PRINCIPLES OF SOFTWARE ENGINEERING

SUGGESTING QUERY COMPLETIONS FOR XAPIAN

FINAL REPORT

Pallavi Gudipati Tanmay Dhote Siddhant Mutha
Parth Joshi Mohammed Shamil

2014, May

ABSTRACT

Xapian is a Open Source Search Engine Library which currently has basic search utility. In this project we have incorporated two new functionalities in the project - Auto-complete and Query Expansion. Auto-Complete saves the time of the user by giving possible solutions while he is typing whereas Query Expansion improves user satisfaction by giving him more relevant searches to his query. Auto-complete has been achieved by implementation of a prefix tree whereas for Query Expansion, we have used WordNet for thesaurus support. A user interface has been created in Qt for user convenience.

CONTENTS

1 OVERVIEW OF XAPIAN

The software that has been chosen for our project is Xapian. Xapian is an Open Source Search Engine Library, released under the GPL. It's written in C++, with bindings to allow use from Perl, Python, PHP, Java, Tcl, C#, Ruby, Lua, Erlang and Node.js.

Xapian is a highly adaptable toolkit which allows developers to easily add advanced indexing and search facilities to their own applications. It supports the Probabilistic Information Retrieval model and also supports a rich set of boolean query operators.

Some of the noteworthy features of Xapian are:

- Free Software/Open Source - licensed under the GPL.
- Supports Unicode (including codepoints beyond the BMP), and stores indexed data in UTF-8.
- Highly portable - runs on Linux, Mac OS X, FreeBSD, NetBSD, OpenBSD, Solaris, HP-UX, Tru64, IRIX, and probably other Unix platforms; as well as Microsoft Windows and OS/2.
- Written in C++, with bindings allowing use from many other languages.
- Ranked probabilistic search - important words get more weight than unimportant words, so the most relevant documents are more likely to come near the top of the results list.
- Relevance feedback - given one or more documents, Xapian can suggest the most relevant index terms to expand a query, suggest related documents, categorise documents, etc.
- Phrase and proximity searching - users can search for words occurring in an exact phrase or within a specified number of words, either in a specified order, or in any order.
- Full range of structured boolean search operators ("stock NOT market", etc). The results of the boolean search are ranked by the probabilistic weights. Boolean filters can also be applied to restrict a probabilistic search.
- Supports stemming of search terms (e.g. a search for "football" would match documents which mention "footballs" or "footballer"). This helps to find relevant documents which might otherwise be missed. Stemmers are currently included for Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish, and Turkish.
- Wildcard search is supported (e.g. "xap*").
- Synonyms are supported, both explicitly (e.g. " cash") and as an automatic form of query expansion.

- Xapian can suggest spelling corrections for user supplied queries. This is based on words which occur in the data being indexed, so works even for words which wouldn't be found in a dictionary (e.g. "xapian" would be suggested as a correction for "xapain").
- Faceted search is supported.
- Supports database files > 2GB - essential for scaling to large document collections.
- Platform independent data formats - you can build a database on one machine and search it on another.
- Allows simultaneous update and searching. New documents become searchable right away.

1.1 CODE DETAILS

1.1.1 ROOT

Top level directory.

1.1.2 api

API classes and their PIMPL internals.

1.1.3 backends

This directory contains a subdirectory for each of the available database backends. Each backend corresponds to a different underlying file structure. For example, the inmemory backend holds its databases entirely in RAM, and the flint backend is a fully featured disk based backend.

The directory also contains the implementation of Xapian::Database::Internal (the base class for each backend database class) and the factory functions which are the public interface for instantiating the database backend classes.

backends/brass

Brass is the development backend for the Xapian 1.2.x series. It's highly efficient, and also (aims to) use significantly less disk space than previous Xapian backends. All Xapian features are supported.

backends/chert

Chert is the default backend for the Xapian 1.2.x release series. It uses a custom written Btree management system to store posting lists and termlists. This is a highly efficient backend, using compression to store the postlists, and supporting the full range of indexing functionality (positional information, transactions, etc).

backends/flint

Flint is the default Xapian backend as of Xapian 1.0. It uses a custom written Btree management system to store posting lists and termlists. This is

a highly efficient backend, using compression to store the postlists, and supporting the full range of indexing functionality (positional information, transactions, etc).

backends/inmemory

This backend stores a database entirely in memory. When the database is closed these indexed contents are lost.

This is useful for searching through relatively small amounts of data (such as a single large file) which hasn't previously been indexed.

backends/multi

The MultiDatabase backend, which enables searches to be performed across several databases. Opening this database involves opening each of the sub-databases and merging them together.

Searches are performed across the sub-databases via MultiPostList and MultiTermList objects, which represent merged sets of postlist and termlist objects.

backends/remote

The remote backend, which enables searches to be performed across databases on remote machines. Opening this database involves opening a communications channel with a remote database.

RemoteDatabase objects are used with RemoteSubMatch objects.

1.1.4 bin

Programs relating to the Xapian library.

1.1.5 common

Header files which are used in various places within the Xapian library code. It does not contain header files which are externally visible: these are kept in the "include" directory.

1.1.6 docs

Documentation, and scripts to automatically generate further documentation from the source code. If you have the appropriate packages installed (currently, this means Perl), these scripts will be run by make.

1.1.7 examples

This directory contains example programs which use the Xapian library. These programs are intended to be a good starting point for those trying to get acquainted with the Xapian API. Some of them are really just toy programs, but others are actually useful utilities in their own right (for example: delve, quest, and copydatabase).

1.1.8 expand

This directory houses the query expansion code.

1.1.9 include

This directory contains the externally visible header files. Internal header files are kept in the "common" directory.

1.1.10 languages

Utilities for performing processing of text in various different languages. Current these comprise stemming algorithms. In future language detection, character set normalisation, and other language related utilities will be added.

1.1.11 matcher

The code for performing the best match algorithm lives here. This is the heart of the Xapian system, and is the code which calculates relevance rankings for the documents in the collection for a given query.

1.1.12 net

The code implementing the network protocols lives here.

1.1.13 queryparser

Implementations of the Xapian::QueryParser and Xapian::TermGenerator classes.

1.1.14 tests

This directory contains various test programs which exercise most parts of the Xapian library.

tests/harness

This contains the test suite harness, which is linked with by most of the C++ test programs to perform sets of tests.

tests/perftest

This directory contains various the performance test suite.

tests/soaktest

This directory contains a testsuite for "soak" testing Xapian - it generates and runs random sequences of operations, checking the validity of the output whenever possible.

