UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Progressive Software Design Tool

J. A. Breytenbach

Supervisor

B. Fischer

Abstract

Visualising software can be a tedious and cluttered affair with the design process and development often being out of sync. Some development methodologies even largely do away with the design entirely and focus on short bursts of coding and validation to make sure the project is still on the right track. This document focuses on deriving a methodology and subsequent tool to iteratively and progressively expand concepts, the understanding of the project and development cycles. An existing visualisation is adapted to better suit the needs of the designer by providing the ability to view the project from different layers of abstraction in one concise visualisation.[13]

Contents

1 Glossary

Project Graph

Underlying graph structure representing the data of the system being designed.

Compound Graph

A graph that contains secondary and possibly tertiary information.

Concept

A single term, idea, abstraction, or logical grouping.

Relation

A connection between two concepts,

Statement

A sentence (or part thereof) that describes a specific piece of the system that has to be designed.

Document

A collection of statements grouped together. THe document can be of a technical or natural language nature.

ANTLR

Parser Generator for Java.

JSON

JavaScript Object Notation. A structure for communication over http using the javascript framework.

XML

Extensible Markup Language. A structure for encoding information into a logical format to be parsed by differnt technologies.

Pin Layout

A lyout enabling "pinning" of speciffic controls to areas of the screen for managing information.

Pin Container

A layout containing controls to be pinned to the Pin Layout.

D3.js

A javascript library assisting in drawing of graphs and graph like structures.

2 Introduction

Designing a new system has always been a mix of creative and logical thinking. At first the designer is confronted with fragmented and sometimes incomplete pieces of information that has to be added together to create the system. Gathering your thoughts as the designer and placing all this information (without missing pieces) has always started for me as a drawing of basic concepts and relations between them. These would then be elaborated on and explored in more detail, slowly growing the conceptual view on the entire program and how it has to fit together.

Even though there are existing methodologies and tools to help with the design of a system, I felt they did not offer the flexibility to first understand the program before committing concepts to specific roles. Neither were they able to trace why design decisions were made after the program was completed and few were able to take the completed project back into the design (reverse engineering).

From these incompatibilities an idea was born to create a tool and methodology to expand the program in a generic way, delaying committing to types until the larger picture was clear and to then generate code to help guide the developer with the implementation of the design, in stead of completely trying to go from design to code.

The following document derives a methodology by looking at each step of the design process with a suggested solution to make it more intuitive while giving the user a greater amount of flexibility. After the methodology is derived a tool is built to help support the methodology.

3 Background

In this section we discuss different technologies, methodologies and concepts that will be mentioned throughout the rest of the document.

3.1 Software Design Methodologies

Software design is the practice of modelling a system to be developed through documentation and diagrams without writing code. These documents and diagrams describe different laye a rs of abstractions, key concepts and sometimes pseudo code to explain the working of specific aspects. The process helps identify potential problem areas early to avoud stumbling across them later during the development cycle. In software design (as in software testing), the later a problem is identified during the development cycle, the more costly it is to address.

3.2 Existing methodologies for software design and development

There are currently 3 basic approaches to software design, and they are as follows:

Waterfall

The waterfall model is broken up into specific steps

- Requirements Analysis
- Software Design
- Implementation
- Testing
- Integration
- Deployment
- Maintenance

This structured methodology breaks up the process into exact refinements but depends on each step to have been correct. Any oversight in one will lead to oversights in the next steps.

Prototyping

The prototyping model is based around evaluating specific aspects of the required program. Small standalone but incomplete pieces are built and presented for evaluation. After prototypes are deemed to be satisfactory in achieving all the specific functionality, a complete system is built with all of them taken into account.

Spiral

Spiral attempts to combine the waterfall model with the rapid prototyping model by having iterations of design and implementation with evaluations in between.

