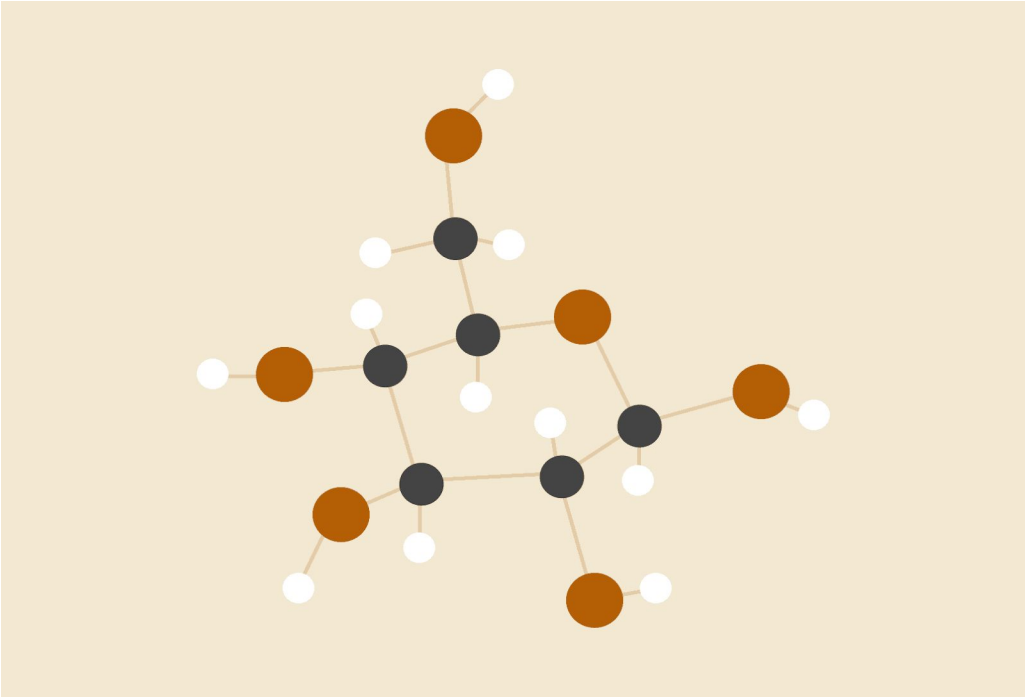
PATENTCONNECT v2.5
Design Specification

Empowering Developers, Designers, Hobbyists, & Alike



DEVELOPERS:

Yousef Hashemi <alhashey@mcmaster.ca>

Jatin Chowdhary <chowdhaj@mcmaster.ca>

Steven Gonder <gondes1@mcmaster.ca>

Yiding Li <liy443@mcmaster.ca>

Varun Verma <vermav6@mcmaster.ca>

LAB: 2 → **GROUP:** 5

By virtue of submitting this document, we electronically sign & date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including our name and student number) for future teaching purposes.

Revisions

Version	Revision
0.0	Originally planned to use B-Tree as the underlying data-structure for the graph due to space concerns. The tree would store collections of Patents, based on connected components.
0.1	Decided against using connected components, as the graph is a digraph, and there do not exist any strongly connected components. Also there was the risk that if the graph was treated as if edges were not directed, then the connected components would become too large, and would not be human-readable.
1.0	Switched to a Red-Black tree, as it was found that the data structure could be stored in memory.
1.1	Child node citations were not hyperlinked.
1.2	Switched to a bidirectional graph so that paths could be found forwards and backwards.
1.3	Instead of fetching more information upon request from Google Patent's database, we link to the specific patent for more information.
1.4	Switched back to a unidirectional graph as the bi-directional graph took up too much space.
1.5	Switched back to a unidirectional graph as the bi-directional graph took up too much space.
2.0	Switched to a Hash based adjacency list due to performance reasons.
2.1	Patent dates were not used to check for impossible patent relations.
2.2	Made labels on user interface interactive
2.3	Added expandable pictures of the patents to the user interface
2.4	Made the graph interactive with clickable nodes and drag-and-drop compatible visualization.
2.5	Added merge sort for the patent citations

Contributions

Name	Role	Contributions (Packages)	Comments
Yousef Al Hashemi	Team Leader	com.patentconnect.backend.db, com.patentconnect.backend.googleAPI	Helped with unit testing and integrating the backend with the gui. Found lots of edge cases for testing
Steven Gonder	Client Interface	com.patentconnect.gui	Implemented the graph interface, along with the project readme. Helped with design decisions.
Jatin Chowdhary	Lead Designer	com.patentconnect.gui, com.patentconnect.tools	Helped with unit testing.
Yiding Li	Developer	com.patentconnect.backend.db, com.patentconnect.backend.global	Although some of the code was removed in the final implementation, Yiding helped a lot with the final version, and ideas for the algorithms used.
Varun Verma	Unit Tester	com.patentconnect.junit	Made Unit test classes and did documentation for them.

Executive Summary

This Design Specification document contains information on PatentConnect; its classes, modules, methods, and other relevant information, such as a UML diagram.

PatentConnect is a patent landscaping application that allows everyone, from the intellectual-property (IP) curious to patent holders, to get a better understanding of the relationship between patents, while simultaneously providing relevant information about the patents. Patent relations are illustrated through patent citations, and information is provided via Google Patents. PatentConnect uses the Stanford Patent Citation Network dataset. PatentConnect is built using Java, allowing for portability across a wide array of computational devices, most notably the personal computer. The goal of PatentConnect is to democratize the patent landscaping industry, and empower (potential) developers, designers, hobbyists, in a multitude of ways. PatentConnect's powerful graphical interfaces allows anyone, even with untrained eyes, to understand Patent relations, by providing visualizations of patent connections, vis-à-vis, citations.