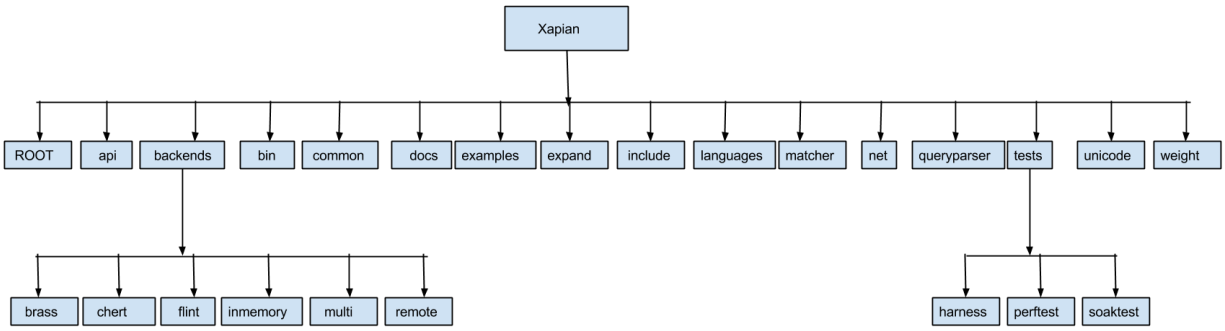
1.1.15 unicode

Unicode and UTF-8 handling classes and functions.

1.1.16 weight

Implementations of weighting schemes for Xapian.

Figure 1: Xapian Directory Structure



2 QUERY COMPLETION AND EXPANSION

2.1 QUERY AUTO-COMPLETION

Query auto-completion (QAC) is one of the most prominent features of modern search engines. Autocomplete involves the program predicting a word or phrase that the user wants to type in without the user actually typing it in completely. This feature is effective when it is easy to predict the word being typed based on those already typed, such as when there are a limited number of possible or commonly used words. Typically, the list of query candidates is generated according to the prefix entered by the user in the search box and is updated on each new key stroke.

Query prefixes tend to be short and ambiguous, and existing models mostly rely on the past popularity of matching candidates for ranking. The efficiency of word completion is based on the average length of the words typed.

2.2 QUERY EXPANSION

Query expansion (QE) is the process of reformulating a seed query to improve retrieval performance in information retrieval operations. In the context of web search engines, query expansion involves evaluating a user's input (what words were typed into the search query area, and sometimes other types of data) and expanding the search query to match additional documents.

Query expansion involves techniques such as:

- Finding synonyms of words, and searching for the synonyms as well.
- Finding all the various morphological forms of words by stemming each word in the search query.
- Fixing spelling errors and automatically searching for the corrected form or suggesting it in the results.
- Re-weighting the terms in the original query.

In query expansion, users give additional input on query words or phrases, possibly suggesting additional query terms. Search engines may suggest related queries in response to a query; the users then opt to use one of these alternative query suggestions. The central question in this form of query expansion is how to generate alternative or expanded queries for the user. The most common form of query expansion is global analysis, using some form of thesaurus. For each term t in a query, the query can be automatically expanded with synonyms and related words of t from the thesaurus. Use of a thesaurus can be combined with ideas of term weighting: for instance, one might weight added terms less than original query terms.

2.2.1 Precision and recall tradeoffs

Search engines invoke query expansion to increase the quality of user search results. It is assumed that users do not always formulate search queries using the best terms. Best in this case may be because the database does not contain the user entered terms.

By stemming a user-entered term, more documents are matched, as the alternate word forms for a user entered term are matched as well, increasing the total recall. This comes at the expense of reducing the precision. By expanding a search query to search for the synonyms of a user entered term, the recall is also increased at the expense of precision. This is due to the nature of the equation of how precision is calculated, in that a larger recall implicitly causes a decrease in precision, given that factors of recall are part of the denominator. It is also inferred that a larger recall negatively impacts overall search result quality, given that many users do not want more results to comb through, regardless of the precision.

The goal of query expansion in this regard is by increasing recall, precision can potentially increase (rather than decrease as mathematically equated), by including in the result-set pages which are more relevant (of higher quality), or at least equally relevant. Pages which would not be included in the result set, which have the potential to be more relevant to the user's desired query, are included, and without query expansion would not have, regardless of relevance. At the same time, many of the current commercial search engines use word frequency (Tf-idf) to assist in ranking. By ranking the occurrences of both the user entered words and synonyms and alternate morphological forms, documents with a higher density (high frequency and close proximity) tend to migrate higher up in the search results, leading to a higher quality of the search results near the top of the results, despite the larger recall.

This tradeoff is one of the defining problems in query expansion, regarding whether it is worthwhile to perform given the questionable effects on precision and recall. One of the problems is that the dictionaries and thesauri, and the stemming algorithm, are driven by human bias and while this is implicitly handled by the query expansion algorithm, this explicitly affects the results in a non-automated manner.

2.3 MOTIVATION

Xapian currently does not have any auto-completion or query expansion support.

The main motivation behind auto-complete is to decrease the average time taken by a user to obtain a relevant search result. Many search engines such as Google, Bing and Yahoo! have already integrated this into their websites and thus show search suggestions when users enter search phrases on their interfaces. These suggestions are meant to assist the user in finding what he/she wants quickly and also suggesting common searches that may result in finding information that is more relevant.

It also serves the purpose of helping the user if he/she is not sure of what to search for, but has a vague idea of what it is that she wants. Users who

find it difficult to type or users on mobile devices with constrained input methods can get to their results much faster because of auto-complete.

Query Expansion is another important feature that may be needed when the user does not know what he is exactly searching for. Query expansion gives us results that may be relevant to the user. User happiness/satisfaction (ie how well the algorithm is working) can only be measured by relevance to the information needs of the user and not by relevance to queries. This is the main motivation behind query expansion.

3 DESIGN ASPECTS AND MODIFICATIONS

3.1 LOGS AND TRIE IMPLEMENTATION

For Query Auto-Completion a special data structure called Trie is used.

