as file system case-sensitivity and illegal characters, so that a repository can be copied between platforms without corruption simply by copying the repository's directory, which improves portability.
Consequences	The revlog's simplicity improves durability: Adding data to a shared datastore can be performed simply by appending data to

Pattern Section	Comments
	the revlog and its index. In addition to using OS locking to ensure atomic write access, the new revlog contents are appended before the new revlog index contents are so that if a write fails, it does not corrupt the repository [5]: failure during the content append never touches previous revisions and the corrupted data can simply be ignored. Failure during the index append can either be recovered by a truncation (thus atomically failing the write without corruption), or by recomputation since the contents are already completely written. In a similar fashion, dependencies between different revlogs are ordered so that any failure is can be rolled back and never affects previously stored revisions. Furthermore, the typically append-only access pattern somewhat improves the usability goal of making it hard to make mistakes that result in data loss: access through Mercurial's API or user interface is almost always safe by virtue of the philosophy of never altering previously committed data. Finally, the simplicity of the revlog format clearly improves modifiability of mercurial itself (a stated goal of mercurial's maintainer).
Related patterns	None of the identified patterns

4.3.3 Client server

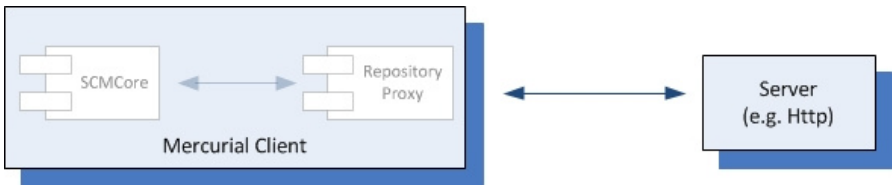

Figure 7: Mercurials Client server diagram

Pattern Section	Comments
Name	Client-Server
Problem Context	Two components who are independent from each other need to communicate with each other and even run on different computers (Mercurial is a distributed system). One of the peers initiates communication by sending a request to other peers. The server must cope with multiple requests at the same time from several peers. Clients wich are using the same server may have different local versions.
Solution Variant	The Client-Server pattern consists of two kinds of components: clients and servers. In Mercurial the server consists of http component, ssh component or local disk component, depending on how the user decides to set up the repository. The client communicates to the server components depending on the users choise.
Rationale	Due to the fact the Mercurial is truly distributed and uses Peer-to-peer pattern (see above), it is necessary to have a Client

	Server pattern. Portability is the key driver supported by this pattern since the pattern allows for high level communication protocols which can be implemented on most major plattforms.
Consequences	Usability is affected in a negative way since setting up repositories can be difficult, depending on the approach chosen [16].
Related patterns	Peer-to-peer, Remote Proxy

4.3.4 Remote Proxy

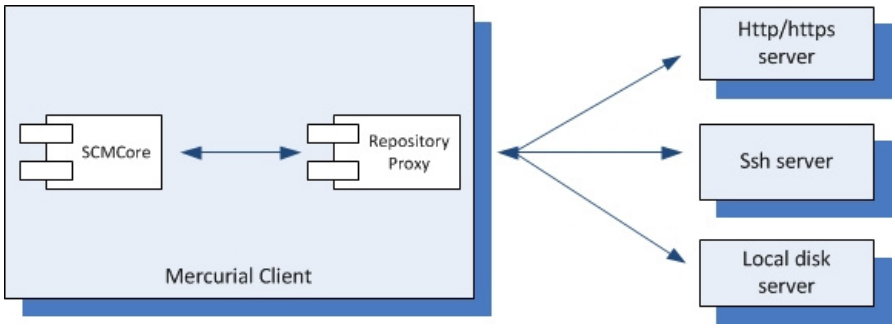

Figure 8: Mercurials Proxy diagram

Pattern Section	**Comments**
Name	Remote Proxy [7]
Problem Context	Client applications should be able to access remote and local repositories in a transparent manner. In a Mercurial users start by cloning a copy of a repository. They can synchronize changesets with it (or another repository) whenever they need to, and other developers can do so concurrently. However, there are many different ways of communicating with a remote repository depending on its type.
Solution Variant	To avoid the complexity, a Mercurial repository communicates via an internal representation called a proxy, which presents an abstract API covering the details of access to the remote repository. It is implemented in the client and contacts any server type such as http, https, ssh, a different directory on the local disk, or even an extension-provided implentation such as for Git or Subversion interoperability (see Figure 8). The proxy object is a surrogate or placeholder for the other repository and controls access to it [17]. The client mercurial module makes a request to the proxy. The Proxy forwards the request using the proper communication protocols (pipelined request/response protocol) and security measures. The remote repository accepts the request and attempts to fulfill it. It sends the response back to the proxy. The proxy interprets the response, and translates it into a canonical representation which is returned to the client.
Rationale	Encapsulating complexity within a proxy object simplifies the internal API, improving extensibility by simplifying development and/or configuration of extensions.