Table Of Contents

Interface Description

The program's code was split into different packages; all located under *com.patentconnect*. In this package, the main components are: *backend*, *gui*, and *tools*.

The backend is composed of three packages, all under the backend package. The three packages are db, global, and googleAPI. The global interface is comprised of one class, PregenSettings which is used by the other backend packages, and contains many constants that are likely to change, such as the MAX_DEPTH that the Breadth First Search used to compute the shortest path will go to compute the path. The googleAPI package also consists of one class, GooglePatents which sends queries to Google Patents in order to get information from it for the front end. It was determined that this should be separate from the db package because it is not involved in any operations in the database, and is only called much later after the patents were formed into the database.

Finally the db package, which was the largest package in the backend, consisting of 6 classes, Patent, CitationRecord, PatentFileParsingReader, CitationPath, MergeSort, and PatentDatabase. Patent was an object used to hold the most rudimentary details about patents that could be used with GooglePatents later to find out more. CitationRecord was a simple class that mostly acted as a directed edge for the PatentDatabase. PatentFileParsingReader was a wrapper for the scanner class that made it easier for the database to be read into, by returning lines as CitationRecords. CitationPath held the Breadth First Search implementation to find the shortest path. It was determined to split the shortest path algorithm off from PatentDatabase into CitationPath as PatentDatabase was already rather large, and having CitationPath would make the path easier to pass around. PatentDatabase was the main class for storing the database, and did this by maintaining an adjacency list made from a hashMap that mapped patentID's to a list of patentID's. It could be built by passing in a PatentFileParsingReader, and could get various Patents that were in the database, and return them as a Patent object whose list of citations were sorted using MergeSort. MergeSort was split into another class for a similar reason that CitationPath was, and only had one public method for sorting strings using the merge sort algorithm.

The package, *gui*, contains all of the user interfaces. There are three user interfaces: *SearchInterface*, *InformationInterface*, and *GraphInterface*.

- *SearchInterface*: Contains a text field, prompting for a Patent ID
- *InformationInterface*: Displays information on a Patent (i.e. Name, number, description, pictures, and citations). Contains a text field, prompting for a second Patent ID
- *GraphInterface*: Graphically illustrates the shortest path from first patent (SearchInterface) to the second patent (InformationInterface).

Each interface is its own class. This was done so multiple instances of the same interface can be spawned. For example, the *GraphInterface* can open up multiple instances of the *InformationInterface*. Simply clicking on a node will open up an *InformationInterface* for that patent. Similar actions can be done from other interfaces (i.e. *SearchInterface* can spawn multiple *InformationInterfaces*). Decomposing the user interfaces into three classes makes the application more accessible for the user, by allowing them to view new information, without getting rid of old information. For instance, if there are three nodes on the GraphicalInterface, the user can open up all 3 *InformationInterfaces* for the respective patents, and easily view the information. Furthermore, those windows can be left open and the user can view information for a different set of related patents.

The package, *tools*, contains helper functions and information used across all other packages and their respective classes. The classes in *tools*, are:

- *ImageLoader*: Loads image assets.
- *InformationGlobal*: Contains data (mostly Strings) that is used across the frontend and backend
- *Node*: An abstract data type that represents nodes
- *Parser*: Checks the validity of the format of strings

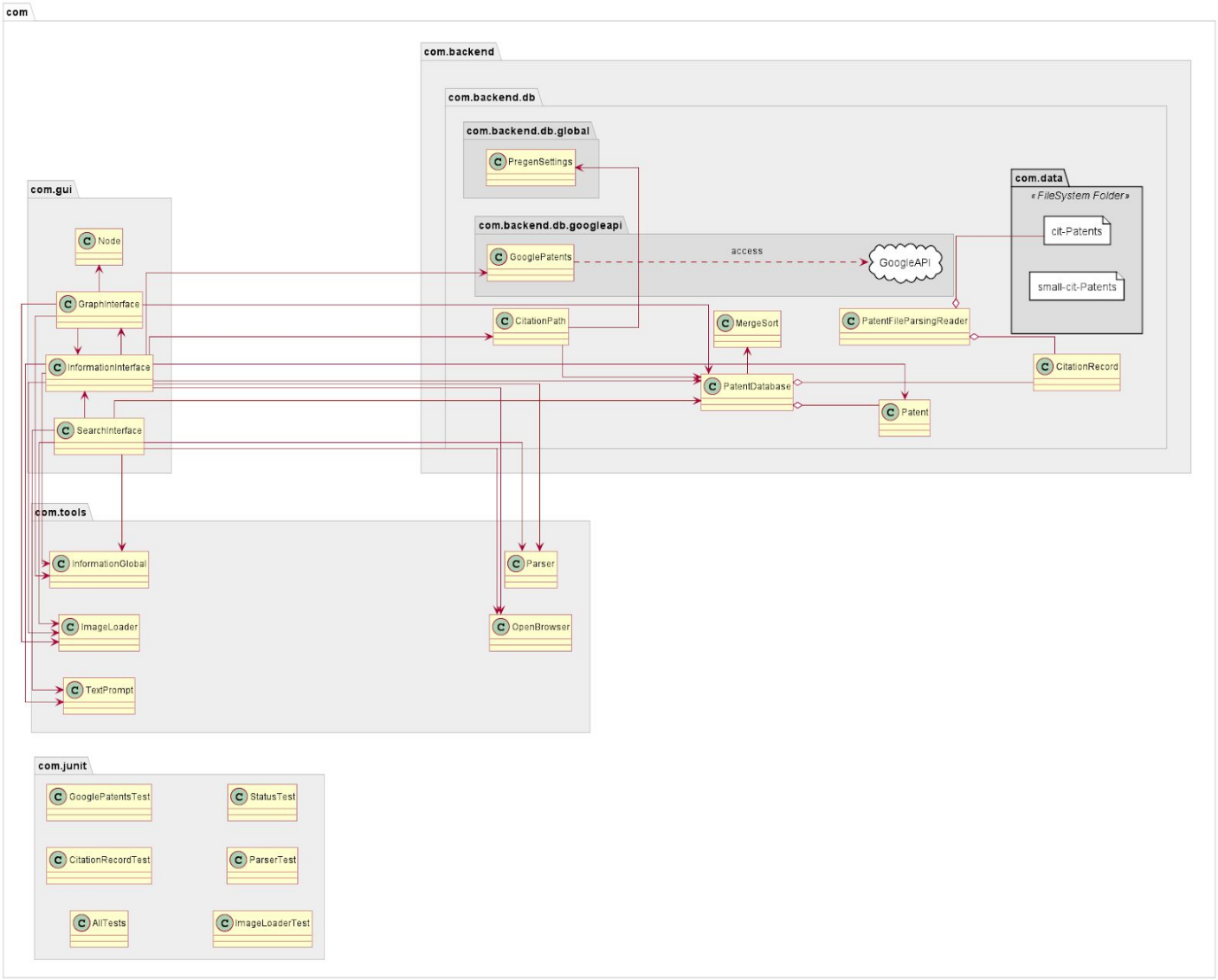
The main reason we have a *tools* package is because we want *PatentConnect* to be designed for change. Each class in *tools* can easily be modified without breaking the entire program. Here are a few examples:

- The strings in InformationGlobal can easily be modified and the changes will reflect the entire program. If we want to change the location of assets or labels, we can very easily do so, without breaking the program. For example, if we need to add, change, or delete a message, we can effortlessly apply the changes to *InformationGlobal*.
- The class *ImageLoader* is used across all user interface classes to load images (mostly icons) and patent thumbnails. If we ever encounter a problem with loading images, we will only need to fix one class. If we decide to add images to the other interfaces, we can easily do so.
- The *Parser* class can be modified to accept more strings and these changes will immediately reflect the *SearchInterface* and *InformationInterface*. For example, if we want the application to accept Patent IDs with spaces in the beginning or end (i.e.

__123456__ (Spaces are represented by underscores)). We can add the respective code to *Parser*. Similarly, we can modify *Parser* to accept floating point numbers (i.e. 123456.0). These changes can swiftly be done without modifying the program.

- The *Node* class is an abstract data type and can be used elsewhere. For example, if we wanted to show multiple paths between patents, the *Node* class can be seamlessly implemented.

UML Diagram for PatentConnect Class Relations



Interface Specification

Backend Modules

PregenSettings

Module
PregenSettings

Uses
None

Syntax

Exported Constants
MAX_LINE_MEMBERS = 2
CITATION_SEPARATOR = "\\s+"
DATA_FILE_PATH = "src/com/patentconnect/data/cit-Patents.txt"
MAX_DEPTH = 100

CitationRecord

Template Module
CitationRecord

Uses
None

Syntax

Exported Types
CitationRecord = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
CitationRecord	String, String		
getFrom		String	
getTo		String	

Semantics

State Variables

from : String

to : String

Access Routine Semantics

CitationPath(startPatent, endPatent):

- Description: Takes in two patentIDs as Strings that represent an edge in the database
- Transition: *from, to* := startPatent, endPatent
- Output: self
- Exception: None

getFrom()

- Description: Retrieves the patentID of the citing patent in the citation.
- Transition: None
- Output: *from*
- Exception: None

getTo()

- Description: Retrieves the patentID of the cited patent in the citation.
- Transition: None
- Output: *to*
- Exception: None

Patent

Template Module
Patent

Uses
None

Syntax

Exported Types
Patent = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Patent	String, seq of String		
getPatentID		String	
getCitations		Seq of String	

Semantics

State Variables

patentID : String
citations : seq String

Access Routine Semantics
Patent(patentID, cit):

- Description: Takes in a patentID and the patentID's of a
- Transition: *from, to* := startPatent, endPatent
- Output: self
- Exception: None

getPatentID()

- Description: Retrieves the patentID of the object.
- Transition: None
- Output: *patentID*
- Exception: None

getCitations()

- Description: Retrieves the citations stored in this Patent.
- Transition: None
- Output: *citations*
- Exception: None

PatentFileParsingReader

Template Module
PatentFileParsingReader

Uses
PregenSettings, CitationRecord

Syntax

Exported Types
Patent = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PatentFileParsingReader	String		FileNotFoundException
hasNextLine		Boolean	
nextLine		CitationRecord	

Semantics
State Variables
filePath : String

Access Routine Semantics
PatentFileParsingReader(path):

- Description: Opens a file to read Patents from.
- Transition: *filePath* := path
- Output: self
- Exception: FileNotFoundException, thrown if the given path does not lead to a file

hasNextLine()

- Description: Looks ahead at the file to tell if there is another line
- Transition: None
- Output: *true* if there are more lines, *false* otherwise
- Exception: None

nextLine()

- Description: Creates a CitationRecord from the line of the given data file. Uses CITATION_SEPARATOR and MAX_LINE_MEMBERS from PregenSettings to help with this.
- Transition: None
- Output: a new CitationRecord created from the next line of the data file
- Exception: None

CitationPath

Template Module
CitationPath

Uses
PatentDatabase, PregenSettings

Syntax

Exported Types
CitationPath = ?