A trie or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

The implementation of Query Auto-completion is done as follows:

- A log of terms that have been previously queried is created.
- Queries generated by all users are recorded on a particular database that is created by modifying Xapian's Database class.
- Each database is automatically allocated a unique id which is used to store the logs in an appropriate file.
- Popular searches can be recorded and suggested back.
- Trie tree is implemented for Query Auto-Completion

Trie Trees are used to complete the half-formed query. All the previously queried terms are stored in a file. For every query, a trie tree is created whose leaf nodes are the stored terms in the database. For every incomplete query, the subtrees with the current string as the root are traversed and the leaf nodes obtained correspondingly are then given out as autocomplete suggestions. The ordering of the suggestions is done according to the frequency that the terms have previously occurred. The terms which occur more often are given a higher ranking than a term which has a lesser frequency of occurrence.

3.1.1 Technical Details

Generating Logs

Xapian does not have any existing support to record query logs. Thus, we had to modify the existing `Xapian :: Database` class to accommodate log generation. We exploited the fact that Xapian assigns a unique id (UUID) to every database. The UUID will persist for the lifetime of the database. Replicas (eg, made with the replication protocol, or by copying all the database files) will have the same UUID. However, copies (made with copydatabase, or xapian-compact) will have different UUIDs.If the backend does not support UUIDs or this database has no sub-databases, the UUID will be empty.

If this database has multiple sub-databases, the UUID string will contain the UUIDs of all the sub-databases.

We can obtain this UUID using the Xapian :: Database :: get_uuid() method that returns a std :: string. We use this string as the name of the file used to store the logs. We also created a folder, xapian − core/logs/ to store the log files.

To change the Xapian :: Database class, we had to modify xapian − core/api/omdatabase.cc and xapian − core/include/xapian/database.h. A new method, Xapian :: Database :: log(const std :: string &) was created to log a query. It is a public function as the administrator using the API must be able to access it. This method creates a new file for the first query of the database and writes the query in it, or appends it for old databases.

One important point is that the administrator has the privilege of not logging the queries. To log a query, all he/she needs to do is call the above mentioned method, eg. db.log(query).

3.1.2 Trie Implementation

We added a new implementation of the data structure Trie to Xapian. To do this, we had to add two new files/modules to Xapian, xapian − core/include/xapian/trie.h and xapian − core/api/trie.cc. The header file defines a structure struct trie_node and a class class Trie.

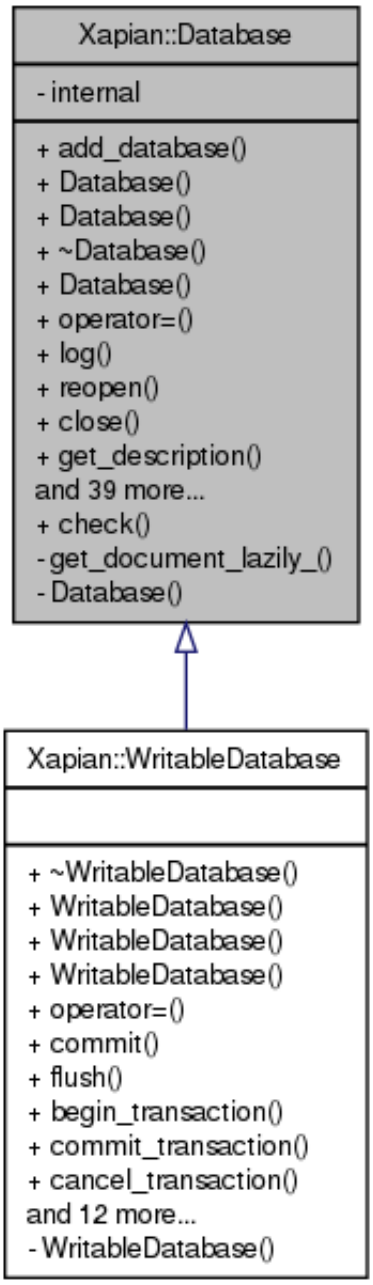
struct trie_node is the structure of each node in Trie. It contains a char value which stores the value of the node, bool is_child which indicates whether the node marks the ending of a legitimate word, int frequency which tells the number of times the word occurs and std :: vector < trie_node * > children which stores the pointers to all its children nodes.

This whole setup is a part of Factorypattern. Trie inherits PrefixMatcher class. This will be explained further in section 6.
class Trie represent the whole Trie. It contains one data member struct trie_node root which is the root node of the tree. It contains the following methods:

- add_term(std :: string term) - This method add te given term to the tree.
- search_term(std :: string term) - Return the pointer to the node that marks the ending of the given term. Returns NULL if the tree does not comtain the given term.
- get_matches(std :: string term) - Returns a vector of std :: strings that are legitimate terms in the tree with the given term as prefix.
- get_sorted_matches(std :: string term) - Same as above but the strings are sorted in descending order based on the frequency.
- build_matcher(Database db) - Build a whole tree consisting of all the previous queries of the given database.

The administrator has to build a tree using build_matcher() and use get_matches() after every keystroke to implement auto-complete. A working example was implemented and will be described in further sections.