Consequences	One of the benefits of using a proxy object is that it hides the fact that an object resides elsewhere. The proxy object is responsible for all the networking complications, which could improve usability (by simplifying configuration), maintainability, and modifiability. The proxy hides the fact that a service is provided remotely, and makes the remote service as easy to use as a local service. By putting all location information and addressing functionality into a Remote Proxy variant, clients are less affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable.
Related patterns	Client-server, peer-to-peer

4.3.5 Interceptor

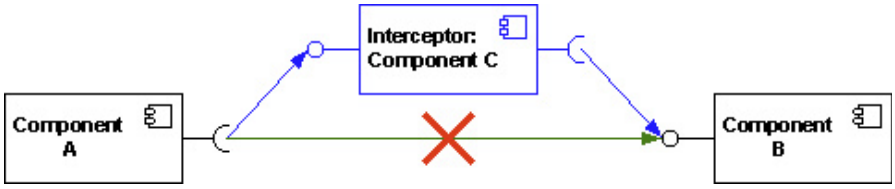

Figure 9: Mercurial Interceptor diagram

Pattern Section	**Comments**
Name	Interceptor
Problem Context	Mercurial has been designed to be part of a continuous integration system that builds and tests software automatically and regularly
Solution Variant	The pattern detected is Interceptor for Mercurial. Interceptor defines the signatures of the hook methods. Hook is a code which handles intercepted function calls,messages and events. They provide a way for version control systems like Mercurial to interact with the outside world.For example if we want to send an email automatically after commit occurs we have Commit hook. We can also invoke a hook if we want to send a mail containing all changesets of a push/pull/unbundle.In Mercurial, users/developers can invoke Python hooks by specifying a Python module and a function name to call from that module. The Mercurial framework facilitates the registration of interceptors through dispatchers.
Rationale	The advantages of detecting interceptor pattern is they are associated with a particular set of events. This mainly improves flexibility. Concrete interceptors specialize interceptor interfaces and implement the hook methods of Mercurial to handle events in an application-specific manner. The interceptor pattern helps to achieve usability and extensibility.
Consequences	This pattern includes strict seperation of framework and interceptors allows independent variation and evolution of service extensions in Mercurial. The complexity of the code increases thereby decreasing maintainability.