Exported Access Programs

Routine name	In	Out	Exceptions
CitationPath	PatentDatabase, String, String		
getPath		Seq of String	

Semantics

State Variables
path : seq of String

Access Routine Semantics
CitationPath(patentDB, startPatent, endPatent):

- Description: Builds a shortest path between the startPatent and the endPatent, up to the MAX_DEPTH in PregenSettings using Breadth First Search. startPatent and endPatent are two strings in the form of patentID's, but if they aren't then shortest path will be null
- Transition: *path* the shortest path between the startPatent and endPatent.
- Output: self
- Exception: None

getPath()

- Description: retrieves the shortest path created from the constructor.
- Transition: None
- Output: *path*
- Exception: None

MergeSort

Module
MergeSort

Uses
None
Syntax

Exported Types
CitationPath = ?

Exported Access Programs

Routine name	In	Out	Exceptions
sort	Seq of String	Seq of String	

Semantics

State Variables
None

Access Routine Semantics
sort(arr):

- Description: Uses the merge sort algorithm to sort a sequence of strings
- Output: newArr such that $(\forall\ ele\ :\ String \mid ele\ \varepsilon\ arr\ =>\ ele\ \varepsilon\ newArr) \wedge (\forall\ i\ :\ \aleph \mid i\ \varepsilon\{0...|newArr|-2\}\ =>\ newArr[i]\ <\ newArr[i+1])$, where '<' is the lexicographic ordering of the Strings in arr
- Exception: None

PatentDatabase

Template Module
PatentDatabase

Uses
Patent, PatentFileParsingReader, CitationRecord, MergeSort

Syntax

Exported Types
PatentDatabase = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PatentDatabase	PatentFileParsingReader		
getPatent	String	Patent	
contains	String	boolean	
size		Integer	
shortestPath	String, String	CitationPath	

Semantics

State Variables
adjList : Map of String to seq of String

Access Routine Semantics
PatentDatabase(reader):

- Description: Builds the database from the reader, by reading all of the CitationRecords from the reader into the adjacency list *adjList*
- Transition: *adjList* becomes an adjacency list containing all of the edges in the input file
- Output: self
- Exception: None

getPatent(patentID)

- Description: retrieves patent object from the database with the matching patentID, and makes sure the list it stores is sorted by using the MergeSort class.
- Transition: None

- Output: (self.contains(patentID) => new Patent(patentID, MergeSort.sort(*adjList*[PatentID])) | true => null)
- Exception: None

contains(patentID)

- Description: takes a given patentID and returns whether it is in the database or not
- Transition: None
- Output: true if the patentID is in *adjList* false otherwise
- Exception: None

size()

- Description:gets the number of nodes in the database
- Transition: None
- Output: |*adjList*|
- Exception: None

shortestPath(startPatent, endPatent)

- Description: takes a starting patent and and ending patent and finds the shortest path between them by using the CitationPath class
- Transition: None
- Output: new CitationPath(self, startPatent, endPatent)
- Exception: None

GooglePatents

Template Module

GooglePatents

Uses

None

Syntax

Exported Types

GooglePatents = ?

Exported Access Programs

Routine Name	In	Out	Exceptions
GooglePatents	String		IOException
getPatentID		String	
getName		String	
getDescription		String	
getURL		String	
downloadPictures		Integer	IOException
arePicturesAvailable		boolean	
getURLFromPatentID	String	String	
isValidPatent	String	boolean	
deletePictures			

Semantics

State Variables

patentID : String
URLString : String
name : String
description : String
pictures[] : seq of String
picsReady : boolean

State Invariant

None

Access Routine Semantics

GooglePatents(patent):

- Description: Finds information about a given patentID from google patents, and stores it.
- Transition: *patentID, URLString* := patent, getURLFromPatentID(patent) *name* and *description* are the name and description of the given patentID respectively. *pictures* is a sequence of Strings containing the urls of the picture resources *pictures*, *name*, and *description* are all found on google patents
- Output: self
- Exception: IOException if there is some problem when accessing the webpage.

getPatentID()

- Description: retrieves the patentID of the patent stored in this object
- Transition: None
- Output: *patentID*
- Exception: None

getName()

- Description: retrieves the name of the patent stored in this object
- Transition: None
- Output: *name*
- Exception: None

getDescription()

- Description: retrieves the description of the patent stored in this object
- Transition: None
- Output: *description*
- Exception: None

getURL()

- Description: gets the URL of the Google Patents page for this patent
- Transition: None
- Output: *URLString*
- Exception: None

downloadPictures()

- Description: downloads the pictures of this patent on the Google Patents page
- Transition: Downloads pictures to a local folder,
  picsReady = true
- Output: the number of pictures downloaded
- Exception: None

arePicturesAvailable()

- Description: tells if pictures are downloaded and available for use
- Transition: None
- Output: *picsReady*
- Exception: None

isValidPatent(patentID)

- Description: tells if given patentID has a corresponding Google Patents page
- Transition: None
- Output: true if the url associated with the given patentID returns a 2xx status code when a HTTPS GET request is called to it.
- Exception: None

Tools

ImageLoader

Module
ImageLoader

Uses
None

Syntax
Exported Types
ImageLoader = ?