Figure 2: Inheritance of class Database

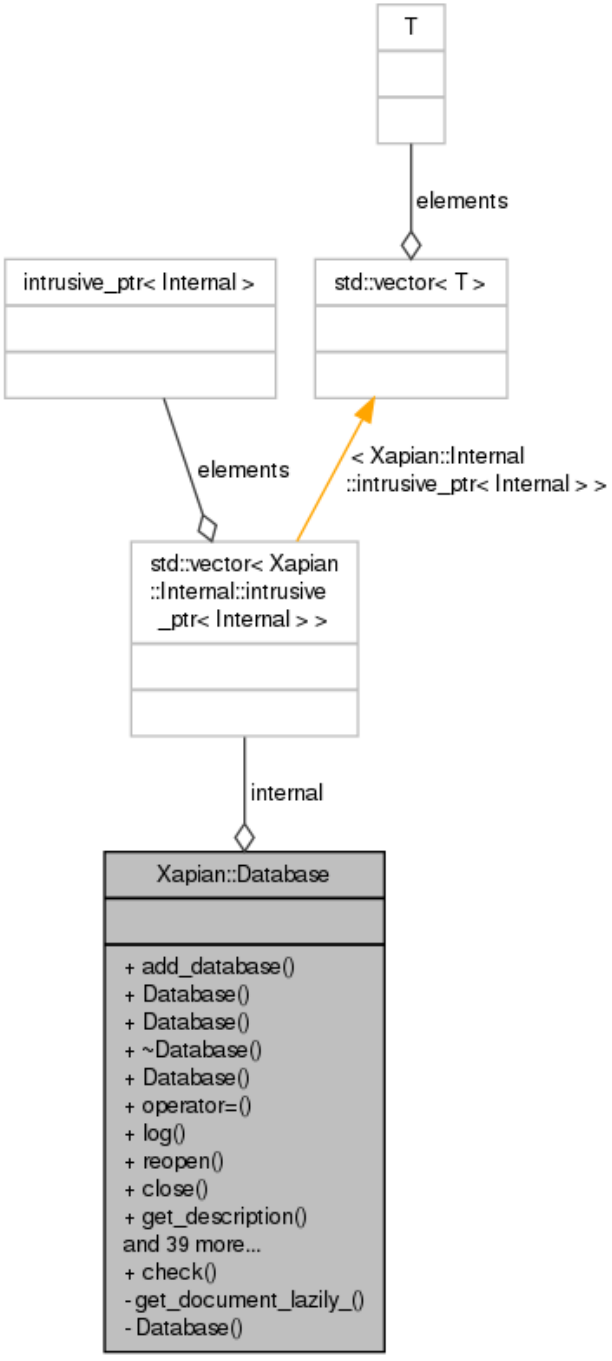


Integration of new modules

To integrate any new module, we have to change:

- include/xapian.h - This is the main header file that needs to be included by any application using Xapian.
- include/Makefile.mk - This is to add the header file to the set of files that get compiled.
- api/Makefile.mk - This is to add the .cc file to the set of files that get compiled.

Figure 3: Structure of class Database

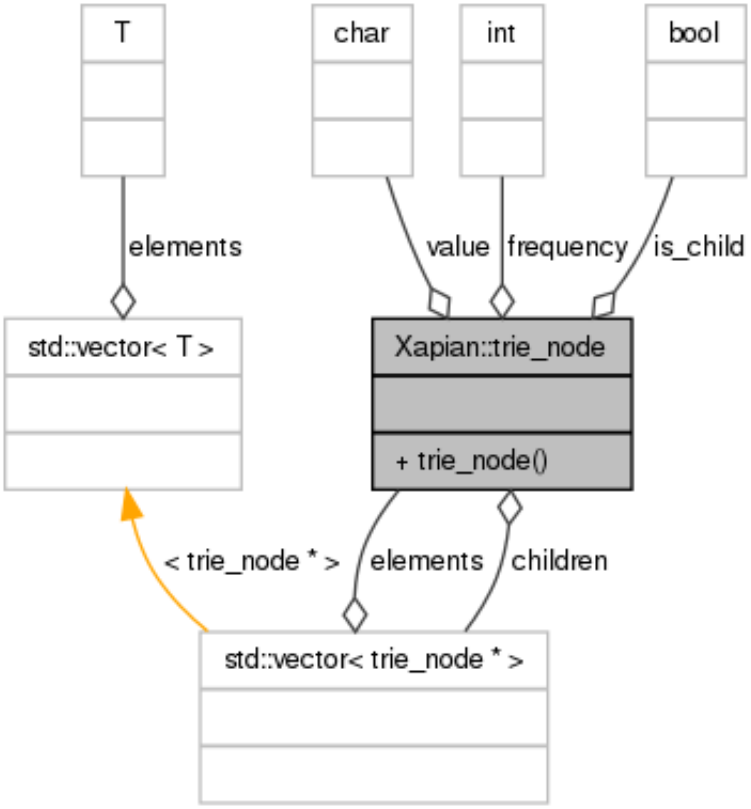


3.2 WORDNET INTEGRATION

Wordnet has been integrated in the Xapian project for supporting query expansion.

Wordnet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations.

Figure 4: Structure of `struct trie_node`



WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings. However, there are some important distinctions. First, WordNet interlinks not just word forms—strings of letters—but specific senses of words. As a result, words that are found in close proximity to one another in the network are semantically disambiguated. Second, WordNet labels the semantic relations among words, whereas the groupings of words in a thesaurus does not follow any explicit pattern other than meaning similarity.