Related patterns	No patterns identified

4.3.6 Relaxed Layers

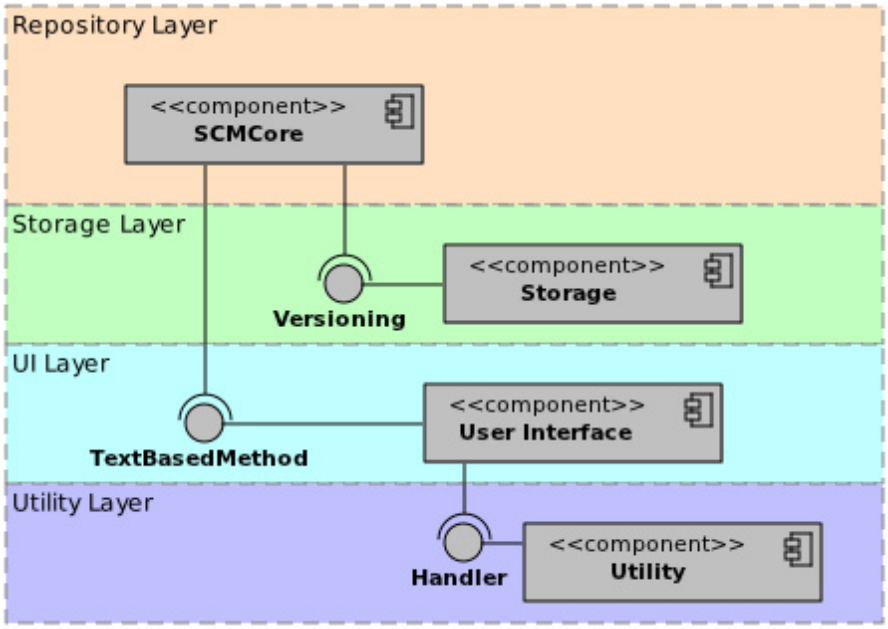

Figure 10: Mercurial Relaxed Layers diagram

Pattern Section	Comments
Name	Relaxed Layers [3]
Problem Context	Mercurial has to be designed with separation of concern, capability to extend, and platform independence
Solution Variant	The separation of concern and extensibility are resolved by dividing Mercurial system into several modules and grouping them into some layers based on their level of abstraction. Platform dependency can also be avoided by providing particular module.
Rationale	The advantage of choosing relaxed layer pattern is the separation of modules into layers but still maintaining the flexibility among modules of different layers, e.g. between top layer with bottom layer, to interact directly. The layers also enables developer or patch/extent developer to extend Mercurial features separately without interfering with other modules or layers. The separation of modules allows the system to be developed in different platforms.
Consequences	This pattern allows layers to interact without passing through any intermediate layer. This could be a problem when a component in a lower layer changes, the developer must examine not only its upper layer but also any layers above which interact directly to it. This could reduce maintainability.
Related patterns	None of the identified patterns

5 Evaluation & recommendations

PBAR [20] has been followed and in this section we evaluate the identified patterns based on the key-drivers. In recommendations issues are identified and given suggestions to solutions, as well as future recommendations.

5.1 Evaluation

We followed PBAR to evaluate the key drivers based on the patterns detected. If a key driver has a positive impact on the pattern, it gets a plus with green background, if a key driver has a negative impact on the pattern, it gets a minus with a red background, if a key driver has both a positive and a negative impact (two different situations) on the pattern, it gets a plus slash minus with a yellow background and finally if the impact is neutral (no impact) then it is left blank with a white background.The rationale for the evaluation is depicted below.

	Usability	Extensibility	Portability	Performance	Reliability
Peer-to-peer	+/-			+	+
Shared Datastore		+	+	+/-	
Client server	-		+		
Remote Proxy	+	+			
Interceptor	+	+			
Relaxed Layers		+	+		

Peer-to-peer

Usability and performance are improved since local changes do not have to be pushed onto a central server and therefore network problems do not affect your work. Usability also gets a minus because when users are not actively pushing and pulling changes it can cause a difficult merge wich can take some time to work out.

Reliablity is improved because every user has a local copy of the whole repository on his local machine.

Shared Datastore

The revlog-based shared datastore patterns improves performance by allowing O(1) seek access regardless of the number of revisions stored. Furthermore, the name-mangling approach to revlog storage improves performance in the face of file system defragmentation. Also, the mangling ensures revlog names do not depend on platform-sensitive characters, enhancing portability. Finally, by hiding complex changeset storage implementation details behind a simple, clean API, extensibility is improved by allowing extension developers to modify the system behavior without being confronted with the complex internals.

However, the shared datastore solution also has a performance downside since it is strongly coupled to the file structure: this means that file renames require a new revlog corresponding to the new name; this new revlog has no previous version to diff to, so it needs to store a complete copy of the data, causing unnecessary overhead. This is particularly problematic as the remote proxy object exports this limitation too; so after a rename a subsequent pull from your repository or push by your repository causes the file to be sent over the network in its entirety again.

Client-Server

Portability is the key driver which is addressed by this pattern. Allowing for several different communication protocols to repositories and between computers, as well as supporting those for all major OS's.

Usability gets a minus due to the fact that it can be difficult to set up and con-

figure you own repository [16] if you choose to do so.

Remote Proxy
The main reason to pick remote proxy is it increases extensibility, it creates for each client local a representation of a different repository and decreases the number of special cases required for accessing different repositories.

The remote proxy is responsible for all the network hassle, so it could influence in usability in a good way. The proxy hides the fact that a service is provided remotely, and makes the remote service as easy to use as a local service.

Interceptor
Usability is the key-driver addressed in this pattern.Concrete interceptors help to handle events in an application specific manner therby satisfying usability. Hence usability gets a plus. Extensibility is also a key-driver which allows independent variation an evolution of service extensions in Mercurial. Hence extensibility is given a plus.

Relaxed Layers
Since Mercurial was developed in layers and modules, its functionality can be extended by module-based development inside certain layer. It is flexible to add new modules as new requirement might appear. Developing new module can be done without interfering other modules. Thus, extensibility increases.