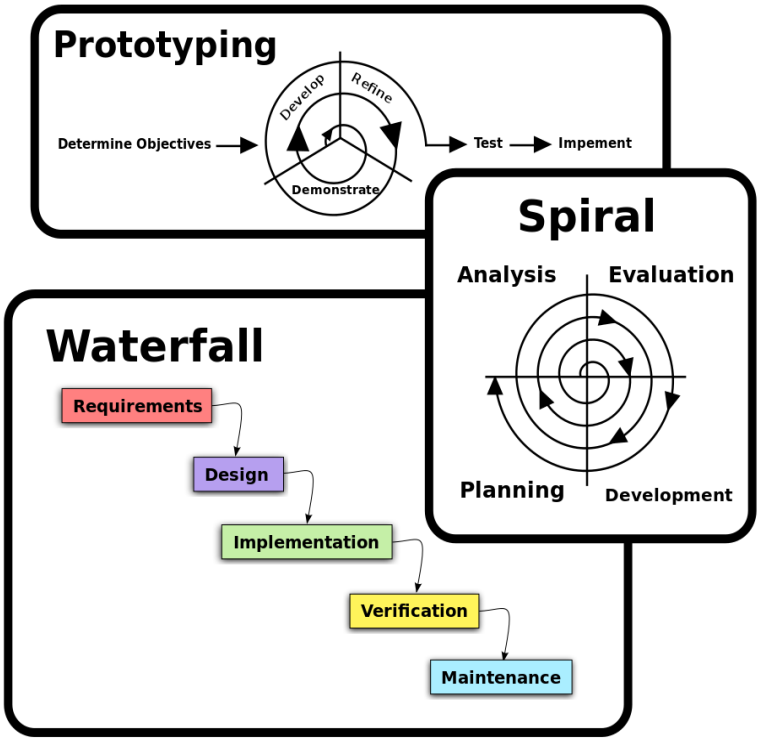


Figure 1: Software Design Methodologies and their Steps [10]

UML Documents

During the design phase of a system, concepts are often presented in a diagrammatic form to assist with the overview and understanding of the system. Of these diagrammatic presentations, UML (Unified Modelling Language) is the most popular form among software architects.

UML diagrams benefit from being a great way to communicate an idea between developers and abstracting different layers of the system for easy understanding. The fixed notations and different types of diagrams allows for anyone

familiar with UML diagrams to easily understand what it representing as well as enabling one person to do the design while another follows it during implementation.

It does however have some flaws to it as well. For large systems the diagrams become hard to maintain and keep in sync as not many tools allow for reverse engineering of an entire project to fit back in with the original documentation. These diagrams also suffer from visual clutter as placing elements in the available space while maintaining clear relations between them become a tedious task with the success of algorithms for auto layouts being limited.

Another shortcoming which is a direct consequence of one of it's strengths, is that it can not view different layers of abstractions from different components easily. Every diagram is created for a specific abstraction and interaction between specific components, requiring the user to create a new diagram each time a different subset or abstraction wants to be viewed.

A final shortcoming is the inability to directly link any specifics within a diagram back to documentation, leaving the implementation entirely up to the thoroughness and understanding of the designer.

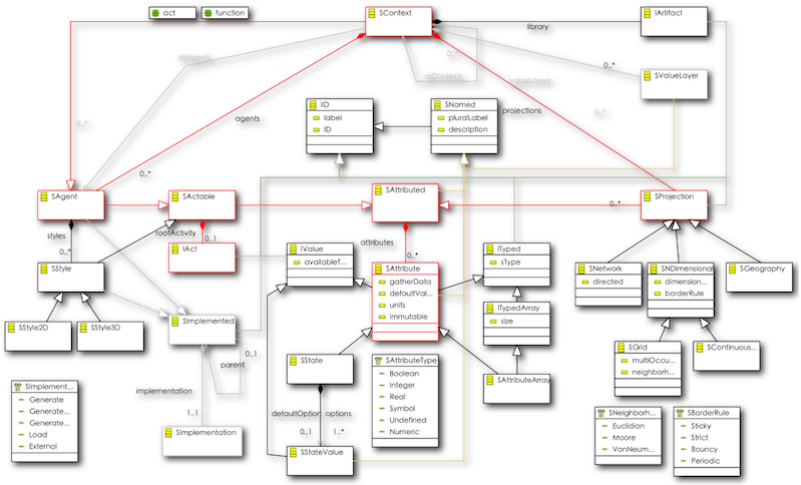


Figure 2: Example of a complex UML diagram [9]

3.3 Mind Maps

Tony Buzan is reagrded as the inventor of the Mind Map and the methodology it employs on expanding ideas. The following is a direct extract from Buzan

and Buzan(2006[13]).