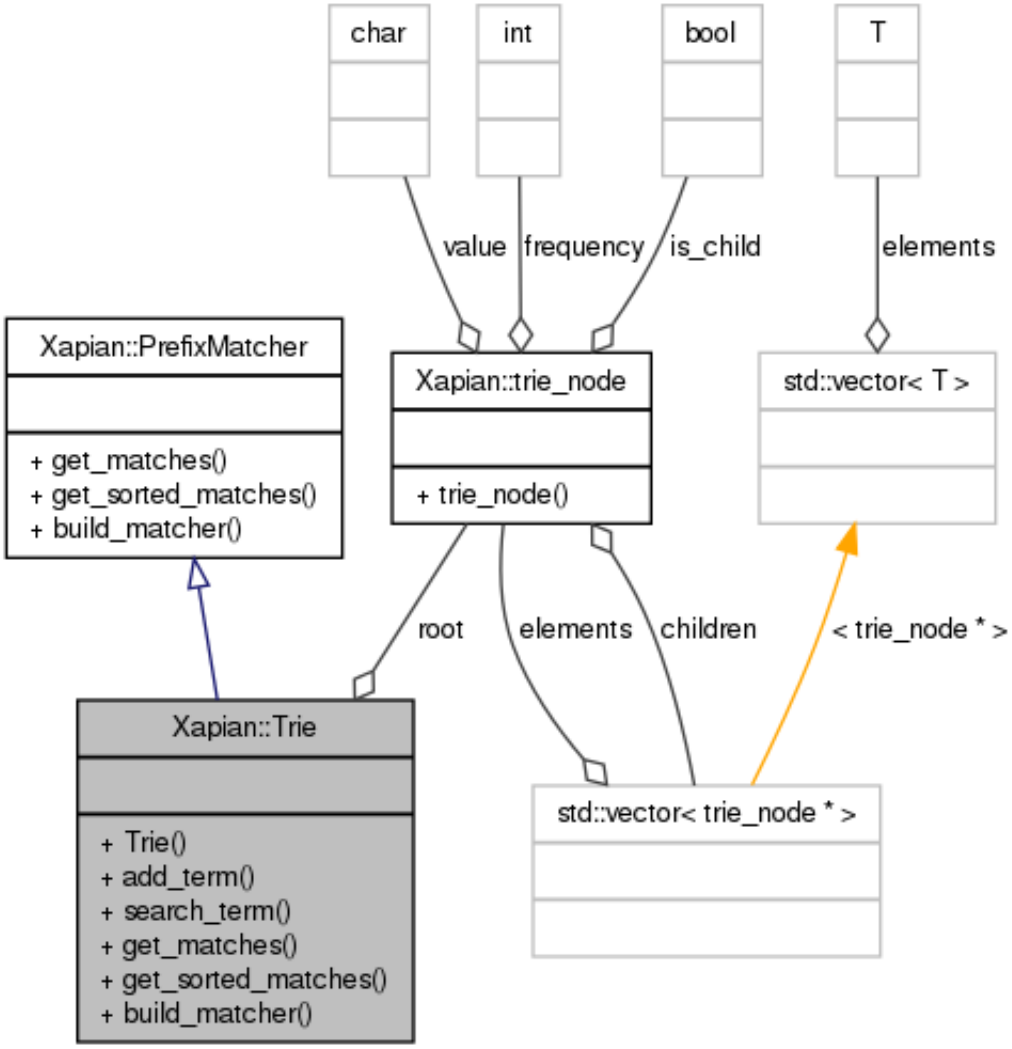
3.2.1 Structure of WordNet

The main relation among words in WordNet is synonymy, as between the words shut and close or car and automobile. Synonyms–words that denote the same concept and are interchangeable in many contexts–are grouped into unordered sets (synsets). Each of WordNet's 117 000 synsets is linked to other synsets by means of a small number of "conceptual relations." Additionally, a synset contains a brief definition ("gloss") and, in most cases, one or more short sentences illustrating the use of the synset members. Word forms with several distinct meanings are represented in as many distinct synsets. Thus, each form-meaning pair in WordNet is unique.

Relations

The most frequently encoded relation among synsets is the **super-subordinate relation** (also called hyperonymy, hyponymy or ISA relation). It links more

Figure 5: Structure of class Trie



general synsets like {furniture, piece_of_furniture} to increasingly specific ones like {bed} and {bunkbed}. Thus, WordNet states that the category furniture includes bed, which in turn includes bunkbed; conversely, concepts like bed and bunkbed make up the category furniture. All noun hierarchies ultimately go up the root node {entity}. Hyponymy relation is transitive: if an armchair is a kind of chair, and if a chair is a kind of furniture, then an armchair is a kind of furniture. WordNet distinguishes among Types (common nouns) and Instances (specific persons, countries and geographic entities). Thus, armchair is a type of chair, Barack Obama is an instance of a president. Instances are always leaf (terminal) nodes in their hierarchies.

Meronymy, the part-whole relation holds between synsets like {chair} and {back, backrest}, {seat} and {leg}. Parts are inherited from their superordinates: if a chair has legs, then an armchair has legs as well. Parts are not inherited "upward" as they may be characteristic only of specific kinds of things rather than the class as a whole: chairs and kinds of chairs have legs, but not all kinds of furniture have legs.

Verb synsets are arranged into hierarchies as well; verbs towards the bottom

of the trees (troponyms) express increasingly specific manners characterizing an event, as in {communicate}-{talk}-{whisper}. The specific manner expressed depends on the semantic field; volume (as in the example above) is just one dimension along which verbs can be elaborated. Others are speed (move-jog-run) or intensity of emotion (like-love-idolize). Verbs describing events that necessarily and unidirectionally entail one another are linked: {buy}-{pay}, {succeed}-{try}, {show}-{see}, etc.

Adjectives are organized in terms of antonymy. Pairs of "direct" antonyms like wet-dry and young-old reflect the strong semantic contract of their members. Each of these polar adjectives in turn is linked to a number of "semantically similar" ones: dry is linked to parched, arid, dessicated and bone-dry and wet to soggy, waterlogged, etc. Semantically similar adjectives are "indirect antonyms" of the central member of the opposite pole. Relational adjectives ("pertainyms") point to the nouns they are derived from (criminal-crime).
There are only few adverbs in WordNet (hardly, mostly, really, etc.) as the majority of English adverbs are straightforwardly derived from adjectives via morphological affixation (surprisingly, strangely, etc.)

3.2.2 Technical Details

To integrate WordNet, we downloaded the latest version of WordNet, *WordNet 2.1*, from `http://wordnet.princeton.edu/wordnet/download/`. Installing the package creates a static library that needs to be linked with Xapian. We had to change *xapian – core/Makefile.am* to include WordNet library to the statically linked libraries. We just had to include *wn.h* to access WordNet functions. Also, *winit()* needs to be called to be able to access the WordNet database.

3.3 SYNONYM BASED QUERY EXPANSION

One of the components of our query expansion phase is finding synonyms of words in the query and offering them as suggestions to the user. For this purpose, we have made use of WordNet, the most popular open-source lexical database for English.

The synonym expansion functionality is implemented in the *SynonymExpansion* class. This whole setup is a part of *decoratorpattern*. This will be elaborated further in dection 6.
Once a query is supplied to the *SynonymExpansion* class, it undergoes processing in the following manner:

- *Stemming* - It is the process for reducing inflected (or sometimes derived) words to their stem, base or root form—generally a written word form.

- *Stop word removal* - This functionality was extended since while Xapian is designed to take stop words into account while performing query processing, it does not include a readily usable implementation of the *SimpleStopper* class. Hence, we created a *PopulatedSimpleStopper* class that extends this class to include a list of common English stop words.

- Synonym expansion - After pre-processing, the WordNet database is queried for the synonyms of each term in turn. Since WordNet returns the synset for a word as a list of senses ordered by their frequency of occurrence in normal usage, we only consider the top few (two) senses for the purpose of expansion. It should be noted that the context in which the word is used is not taken into account as of now. Currently we are only expanding Nouns. For each sense, we choose one synonym and create an autocomplete suggestion by replacing the term by its synonym in the original query.

3.3.1 Technical Details

SynonymExpansion inherits QueryExpansionDecorator class. It only has to implement the expand() function. The following operations can be performed on the class:

- get_original_query() - Returns the original query to be expanded.
- get_results() - Returns all the expanded queries as a vector of strings.
- expand() - Expands the query and appends the results to the results data member that already contains the results generated by the objects that is passed to this decorator.
- results - This is a protected data member that contains the results.
- original_query - This is a protected data member that contains the original query.