Mercurial has repository layer which contains proxy object. This layer enables developer to implement specific platform-based module especially in term of remote repository. This mechanism allows Mercurial to be more platform independent, so it increases portability.

5.2 Recommendations

5.2.1 Publish Subscribe

One problem of the Mercurial system is in concurrent clients/users access. More precisely, when multiple users commit and exchange their changesets concurrently, they will often encounter multiple heads (multiple new versions) whose changes need to be merged (which again might be committed concurrently, possibly causing more heads again). Though some level of merging is unavoidable, the current process imposes needless friction, hampering usability: one "cycle" of editing involves many steps. A user must manually pull and update (to check if new versions were made and apply their changes locally); then he can create and commit a new version. Then he must again pull (to check for new heads to merge with, involving a network wait). The user may need to merge the new head with his own and then commit that merge, and then must push the changes (again involving a network wait). Assuming there are new remote changes, this implies (at least) 6 or 7 explicit invocations in addition to those necessary for the user to query Mercurial about the state of the system and the changes themselves. Also, 3 of the invocations involve the user waiting for the network. In many cases, however, only one or two of these actions are actually meaningful to the user (namely the initial commit, and possibly the merge, if non-trivial).

We propose to use the *Publish-Subscribe* pattern to replace the Client-Server pattern for communication between peers. This pattern could maintain a connection between peers and perform much of the bookkeeping automatically. Furthermore, the (slow) networking steps could be initiated in the background when a new version is available, well before the user actively checks for such changes. This pattern would improve usability both by reducing the number of steps required in common workflows, and allowing simplified workflows which complete elide the behind-the-scenes pulling and pushing from novice users. It could also improve performance by starting expensive operations in the background before the user requests their results. Finally, when the network is unstable, this could improve reliability by executing network operations whenever the network is available without requiring user interaction. A downside would be that this would mean a significant change in the mercurial's design since it would required a constant background process to be active,

whereas the current design simply performs one action then terminates.

5.2.2 Relaxed Layers (modification)

The employment of relaxed layers pattern by Mercurial is proper enough. However, the naming of interface layer and UI layer seems ambiguous. It would more make sense if UI layer is renamed as User Message Manager, for instance, since it deals with messaging. Otherwise, the layers should be reorganized to be more idiomatically layered, i.e. with a top interface layer and deeper core layers. Although the relaxed layers pattern can decrease maintainability[3], this is acceptable since maintainability is not key-driver; also, as the interaction between the lower layers is minimal, the maintainability is unlikely to decrease in practice since decreased maintainability stems from greater number of dependencies in a relaxed layered pattern compared to a strictly layered pattern.

5.2.3 Integration Reverse Proxy (or related)

In the document, usability is listed as a key-driver. In the evaluation usability has many pluses and minuses and therefore we would like to recommend an improvement of this. All mercurial installations include a built-in server hgserve. However it is rarely used since mercurial itself does not handle authentication; using it means allowing anyone that can connect access. This is a usuability and portability problem since setting up a server is complex and highly platform-dependent.

We recommend Integration Reverse Proxy which provides a homogenous view of collection of servers, without leaking the physical distribution of individual machines to end users. This will also provide security to the data from the end users. This patterns will improve usability as per the user/developer perspective. We can implement this in our system by configuring a back-end for Hgweb server[21]. Configuring a mercurial repository as a server would thus be simpler, since authentication could be handled by the Integration Reverse Proxy.

On the downside, the Integration Reverse Proxy is itself a centralised service, conflicting with Mercurial's peer-to-peer nature. A distributed authentication provider would be better; we did not find such a pattern.

Glossary

changelog	The changelog is part of a Mercurial repository. It contains all changesets of a repository and is stored in revlog format. For each changeset of the repository, there is exactly one file revision in the revlog file. [22]
changeset	An atomic collection of changes to files in a repository. I contains all recorde local modification that lead to a new revision of the repository. [23]
cloning	To clone a repository is to make a copy of it at a point in time. The new repository is self-contained. [24]
commit	The creation of a new changeset in a repository, based on the state of the working directory. [25]
conflict	A conflict occurs when two independent changesets modify overlapping sections of a file in different ways. During a merge, Mercurial may require the assistance of the user through a merge program which can be used to resolve those conflicts. [26]
diff	The term diff can mean one of two things: • The term originally referred to the diff program, which