> *" The mind map is an expression of radiant thinking and is therefore a function of the human mind. It is a powerful graphic technique which provides a universal key to unlocking the potential of the brain. The mind map can be applied to every aspect of life where improved learning and clearer thinking will enhance human performance. The mind map has four essential characteristics:*
>
> - *The subject of attention is crystallized in a central image.*
> - *The main themes of the subject radiate from the central image as branches.*
> - *Branches comprise a key image or key word printed on an associated line. Topics of lesser importance are also represented as branches attached to higher level branches.*
> - *The branches form a connected nodal structure.*
>
> *"*

> *" Your brains thinking pattern may thus be seen as a gigantic, branching association machine (BAM) a super bio computer with lines of thought radiating from a virtually infinite number of data nodes. This structure reflects the neuronal networks that make up the physical architecture of your brain. From this gigantic information processing ability and learning capability derives the concept of radiant thinking of which the mind map is a manifestationa mind map, which is the external expression of radiant thinking always radiates from a central image. Every word and image becomes in itself a subcentre of association, the whole proceeding in a potentially infinite chain of branching patterns away from or towards the common centre. Although the mind map is drawn on a twodimensional page it represents a multidimensional reality, encompassing space, time and colour. "*

A mind map is a logical way of deriving concepts and a hierarchical structure from content. A mind map can be utilised in a generic way where a concept is not comitted to a type, but just a logical grouping of information. Mind maps suffer from similar problems to UML diagrams though, they do not scale well for the vailable screen real estate.

Figure 3 and 4 shows 2 different examples and layouts of mind maps.

3.4 Hierarchical Edge Bundling

Holten (2006[14]) describes a visualisation technique for compound graphs (graphs with hierarchical and other relations). A standard visualisation for tree data is used on the outside while b-splines are used to neaten up the remaining adjacency relations between nodes. The visualisation has been applied to call graphs which is also presented during part of this project.

Figure 3: Mind Map Diagram Example [6]

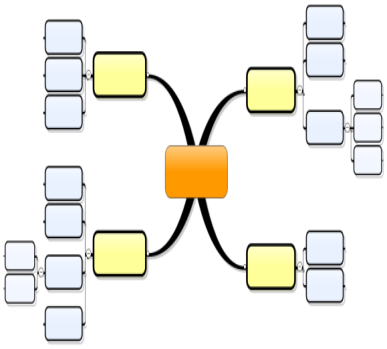


Figure 4: Mind Map Diagram Example [7]

3.5 ANTLR

The following description is directly extracted from the ANTLR website [12].

> *"ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees."*

3.6 Dr Garbage and ASM

Dr Garbage offers visualisations for java bytecode, sourcecode and control flows. The tool is presented as an Eclipse plugin and is open source [2].

The following description is directly extracted from the ASM website [1].

> *"ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. Provided common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools. ASM offer similar functionality as other bytecode frameworks, but it is focused on simplicity of use and performance. Because it was designed and implemented to be as small and as fast as possible, it makes it very attractive for using in dynamic systems."*

3.7 Requirements

Due to the research focused nature of this project, the final requirements and design has changed frequently during development. This has resulted in a project that shares some of the same fundemental ideas of the originaly designed project but is no longer in sync with the requirements and design documentation provided earlier. The following table gives details of requirements and aims that are more in line with the final outcome of the project.

Aim 1 Create a methodology that allows for intuitive expanding of a project

Aim 2 The methodology must delay assigning types to concepts as long as possible

Aim 3 Derive a method of tracing specific parts of the design to documentation

Aim 4 Construct a visualisation that scales well with little clutter while giving the user a good oversight of the system being designed

Requirement 1 Build a tool that supports a de-cluttered visualisation of the project

Requirement 2 Build a tool that can generate code from the designed project (only supports Java projects)

Requirement 3 Build a tool that can take an existing Java Project and revert it into a design that can be displayed in the tool again (only supports Java projects)

Requirement 4 The tool must have a web based component (to allow for future collaboration)

Requirement 5 The tool must be able to easily move concepts around to restructure the system quickly

4 Methodology

In the following section we derive a methodology to assist the user in designing a new system. The methodology is aimed at giving the user an intuative process of identifying key concepts and elaborating on them to first derive a conceptual understanding of the project before assigning specific types and roles to these concepts. The methodology also aims to address some of the following flaws of other design methodologies and tools:

Visual Clutter

Making effective use of the screen real estate available to the user.

Viewing Sub Systems

Allow the user to see different levels of abstraction for different concepts.

Delaying commiting to Types

Allow the user to expand the understanding of the system being designed before committing specific concepts to types.

Traceability

Allow the user to trace a concept back to documentation to justify it being created.