Figure 6: Inheritance of class PopulatedSimpleStopper

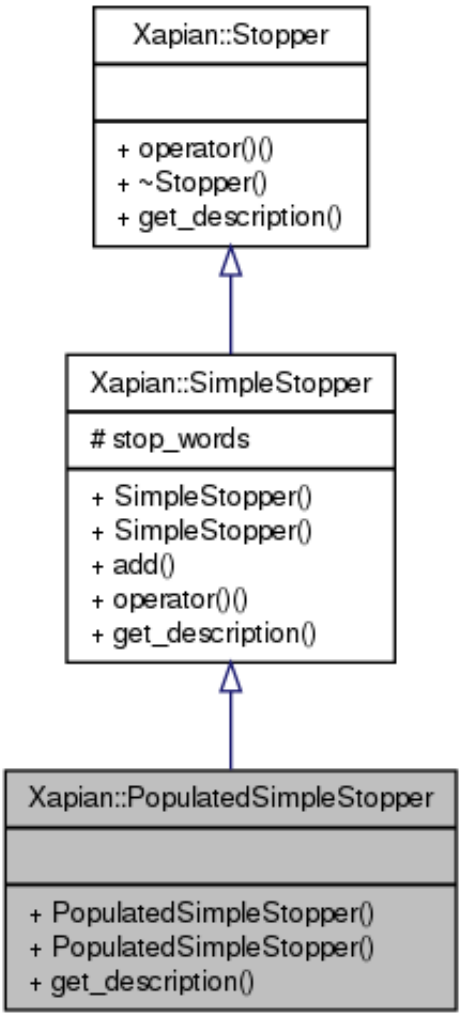


Figure 7: Structure of class PopulatedSimpleStopper

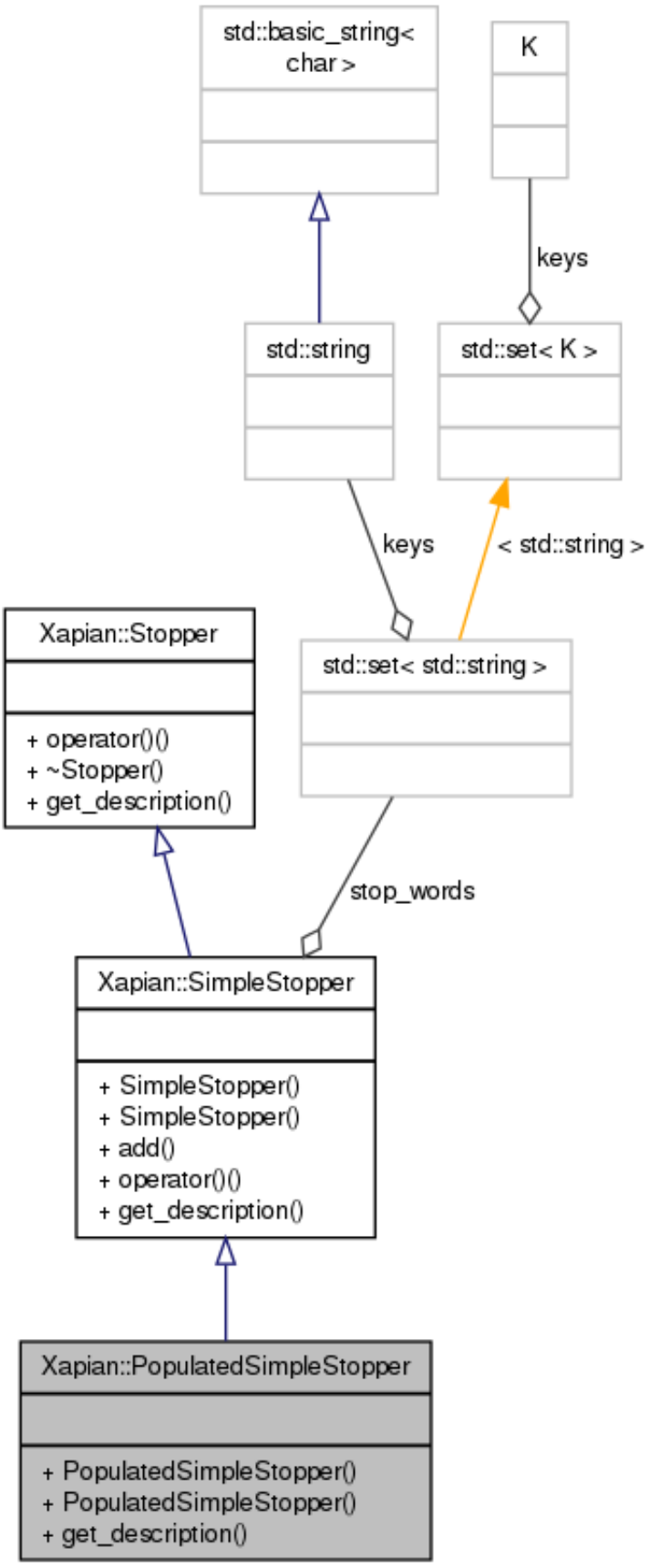


Figure 8: Structure of class SynonymExpand

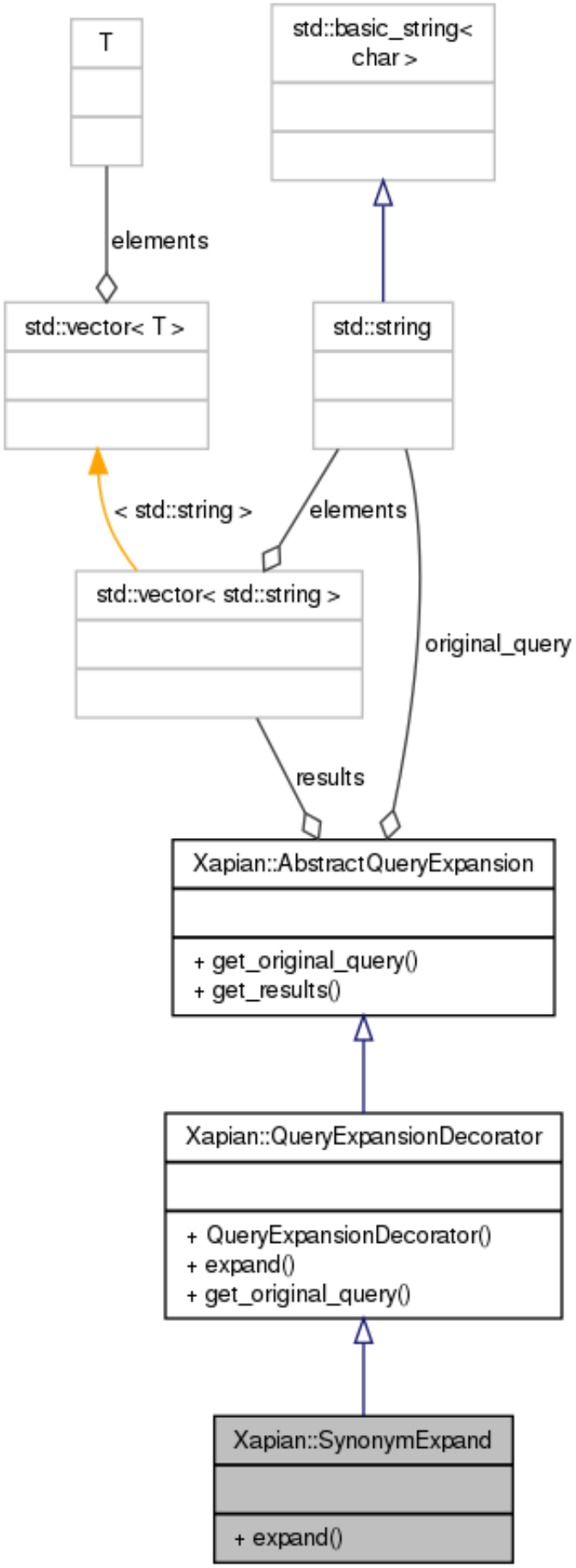
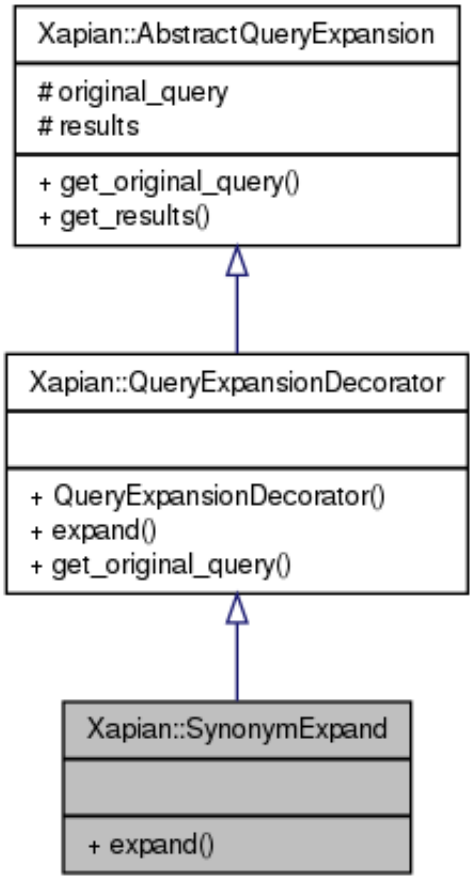


Figure 9: Inheritance of class SynonymExpand



4 DIFFICULTIES

Some of the difficulties we faced are:

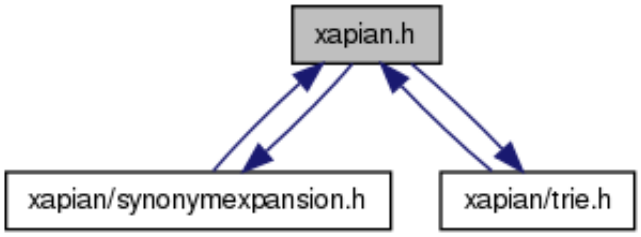
- Finding the optimum position to insert new code due to the large existing codebase.
  Xapian is bigger that any previous codebases that we have worked with. To seach for an optimum point for insertion of a new bit of code was a big task. Even a change as simple as logging of queries took some brainstorming to agree upon the method of execution.
- Integrating WordNet with Xapian.
  Integrating two existing large-scale projects is difficult. To integrate WordNet libraries and databse into the existing build framework of Xapian was very challenging. The developers of Xapian were a huge help in this regard.
- Linking of GUI made it Qt with Xapian.
  Using the library we built from the point of view of the user was eye-opening. We got interesting insights into the API we created.

5 LEARNINGS

Working on Xapian was a rewarding experience. Some of our take-aways are listed below:

- Experience of working with an existing large code base of an Open Source Project.
- To inspect the library from a user's point of view, we adopted the role of a user. We created a GUI using Qt which implemented Query Auto-completion and Query-expansion on a sample database. This was a very rewarding experience as we got important feedback about the usability of the API as well as the performance of our library.
- Interaction with the Open Source developer community.
  Integrating WordNet with Xapian would have been extremely difficult without the support from the developers at Xapian. Understanding the whole build framework would have been really hard to accomplish.
- Qt had to be learnt for creating the GUI.
- How to integrate two different projects.