	generates a text file that describes the differences between two files or source trees. • These days, the term is typically used to describe one of its output files, as a synonym for patch file. [27]
dirstate	Mercurial tracks various information about the working directory (the dirstate): • what revision(s) are currently checked out • what files have been copied or renamed • what files are controlled by Mercurial [28]
DVCS	Distributed Version Control System
filelog	When Mercurial tracks modifications to a file, it stores the history of that file in a metadata object called a filelog. Each entry in the filelog contains enough information to reconstruct one revision of the file that is being tracked. A filelog contains two kinds of information: revision data, and an index to help Mercurial to find a revision efficiently. [29]
head	A head is a changeset with no child changesets. The tip is the most recently changed head. Other heads are recent pulls into a repository that have not yet been merged. [30]
hg	The hg command (hg and then e.g. help, pull, push) provides a command line interface to the Mercurial system.
hook	A code which handles intercepted function calls, messages and events which are implemented as either external programs or internal python calls. Section 4.3.5[31]
IDAPO	A Process for Identifying Architecture Patterns in OSS [19]
log	An annotated history of changes
manifest	The manifest is the file that describes the contents of the repository at a particular changeset ID. It primarily contains a list of file names and revisions of those files that are present. The manifest ID identifies the version of the manifest that goes with a particular changeset. The manifest ID is a nodeid. Multiple changesets may refer to the same manifest revision. [32]
merge	A merge combines two separate changesets in a repository into a merge changeset that describes how they combine. Merging is the process of joining points on two branches into one, usually at their current heads. [33]
O(1)	A kernel scheduling design that can schedule processes within a constant amount of time, regardless of how many processes are running on the operating system.
OS	Operating System
parent	A parent revision serves as a baseline for defining a set of changes. Such a set of changes can either be a changeset in the repository or the local modifications in the working directory. The parent revision(s) of the working directory will naturally be used as the parent revision(s) of the changeset created by committing the local modifications. [34]
PBAR	Pattern-Based Architectural Review [20]

pull	A pull propagates changesets from a "remote" repository (the source) into a local repository (the destination). The source (or "remote") repository may be located on the same or on a different computer. In the latter case, the changes are pulled over the network using the protocol specified by the provided URL of the remote repository. Pull transfers only changesets which are missing in the destination repository. [35]
push	A push propagates changes from a local repository to a remote one. [36]
repository	In Mercurial the repository contains the history of your project. Strictly speaking, the term repository in Mercurial refers to the directory named .hg (dot hg) in the repository root directory. The repository root directory is the parent directory of the .hg directory. Mercurial stores its internal data structures – the metadata – inside that .hg directory. [37]
revision	In Mercurial, the term revision can have two different meanings, depending on context. Most often, "revision" refers to a distinct changeset in a repository, for example by specifying a repository-localrevision number for the --rev option of the hg update or the hg merge command (conceptually referring to "a revision of the repository"). The term "revision" is also used as the short form for the longer term "file revision", which refers to a certain version of a file stored in the repository (conceptually referring to "a revision of a file"). This dual-use of the term "revision" naturally follows from the fact that Mercurial uses the same technical method for storing changeset information and for storing file versions in the repository, namely the revlog format. [38]
revlog	A revlog, for example .hg/data/somefile.d, is the most important data structure and represents all versions of a file in a repository. Each version is stored compressed in its entirety or stored as a compressed binary delta (difference) relative to the preceeding version in the revlog. Whether to store a full version is decided by how much data would beneeded to reconstruct the file. This system ensures that Mercurial does not need huge amounts of data to reconstruct any version of a file, no matter how many versions are stored. [39]
SCM	Abbreviations: "software configuration management" or "software configuration manager". The automated management of the components of a software project as it evolves. There are two kinds of SCM: CentralisedSCM and DistributedSCM. Mercurial is a distributed SCM. [40]
tip	The tip revision (usually just called the tip) is the most recent changeset in the repository. The tip is the most recently changed head. [41]
update	Update the repository's working directory (the "working copy") to the specified revision of the repository or to the tip revision of the current (named) branch if no revision is specified. [42]
user-data	In Mercurial it is usually source code
working directory	In Mercurial the working directory contains a snapshot of your project at a particular point in history. It is the top-level directory in a repository, in which the plain versions of files are available to read, edit and build. Files in the working directory are usually from the tip, but may be from older revisions, or modified and not yet committed. It is sometimes called the “working copy”. [43]

References

[1] Mackall et al. Mercurial SCM 2011.