4.1 Describing a Project

The inception of any idea or system starts with a description of some of it's characteristics and expected functionality. The intermediate steps between the inception and development depends on the methodology being employed by the company, team or individual designing the system.

Often one of these steps would be to create design specifications, requirements documents and other forms of documentation that at least describes the system in some detail (even it is not of a technical nature).

Here we derive our first step in the methodology, `Describing a Project`. Understanding what is expected from the system and how to best implement it requires some form of guidance describing its properties and behaviour. To achieve this, as many `Documents` should be acquired as possible, describing any such properties and behaviour. The documents can be of any nature (technical or natural language) from which individual `Statements` can be derived detailing some part of the system.

After the `Documents` have been acquired, individual `Statements` are then extracted from these `Documents` for further analysis. A `Statement` is usually a sentence (or part thereof) from the `Document` describing some `concept(s)` and `relation(s)` of the system. From these `Statements`, a conceptual view can be

built up to better understand the system.

4.2 Understanding and expanding a Project

After the initial extraction of `Statements`, the `Statements` can be analysed individually to add information to our current conceptual understanding of the system. The designer makes use of the principles of "Mind mapping" to help group together related `concepts`. `Concepts`, even though generic, will primarily have 2 different kinds of `relations`.

Hierarchical Relation

The relation implies that there is a logical structure or grouping from the parent to the child.

General Relation

The relation implies that there is a connection between the concepts, but that they are not part of each other in terms of hierarchy. This relation will mostly take the form of exchanging of information or similar types of interaction.

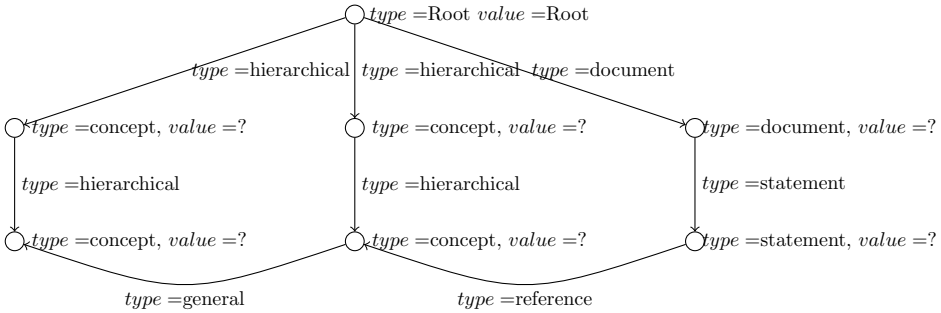
To understand why certain `concepts` and `relations` came into existence in the system design, we would like to keep track of the `Statement`(source) from which they were derived.

To store all this data we have to describe a data structure to represent the system. We describe the `Project Graph` as $G = \{V, E\}$ with a Root node to help with graph traversal later. This Root node may never be removed from the graph. Furthermore we already know of the following information that has to be stored in the `Project Graph`:

- Documents
- Statements
- Concepts
- Relations (hierarchical and general)
- References from concepts to statements

We assign the following attributes to vertices and edges, *type* and *value* so that we may have the following to describe the current information.

The current structure of a `Project Graph` would be presented as follows:



4.3 Refining the design of the system

If contradictions or ambiguous concepts arise from the Mind Map, trace it back to the documentation and seek clarity about the situation from stakeholders, before updating the required document(s) and continuing.

To understand the role and interactions of a newly added `concept`, the user is encouraged to ask himself the following question help improve the design of the system.

When is this concept being called?

During which phase, or on what kind of action should this `concept` be called or interacted with? These callee's are also `concepts` on their own rights and should be added to the `Project Graph`. This is similar to the listener pattern as the current concept would be listening for certain events to happen or information to be shared.

How (if applicable) does the concept work in steps?

To understand how this concept works, it will most likely be doing a set of operations on data, these steps and gathering of data are `concepts` on their own and should also be added to the `Project Graph`.

Are the concepts related?

When adding `concepts` under the same parent it is important to be able to easily distinguish between child `concepts` of similar nature or abstraction. Thus every time a new concept is added to the `Project Graph` it is important to see if the `concepts` can be grouped together. Algorithm 1 serves as a guide when placing new `concepts` in the `Project Graph`.

