Mercurial

Software Patterns
Assignment 2
Group 4
Version 1

Divya Avalur
Assel Bekbatyrova
Eamon Nerbonne
Sunna Björg Sigurjónsdóttir
Edy Suharto

4. Architecture → logical view → process / C&C

5. Pattern Documentation
5.1 ↳ Part 1 / sub-system
  ↳ pattern 1
  ↳ pattern 2
5.2 ↳ Part 2 / sub-system
  ↳ pattern 3
  ↳ pattern 4
↳ - - -

6. Quality attribute evaluation
6.1 KD / QA   quality attribute matrix
6.2 KD / QA

7. Recommendations
7.1 Negative impact
7.2 Conflicting impacts
7.3 QA not addressed
7.4 Subsystems with no patterns

8. Conclusions

- SH, concerns, KD
- any kind of pattern
- how were patterns found?
- what variants of patterns
- interaction of patterns

- consequences of patterns on QA
- ways to improve (new patterns, new variants; re-engineered design; link to actual design, future reqs, QA, FR)

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 Edy: System context diagram Sunna: Document creation, stakeholders and concerns, patterns

PBAR:

- identify important Quality reqs from doc
- learn about overall SA
- identify patterns
- determine impact of patterns on QA

- review major reqs
- review AR reqs
- identify patterns
- identify effect of patterns on QA
- identify issues

IDAPO: $\rightarrow$ part of pattern doc

- identify type/domain
- identify techn.
- study techn.
- identify cand. patterns
- read pattern lit.
- study documentation
- study source code
- study C&C
- identify patterns/variants
- Validate patterns
- get feedback from community
- register pattern usage

1 Introduction

for ?

The document presents the pattern-based recovery and evaluation of the architecture of an open-source system, Mercurial. It is a distributed version control system. This document gives an overview of the patterns that represent the core of the Mercurial system. We use the IDAPO for identifying the architectural patterns in Open source software. We also discuss about the specific variants of the patterns which we chose. Furthermore we need to evaluate the found patterns using Pattern-Based Architectural Review method. the evaluation of the consequences of patterns is based on the system's quality attributes.

The structure of the document is as follows:

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

Chapter 4 describes the various architectural views for the system and also the architectural patterns identified for the Mercurial system using IDAPO. It also discusses how the various patterns are related with each other and the impact of each pattern on the system architecture and also discusses about the variants used . It also includes the decomposition of the patterns, responsibilities of each component in the patterns and also about the interfaces.

Chapter 5 is mainly focused on the architectural evaluation of the Mercurial system using PBAR approach. It also shows how each quality attribute identified are addressed.

2 System context (5%)

Mercurial 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. such as ?

2.1 Mercurial's functionality

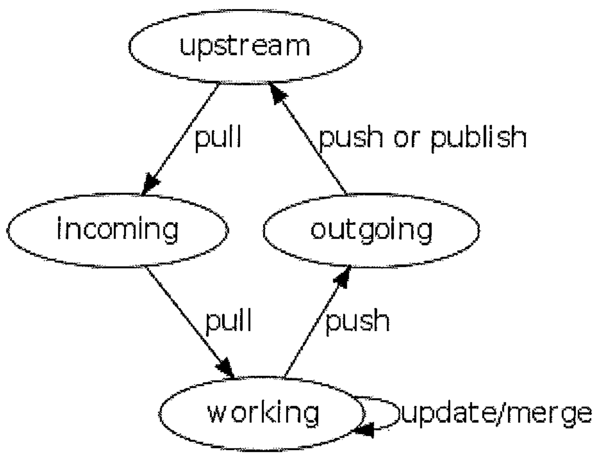
TODO: *relevant to explain source control in more detail than this?*

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 call 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 understand (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.

Mercurial allows distributed development teams by allowing any developer to pull others' changes or push his own to an external repository. 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.



?

?

Business model: System context, Architectural relevant business parameters.
Show system context and OSS community.