Exported Access Programs

Routine Name	In	Out	Exceptions
returnImage	String, String	Image	IOException

Semantics
State Variables
None

State Invariant
None

Access Routine Semantics
returnImage(imagePath, fileName):

- Description: Takes in a path and file name (the image), and tries to load it.
- Transition: None
- Output: Image
- Exception: IOException, if image could not be loaded correctly

Parser

Module
Parser

Uses
None

Syntax
Exported Types
None

Exported Access Programs

Routine Name	In	Out	Exceptions
parseExitCode	String	byte	NumberFomatException, Exception
isCorrectSize	String	boolean	

Semantics
State Variables
None

State Invariant
MIN_SIZE_PATENT_ID = 5
MAX_SIZE_PATENT_ID = 7

Access Routine Semantics

parseExitCode(parseString):

- Description: Takes in a String and check to see if it is valid input
- Transition: None
- Output: byte
- Exception:
    - NumberFormatException: If the input string could not be converted to an integer
    - Exception: For all other exceptions; this is a generic exception to catch anything else

InformationGlobal

Module

InformationGlobal

Uses

None

Syntax

Exported Types

InformationGlobal = ?

Exported Constants

ICON_PATH = path for icon
PATENT_IMAGE_PATH = image path for patent images and thumbnails
SEARCH_ICON = magnifying glass icon
LINK_ICON = chain link icon
STATUS_MSG_CONSOLE = string array of status messages intended for console
STATUS_MSG_USER_INTERFACE = string array of status messages intended for user
PATENT_THUMBNAIL_START = all patent thumbnails start with this
PATENT_FILENAME_ENDING = all patent image filenames end with this
PNG = ending extension for image

Exported Access Programs

Routine name	In	Out	Exceptions
returnStatusLabel	int	String	
returnLabelColor	int	Color	

Semantics

State Variables

None

Access Routine Semantics

returnStatusLabel(exitCode):

- Description: Returns the appropriate Status Label based on exit code of Parser. These will be displayed on the User Interfaces.
- Transition: None
- Output: $exitCode >= 0 \land exitCode < 4$
  $=> STATUS\_MSG\_USER\_INTERFACE[exitCode] \mid exitCode < 0 \lor exitCode >= 4$
  $=> STATUS\_MSG\_USER\_INTERFACE[4]$
- Exception: None

returnLabelColor(exitCode)

- Description: Returns a Color object based on the exitCode
- Transition: None
- Output: Color
- Exception: None