  We had to integrate WordNet with Xapian. We also made a GUI on top of Xapian. These tasks taught us a lot about GCC flags, linker errors, etc.
- Using doxygen to create call graphs.
  Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL, Tcl, and to some extent D.
- Generating call graphs using Doxygen alerted us to the fact that we were including header files recursively. This prompted us to pay attention to the include graphs, inheritance graphs , call graphs etc. generated using Doxygen.

Figure 10: Recursive inclusion of header files



6 CONCEPTS RELATED TO POSE

We have tried to incorporate different design patterns in the new modules that we have added to Xapian.

A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

The uses of design patterns are:

- Mapping and Creating Objects.
- Creating complex object structures.
- Reading complex structures.
- Implementing dynamic (polymorphic) classes.
- Communication between objects in the project.
- Communication with other systems.

6.1 FLYWEIGHT PATTERN

Flyweight pattern has used in designing our code to reduce memory usage.A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the flyweight objects temporarily when they are used.

The whole trie data structure in our code is represented using a light weight

pointer to its root node rather than all nodes of the tree, thus saving lot of memory. The data member root is a pointer of type $struct trie_node$ and points to the root of the trie tree.

6.2 DECORATOR PATTERN

Decorator pattern allows to add new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new decorator class that wraps the original class.

The original AbstractQueryExpansion class is subclassed into a QueryExpansionDecorator class and a SimpleQueryExpansion class. QueryExpansionDecorator is further subclassed into two classes, with SynonymExpand and HomonymExpand being the two decorators. We haven't implemented HomonymExpand but have created an empty class to give future developers an idea about the pattern.

This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s). The decorators and the original class object share a common set of features.