4.4 Assigning Types

After the `Project Graph` has been expanded and refined to a satisfactory level, the user would expect to start coding the structure derived. The methodology

Algorithm 1 Adding a Concept to the Graph

```
1:  procedure AddConceptToGraph
2:      concept
3:      parent
4:      graph
5:  top:
6:      if parent == null then
7:          parent ← graph.root
8:  loop:
9:      for child of parent do
10:         if child is logical abstraction or grouping of concept then
11:             parent ← child
12:             goto loop.
13:     if child differs from parent.children then
14:         create groupings
15:         goto loop.
16:     parent.children add concept
```

and tool in this iteration only accomadtes Java projects and it's structure. To generate scaffolding to guide the developer, types need to be assigned to some `concepts` and `relations`. Section 5.2 describes the types that can be assigned to construct a valid Java hierarchy to generate skeleton code from.

4.5 Reverse Engineering

After an initial itteration of design and development the user would surely expect to read back new information discovered during development back into the visualisation. This is an area other tools often fall short in as it is a cumbersome process. Having the current state of the system reflect in the design helps identify areas that can be improved, abstracted out or to help visualise the flow of information for other team members. A detailed description of how an existing project gets read back into the design can be found in Section 5.6.

4.6 Summary

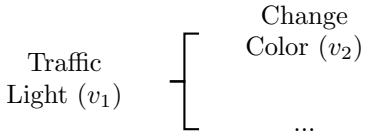
The summary serves as a quick guide to the methodology. Notations are also introduced here that will be used throughout the rest of the document and appendices (examples).

Given a directed graph $G = \{V_G, E_G\}$ and directed tree $T = \{V_T, E_T\}$ such that $T \subseteq G$.

Hierarchical Relation and Notation

$v_1 \rightarrow v_2$: Follow the edge from v_1 to v_2. If v_2 does not exist, add v_2 to V_G and V_T. Add a directed edge e, with source v_1 and destination v_2 to E_G with $e_{type} =$ `hierarchical relation`.

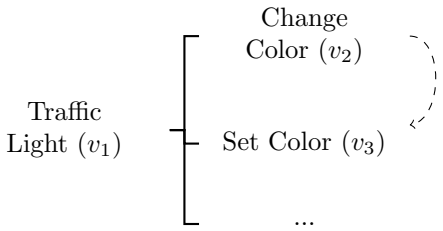
Example: Traffic Light (v_1) $\rightarrow$ Change Color (v_2)



General Relation and Notation

$v_1 \rightarrow ... \rightarrow v_i \implies u_1 \rightarrow ... \rightarrow u_i$: Follow the edge(s) from v_1 to v_i and from u_1 to u_i as described by the `Hierarchical Relation`. From v_i to u_i create a directed edge e with $e_{type} =$ `general relational` and add it to E_G.

Example: Traffic Light (v_1) $\rightarrow$ Change Color (v_2) $\implies$ Traffic Light (v_1) $\rightarrow$ Set Color (v_3)



Steps

- Gather all available Natural Language documents (or create some with a rudimentary description of what the intended system should be able to do).

- From these Natural Language documents, extract `statements`. A `statement` is usually a single line of text from the document, but can be any piece of arbitrary text. These `statements` will help guide the design.
- From a given `statement`, extract `concepts` and their `hierarchical relation`[1] to each other.
- From these `concepts`, identify `general relations`[2]
- Lastly, from the `statement` identify `constraints` and add them as `concepts` as well.
- After all `concepts` have been added and refinements applied, assign more specific `types` to the `concepts` and `relations`.

Refinement Process

Whenever a new `concept` or `relation` is added to G, follow the following refinement steps:

Ask when? When should this `concept` be evaluated? There is an implicit `general relation` from the answer to the current `concept`. Asking this allows us to understand where and when this concept should be reached in the project being designing.

Ask how? How should this `concept` gather information to execute the expected functionality or to which other `concepts` should some of the work be delegated? There is an implicit `general relation` from the `concept` to the answers of this question. Asking this allows us to understand communication and refinements that will be needed from other parts of the project being designed.

Is it related? The design of a project requires logical groupings and abstractions of similar `concepts`. When a `concept` c_{new} is added with a parent c_{parent}, it is required to evaluate the `concept` in terms of its siblings (other children of c_{parent}). If there is a clear separation in the children of c_{parent}, create 2 new `concept` c_1 and c_2 to use as groupings, adding each `concept` c_i from the children of c_{parent} under the appropriate concept c_1 or c_2. Finally add c_1 and c_2 as children of c_{parent}

[1] Hierarchical relations can be viewed as logical groupings, abstractions etc.
[2] General relations can be seen as interactions, types etc.