GUI

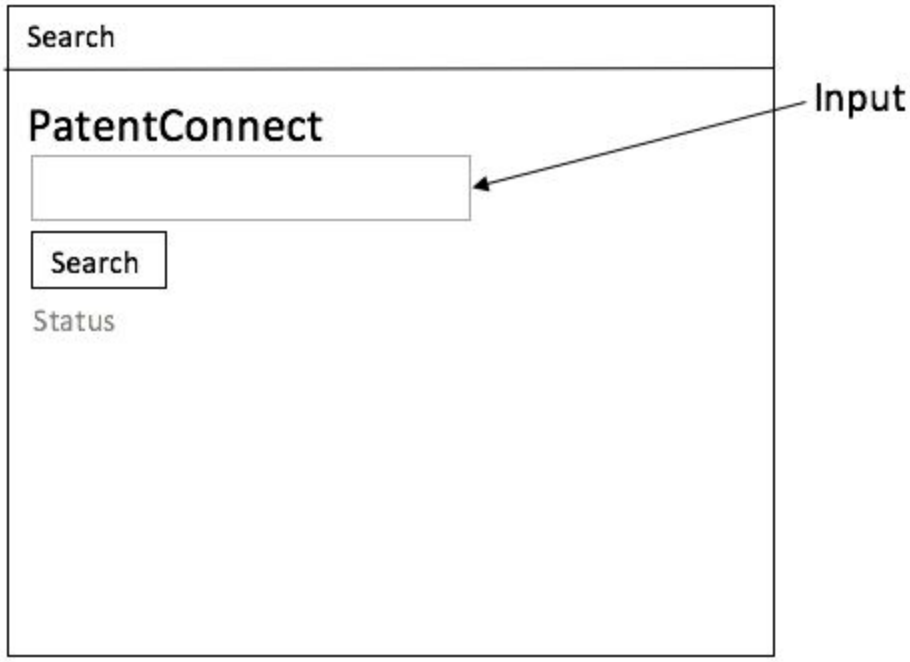
Search Interface

Module

SearchInterface

Uses

ImageLoader, InformationGlobal, Parser, PatentDatabase



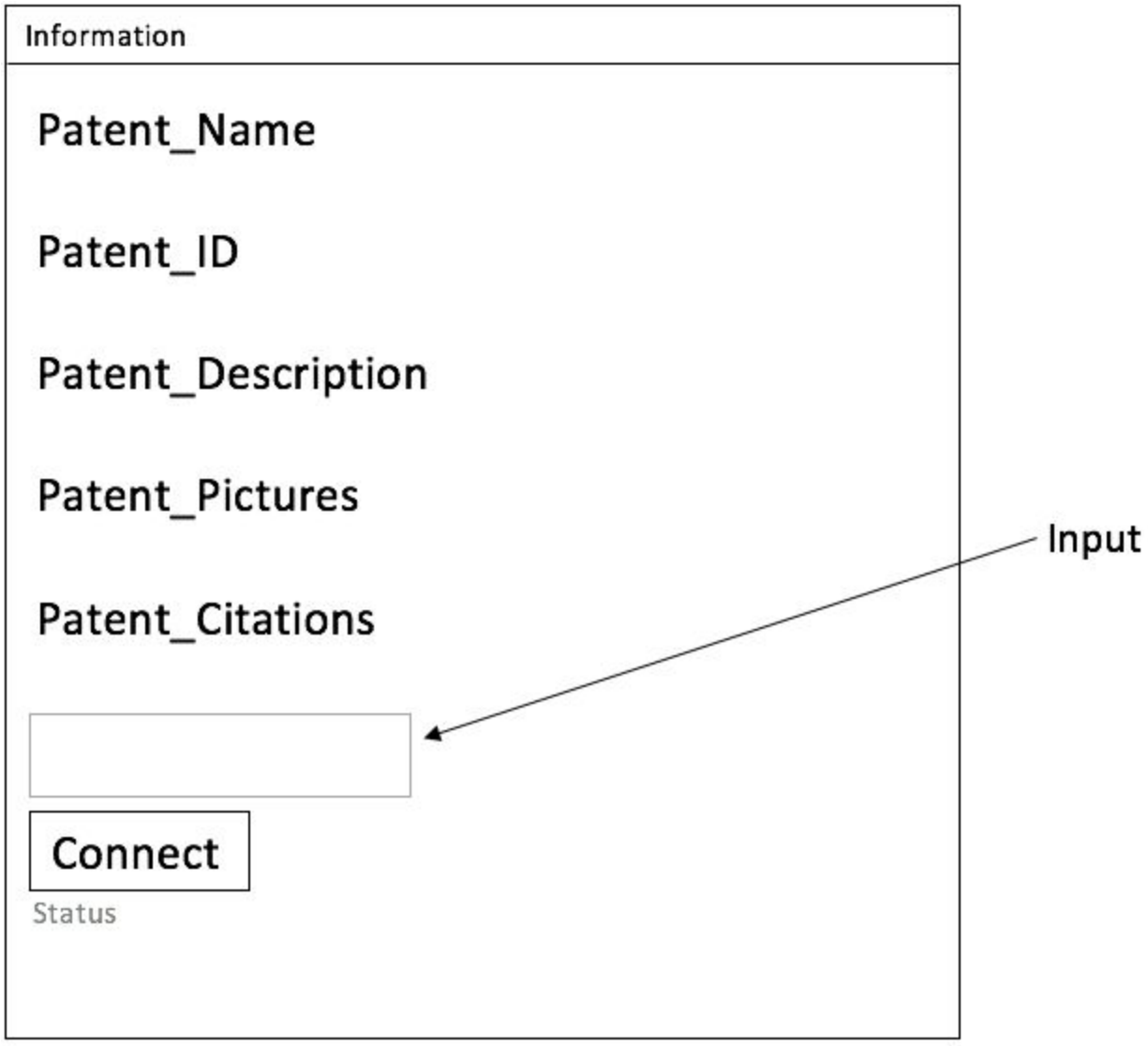
Information Interface

Module

Information Interface

Uses

ImageLoader, InformationGlobal, Parser, Patent, PatentDatabase, GooglePatents