[2] Mercurial book, motivation , HgInit—a Mercurial tutorial, The Mercurial User's Guide, The Mercurial Wiki

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad.Pattern-Oriented Software Architecture Volume 1: A System of Patterns. 1996.

[4] Bryan O'Sullivan. Mercurial: The Definitive Guide. 2009. http://hgbook.redbean.com/read/behind-the-scenes.html#x8-640004

[5] Amy Brown and Greg Wilson. The Architecture of Open Source Applications. 2011. http://www.aosabook.org/

[6] Wikipedia

[7] Mercurial Project Architecture. http://mercurial.selenic.com/wiki/WhatGoesWhere

[8] Interceptor Pattern. http://en.wikipedia.org/wiki/Interceptor_pattern

[9] Mercurial: The Definitive Guide by Bryan O'Sullivan http://http://hgbook.redbean.com/read/collaborating-with-other-people.html

[10] About Mercurial: advantages of using Mercurial, Mercurial homepage

[11] Mercurial Wiki: Mercurial Development Process.

[12] Introduction to Bite-Sized Mercurial posted by Pete Vidler 2010-03-21

[13] About Mercurial: advantages of using Mercurial, Mercurial homepage

[14] Bite sized Mercurial introduction posted by Pete Vidler 2010-03-21

[15] Introduction to distributed version control an article in betterexplained.com by Kalid Azad

[16] Mercurial: Publishing Repositories In *Mercurial wiki*, Last update 2011-12-22.

[17] Remote proxy

[18] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2

[19] Stol, K.J. and Avgeriou, P. and Ali Babar, M. Design and evaluation of a process for identifying architecture patterns in open source software. In *Software Architecture*, pages 147–163. Springer, 2011

[20] Harrison, N. and Avgeriou, P. Pattern-based architecture reviews. In *IEEE Software*. IEEE, 2010

[21] Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, 2005

[22] Mercurial changelog In *Mercurial wiki*, Last update 2011-02-25.

[23] Mercurial changeset In *Mercurial wiki*, Last update 2011-11-04.

[24] Mercurial cloning In *Mercurial wiki*, Last update 2011-02-25.

[25] Mercurial commit In *Mercurial wiki*, Last update 2011-02-25.

[26] Mercurial conflict In *Mercurial wiki*, Last update 2009-05-19.

[27] Mercurial diff In *Mercurial wiki*, Last update 2011-02-25.

[28] Mercurial dirstate In *Mercurial wiki*, Last update 2010-10-26.

[29] Mercurial filelog In Mercurial: The Definitive Guide by Bryan O'Sullivan

[30] Mercurial head In *Mercurial wiki*, Last update 2011-02-25.

[31] Mercurial hook In *Mercurial wiki*, Last update 2011-07-29.

[32] Mercurial manifest In *Mercurial wiki*, Last update 2011-02-25.

[33] Mercurial merge In *Mercurial wiki*, Last update 2011-02-25.

[34] Mercurial parent In *Mercurial wiki*, Last update 2011-02-25.

[35] Mercurial pull In *Mercurial wiki*, Last update 2011-02-25.

[36] Mercurial push In *Mercurial wiki*, Last update 2011-02-25.

[37] Mercurial repository In *Mercurial wiki*, Last update 2011-02-25.

[38] Mercurial revision In *Mercurial wiki*, Last update 2011-02-25.
[39] Buehlmann, A. et al. Revlog. In *Mercurial wiki*, Last update 2011-02-25.
[40] Mercurial SCM In *Mercurial wiki*, Last update 2011-05-19.
[41] Mercurial tip In *Mercurial wiki*, Last update 2011-02-25.
[42] Mercurial update In *Mercurial wiki*, Last update 2011-02-25.
[43] Mercurial working directory In *Mercurial wiki*, Last update 2009-10-30.
[44] Matt Mackall In *Mercurial wiki*, Last update 2011-10-18.
[45] Mercurial community In *Mercurial wiki*, Last update 2011-12-27.
[46] Mercurial mailinglists In *Mercurial wiki*, Last update 2012-01-30.
[47] Mercurial IRC In *Mercurial wiki*, Last update 2011-11-07.
[48] Mackall, M. Mercurial v0.1 - a minimal scalable distributed SCM. On the *Linux Kernel Mailing List*, April 19, 2005
[49] Mackall, M. Towards a Better SCM: Revlog and Mercurial. 2006