5 Design

5.1 Visualisation

During the previous sections we have identified the underlying structure as a graph with 2 initial types of edges (`relations`). We thus need to construct a visualisation that clearly highlights these differences.

Holten (2006 [14]) has done extensive research into visualisation of compound graphs to minimise the amount of clutter created by the edges in them. Figure 5 and 6 shows 2 versions of the same graph. On the outside is a tree structure that represents the hierarchical part of the graph while the edges within the inner circle shows added relations between them The edges go from green(source) to red(target). Figure 5 shows the labels with a tension of 0 while Figure 6 shows them with a tension closer to 1. The tensions determines the path that is created by the b-spline by following the placement of the hierarchical structure of which the edge forms a part.

Figure 6 gives a good overall indication of how certain parts of the system interacts with each other as the visualisation displays calls among the classes of a system. The work done by Holten (2006 [14]) has mainly been focused on call graphs, while this kind of visualisation will form the basis of the tool with added functionality to control the outside hierarchy and information being displayed by the edges as well as using the visualisation for the modelling of the conceptual overview.

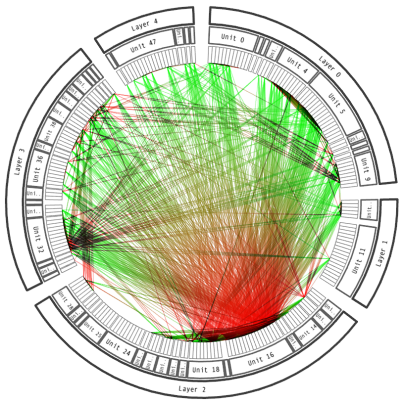


Figure 5: Hierarchical Edge Bundling [14] with 0 tension.

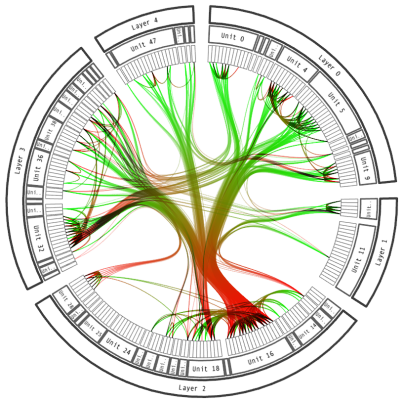


Figure 6: Hierarchical Edge Bundling [14] with close to 1 tension.

5.2 Assigning Types

At some point during the design the user will have a thorough understanding of a specific concept and will want to commit it to a specific type for later use. The following types will be available for this iteration of the tool to support code generation and reverse engineering of Java projects.

Expected Types:

Document A collection of statements.

Statement A statement with a piece of content explaining a piece of the system.

Java File A .java file containing 1 or more class and/or interface.

Java Class A single Java class consisting of variables and methods. When the Java Class is added a `Gen Header` and `Gen Footer` is automatically added to that `concept`.

Java Method Group An abstraction of a group of Java Methods that share the same name but different method signatures. When the Java Method Group is added a `Gen Header Node` is automatically added to that `concept`.

Java Method A specific implementation of a method in a Java Class

Java Variable A variables specified in a Java Class

Java Doc Description of the Class, Interface or Method.

Package A Java Package

Unknown The `Node` is not used for a specific part of a Java structure but rather for house keeping.

Gen Header Code to be inserted during Code Generation phase at the start of the `concept`.

Gen Footer Code to be inserted during Code Generation phase at the end of the `concept`.

Concept A general `concept` in the `Project Graph`.

Java Method Call A method call from one `concept` to another `concept` labelled as a Java Method.

Expected Call An expected call from one `concept` to another `concept`.

Hierarchical Relation A `relation` between two `concepts` of a hierarchical nature.

General Relation A `relation` between two `concepts` of a general nature.

Reference A reference from a `statement` to a `concept`.

5.3 Hierarchy Extraction

After the initial conceptual view of the system has been constructed, the designer can now start assigning specific types to the `concepts` and `relations` in the `Project Graph`.

This however creates a problem during the visualisation. Previously the hierarchical part of the visualisation could easily be extracted from the graph by following the `Root` node and building a tree following edges of type `hierarchical relation`. The types of the nodes and edges have now been changed and it is no longer clear which were `hierarchical` and which were `general`.