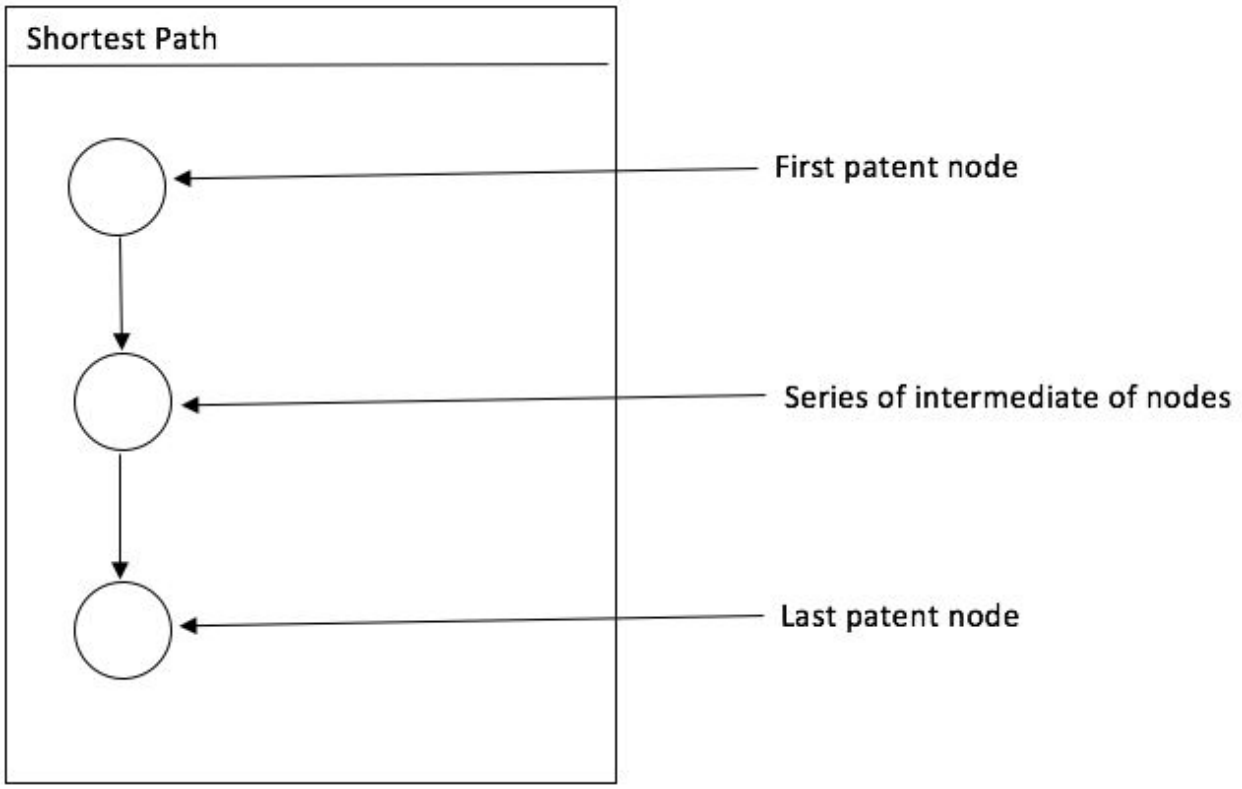
Graph Interface

Module

GraphInterface

Uses

ImageLoader, InformationGlobal, PatentDatabase



UML Machine State Diagrams

Diagram for PatentDatabase

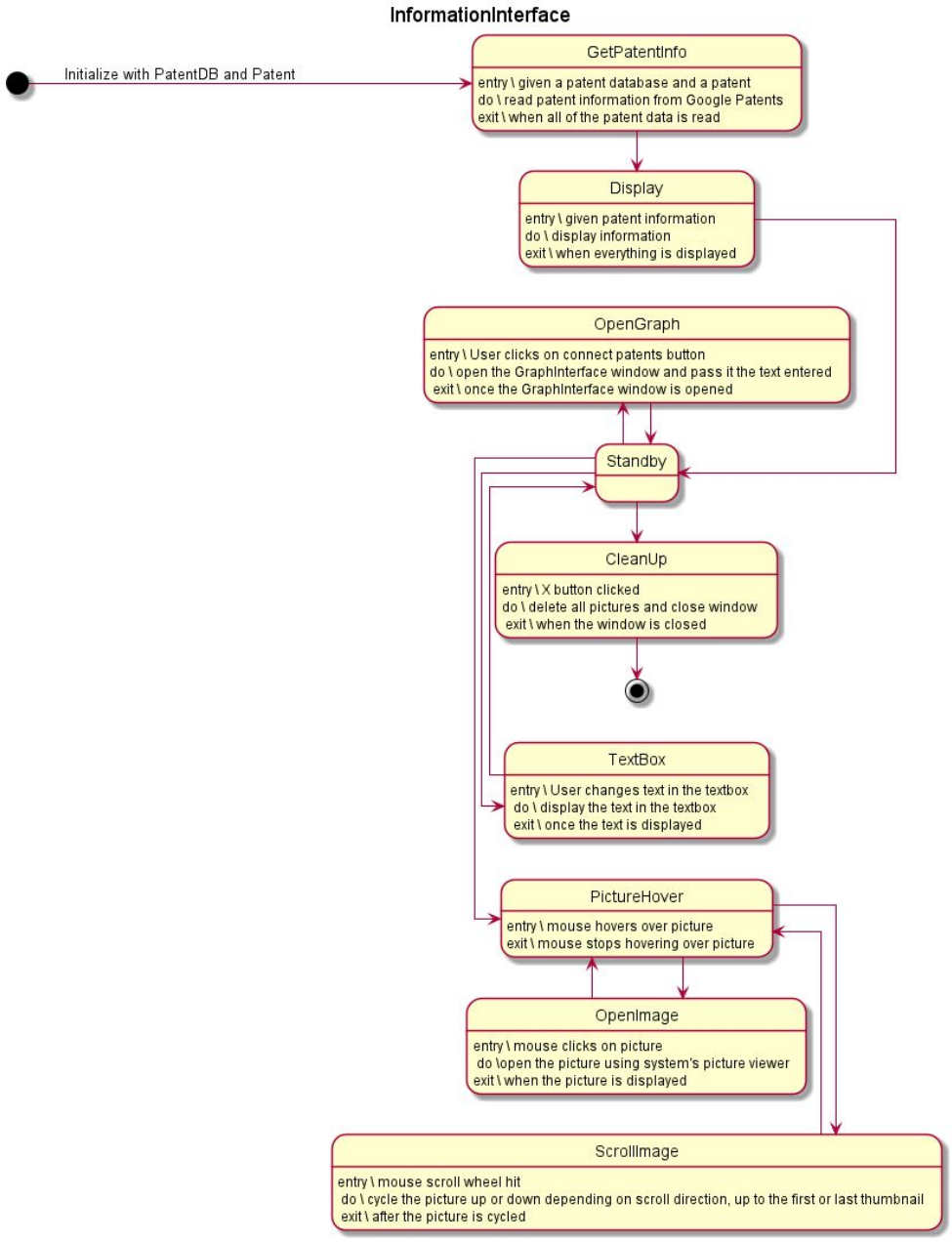
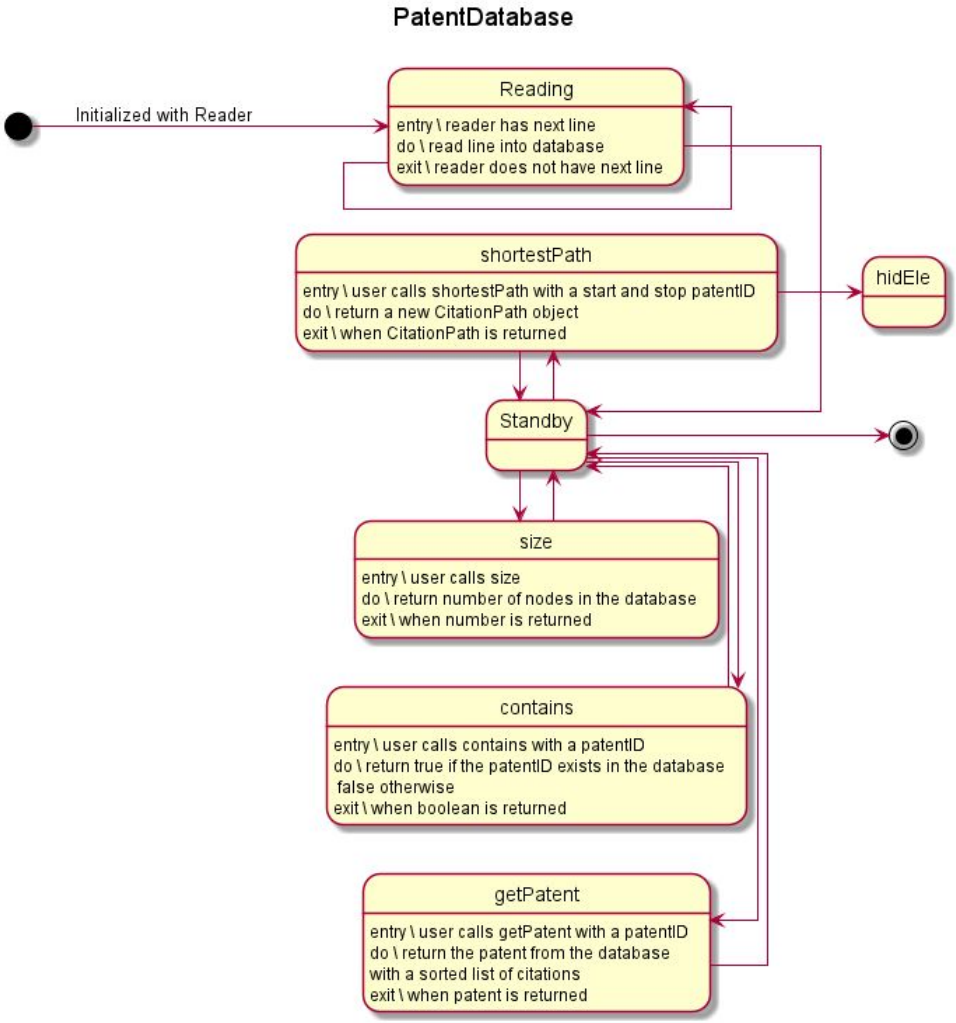