6.3 FACTORY METHOD PATTERN

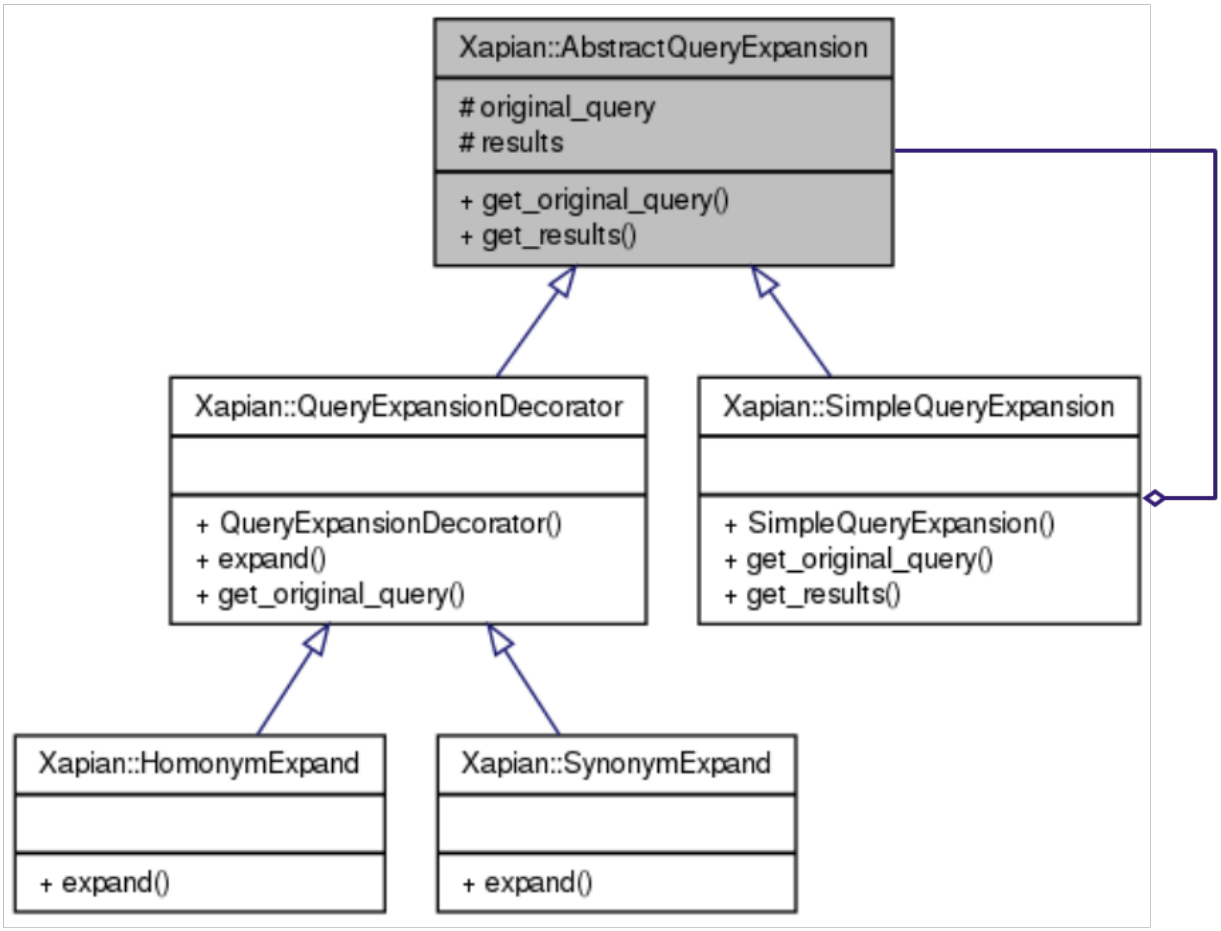
Factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. This is done by creating objects via a factory method, which is either specified in an interface (abstract class) and implemented in implementing classes (concrete classes); or implemented in a base class, which can be overridden when inherited in derived classes; rather than by a constructor.

We have created a PrefixMatcher class that is subclassed into Trie and HashTable. We have also created a PrefixMatcherFactory class that takes in the type of the prefix matcher needed and outputs a pointer to the appropriate matcher. Similar to HomonymExnd, we haven't implemented the HashTable class but have left it empty for future reference.

6.4 ACCESS MODIFIERS

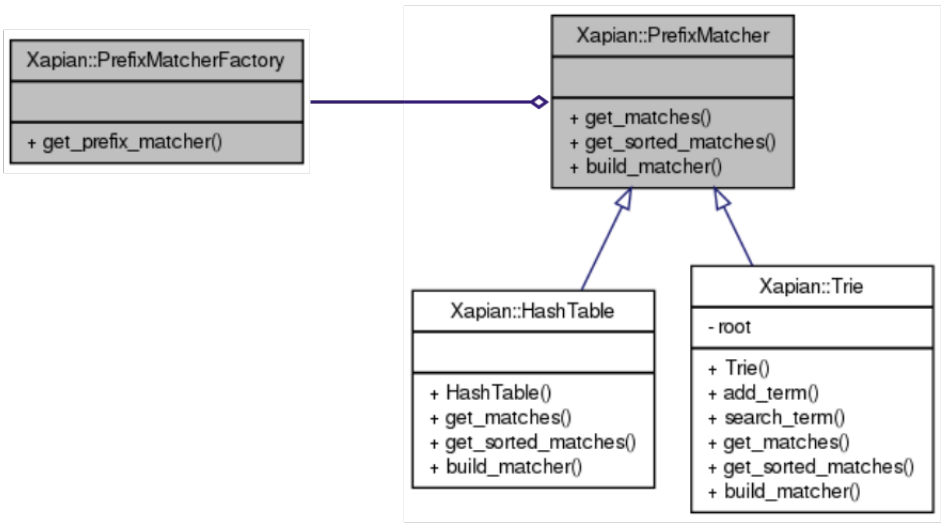
We have used the public, private and protected access modifiers appropriately. For examples:

Figure 11: Decorator pattern



- root is private is Trie class. No one can access the tree in general, but can only get the prefix matches.
- original_string and results are protected in AbstractQueryExpansion class as this restricts the access at the same time allowing it to be inherited.
- stop_words is protected in PopulatedSimpleStopper class thus allowing it to be inherited.

Figure 12: Factory pattern



BIBLIOGRAPHY

Doxygen

v 1.8.7 *Tool for generating documentation*, http://www.stack.nl/~dimitri/doxygen/.

Project, Qt

v 5.2.1 *Cross-platform application and UI framework*, http://qt-project.org/.

WordNet

v 2.1 *Lexical database of English*, http://wordnet.princeton.edu/.

Xapian

v 1.3.1 *Open Source Search Engine Library*, http://xapian.org/.