With an underlying directed graph structure, many different hierarchies can be constructed to visualise the system from different view points. For this we must specify a language that can traverse the graph and extract a tree from it. The basis idea of the language is to start at the Root node and gather `concepts` and `relations` until a spanning tree is formed. This would then be used as the outside hierarchy while the remaining edges would be visualised by the edge bundles.

Suggested Language specification:

```
step ( node | edge ) ( type | value) parameter+ recursive?
```

The language specification allows us to traverse the graph by nodes, edges, types and values. The recursive part at the end allows for the same rule to be applied until no more new edges have been gather before moving on to the next step. Algorithm 2 gives a basic overview of how a rule may be applied to gather the `concepts` and their `relations`.

Example for rule set to extract the Java hierarchy

```
1.  node value Root
2.  edge type ( Hierarchical Relation | General Relation ) recursive
3.  node type Package recursive
4.  edge type ( Hierarchical Relation | General Relation ) recursive
5.  node type ( Java File )
6.  edge type ( Hierarchical Relation | General Relation ) recursive
7.  node type ( Java Class | Java Interface )
8.  edge type ( Hierarchical Relation | General Relation ) recursive
9.  node type ( Java Method Group | Java Property )
10. edge type ( Hierarchical Relation | General Relation ) recursive
11. node type ( Java Method )
12. node type ( * )
```

5.3.1 Diamond Structures

During the traversal of the graph, a situation might arise where there is no clear solution to which parent a node should be assigned to in the existing hierarchy.

Algorithm 2 Graph Traversal

```
1: procedure BuildTree
2:     graph
3:     set
4:     rule
5: top:
6:     tmpSet ← empty
7: loop:
8:     for v in set do
9:         if rule.node then
10:            for edge connected to v do
11:                if rule.type then
12:                    if v_other_type == rule.parameter then
13:                        tmpSet.add(v_other)
14:                else if rule.value then
15:                    if v_other_value == rule.parameter then
16:                        tmpSet.add(v_other)
17:        else if rule.edge then
18:            for edge connected to v do
19:                if rule.type then
20:                    if edge_type == rule.parameter then
21:                        tmpSet.add(v_other)
22:                else if rule.value then
23:                    if edge_value == rule.parameter then
24:                        tmpSet.add(v_other)
25:    set ← tmpSet + set
26:    if rule.recursive then
27:        set ← BuildTree(graph, set, rule)
28:    return BuildTree(graph, set, rule + 1)
```

Figure 7 displays this situation and is in the form of a diamond structure. There are 2 suggested ways to resolve the situation but further solutions may be needed to get a better end result during the visualisation.

- The first path in the tree traversal to reach a specific node (Figure 8)
- Duplication (Figure 9)

For the current tool the first path in the tree traversal to a specific node is used as the hierarchy.

5.4 User Interface

Apart from the previously stated visualisations, the following is needed for the user to be able to interact with the tool.

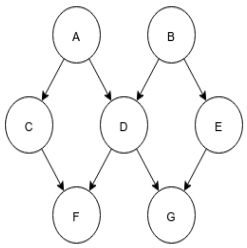


Figure 7: Diamond Structure

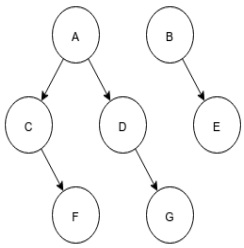


Figure 8: Hierarchy on First come basis

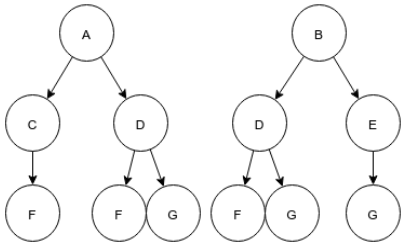


Figure 9: Hierarchy with duplication

5.4.1 Pin Layout

To easily organise the information and controls that the user will have available, a `pin layout` is created where each piece of information is created in it's own container that can be moves around. Three main pinning areas are then created and each `pin containers` can be pinned to any of these areas.

The areas are represented in Figure 10.

Figure 10: `Pin Layout` with it's 3 sections to pin to

5.4.2 Tree

To control the hierarchy on the outside part of the visualisation, a tree visualisation is constructed to allow the user to expand and collapse different `concepts` in the tree as well as reordering them. Figure 11 shows the tree while Figure 12 shows a `concept` being dragged to be grouped under a different `concept`.