Diagram for GraphInterface



Internal Review and Evaluation

We based our design choices largely off of the textbook, *Introduction to Software Engineering*, by Ghezzi et. al. The chapter we referenced is *Software Engineering Principles*. We followed the waterfall model of the software life cycle wherever possible. Inevitably, there were times we had to “fake it.”

We weren’t able to perform a large-scale feasibility study, per “Requirements and analysis specification.” Prior to our project proposal, however, we did an analysis of similar applications on the market, most notably *Teqmine*. We considered what seemed to be the most plausible outcomes of a feasibility study, with a specific focus on how we would differentiate ourselves and serve user needs.

Our MIS was both the centrepoint and starting point of the design portion of our project. We followed our mostly unambiguous informal specification, which made understanding class hierarchies a lot easier, along with communicating expected method behaviour in a sufficiently precise way. The MIS served as a reference whenever we needed clarification as to what something should do. However, we did not formally split the process into architectural and detailed design phases.

We then began working through the coding process, implementing our design choices. Changes were made to the MIS, if appropriate. We aimed to uphold the software principles we were familiar with from Ghezzi, et al.; namely rigor and formality, separation of concerns, modularity, abstraction, anticipation of change, generality, and incrementality.

With respect to rigor and formality, we used JUnit to test a variety of edge cases, although our approach to formality left a bit to be desired; in particular, we could have used more discrete math in our MIS. We were able to effectively separate concerns, as we had our developers mostly working productively on individual components. This “pipelining” that came from connecting components was unproblematic, and we largely attribute this to how modular we made the classes and our implementation decisions early in the project. We were able to abstract away irrelevant details through the classes’ effective APIs, and we anticipated change early on, especially with realizing the need for reusability in our visual interface. For example, in the GUI, *InformationInterface* and *GraphInterface* have a cyclic uses relation, which allows us to have recursive patent search functionality, each in its own specific set of application windows. Generality, however, was less of a concern, as we realized it tended to be significantly more efficient to have most modules configured solely for their intended use. We weren’t able to get incremental feedback from prospective customers, but we were able to increment our code proficiently. The classes, *InformationInterface* and *GraphInterface,* had features added successively (i.e. With the graph, we built the nodes and edges first, then added proper labels,

then added link functionality, then finally added customizable node positioning via drag and drop). This theme, of adding features successively, permeated our project.

Integration and system testing were performed manually, which is one area we could have greatly improved on. If we were able to automate integration testing, we would have had more confidence in the reliability of the visual components of PatentConnect. Unfortunately, it was significantly harder to test the GUI than the module logic.

The submission of PatentConnect will effectively put us into the maintenance phase of the software life cycle. We are proud of the results we were able to accomplish, and hope it works as well for general users as it has in isolated testing.

Furthermore, we are especially proud of our design decisions. *PatentConnect* was designed for change. Each class can be easily modified and the changes will be reflected throughout the program. Making changes will not break the program. During the development of *PatentConnect*, we refactored the code many times and each time, we did not run into errors. These design decisions have allowed *PatentConnect* to be a very scalable application.