3 Stakeholders and concerns (10%)

3.1 Stakeholders and concerns from where?

End user - The end user (software developer) can be a large team of people as well as one person.
Mercurial management -
Mercurial software developer - There is currently one full-time developer working on Mercurial
Patch/extension developer - Mercurial welcomes patches and extensions from any software developer.

Stakeholder	Concerns	Concern priority	Stakeholder priority
End user	**Reliability:** The system should be stable and reliable to use for projects **Adaptability:** The system should adapt to a users needs in terms of extensions needed **Dependability:** Revisions should be accurate and easily ~~exchanged~~ **-ablity:** Data should not be lost in case of a breakdown in the system **Scalability:** They system should be able to handle large amounts of new data being submitted to the system at a particular time **Usability:** The system should be easy to use **Learnability:** The system should be easy to learn, not take more than a couple of days of using to get well accustom to ~~it~~ **Supportablity:** The system should come with user support **-ability:** The system should be perfectly suitable for large ~~projects~~, for that it has to work really fast.		
Mercurial management	**Changeability:** The application server should be easy changeable to adapt to the specific needs for the application which is run on top of it. **Adaptability:** The system should easily integrate in various environments		
Mercurial software developer	**Usability:** The code should be well documented **Efficiency:** For effective work of the system WAN bandwidth and disk seek rate should be optimized for the system.		
Patch/	**Usability:** The code should be well documented		

extension developers			

3.2 Key drivers

from where?

Usability: The whole system should be as simple as possible to get start to work on it

Scalability/Extensibility: The system repository should be extensible to store entire data

Changeability: The application server should be easy changeable to adapt to the specific needs for the application which is run on top of it.

4 Architecture (30%)

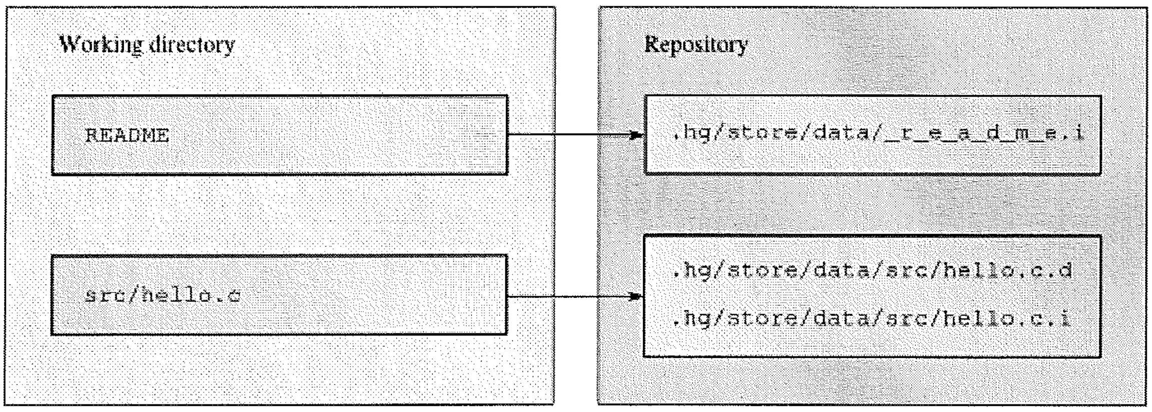
Show the individual patterns, their variants and how they integrate in the entire architecture; show at most 2 views.

4.1 Architectural views

4.2 Architectural patterns

4.2.1 Peer-to-peer

4.2.2 Shared repository


Working directory
README
src/hello.c
Repository
.hg/store/data/_r_e_a_d_m_e.i
.hg/store/data/src/hello.c.d
.hg/store/data/src/hello.c.i

4.2.3 Pipes and filters

4.2.4 Client server

4.2.5 Broker

4.3 Decomposition, component responsibilities & interfaces

5 Evaluation & recommendations(35%)

Follow PBAR to evaluate the quality attributes based on the patterns.
Emphasize on recommendations on how to improve the system.