5.4.3 Documents And Statements

`Documents` and `Statements` form a crucial part of the derivation of the system and are thus displayed on their own as `pin containers` to be pinned and used as seen fit. From each of these containers the user is able to add a `concept` directly to the `Project Graph` with an implicit `reference relation` added as well. Figure 13 shows a `pin container` for a `Document` listing all the statements while Figure 14 shows a `pin container` for a single `statement` to add a `concept` from.

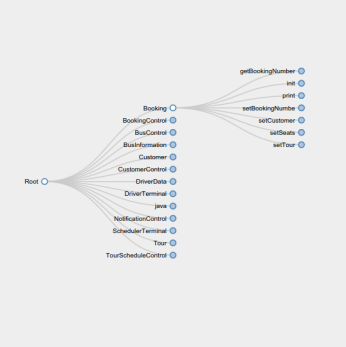


Figure 11: Tree Layout

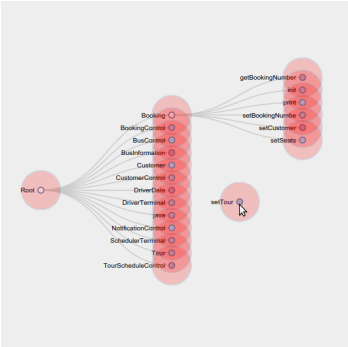


Figure 12: Tree Layout with `concept` being moved

Figure 13: Tree Layout

Figure 14: Tree Layout with `concept` being moved

5.5 Code Generation

By the time code needs to be generated, the structure would have already been derived by the user. During type assignment, extra `concepts` namely `Gen Header` and `Gen Footer` has been added to the graph each time a type is set for `Java Class`, `Java Method` or `Java Property`. These `concepts` contains the exact code to be inserted into the file.

Code generation traverses the graph as follows:

Root The start of the traversal

Package For each package, a folder is created

Java File A java file is created with the specified name.

Java Class The `Gen Header` is directly inserted into the file with the `Gen Footer` added after the rest of the code is generated.

Java Property The `Gen Header` is directly inserted into the file with the `Gen Footer` added after the rest of the code is generated.

Java Method The `Gen Header` is directly inserted into the file with the `Gen Footer` added after the rest of the code is generated.

5.6 Reverse Engineering existing Java Projects

After the basic structure was code generated and some code implemented, it is expected that the modifications can be read back into the design as this is an iterative process during development. To gather all the related data we employ 2 different analysis on the current Java project to extract all the information we will need.

- Java Byte Code Analysis
- Java Source Code Analysis

5.6.1 Byte Code Analysis

To perform `Java Byte Code Analysis` on an existing project an external tool is used to analyse the `class` files. For this we use Dr. Garbage and the ASM library that it utilises. The directory where the Java project is located is traversed and every file marked as a `class` file is explored. From here instructions are extracted to build a call graph between different components of the system. This is converted into a `Project Graph` to be merged with the `Project Graph` created during the `Source Code Analysis`.

5.6.2 Source Code Analysis

To perform `Java Source Code Analysis` on an existing project an external tool is used to analyse the `java` files. For this we use ANTLR.

ANTLR combined with a predefined file that specifies the Java Language allows us to traverse the source to extract specific parts we want to use again for code generation later. After all, all this effort has been done to code the program in a specific way and allows us to keep to higher levels of abstraction without having to know finer details within a method for example as the calls have been identifier during the `Byte Code Analysis`.

The Java Language exists out of a set of rules to determine if the current syntax is correct and can be compiled. It is inherently broken up into pieces that are combined to form more complex structures. By using a grammar that has already been created for the Java Language we can now continue by working with these combined structures called rules. ANTLR provides functionality to access the content of the individual parts of these rules as well as functionality to handle when the rule gets entered and when it exists again.

```
genericMethodDeclaration
@init{xmlRuleStart(_localctx.getRuleIndex());}
@after{xmlRuleEnd(_localctx, $text);}
    :   typeParameters methodDeclaration
    ;
```

In this example the rule being discussed is `genericMethodDeclaration` that is compromised from `typeParameters` and `methodDeclaration`. Just after specifying the name 2 controls are specified namely `init` and `after` that will execute our helper methods `xmlRuleStart` and `xnkRuleEnd`. The variable `text` is passed on to `xmlRuleEnd` to allow us to capture all the text that is contained in this rule, including tokens on the hidden stream. This allows us to keep the text formatting in stead of trying to recompile it ourselves.

Following this structure creates a lot of excess information in the xml, but with with some helper functions the xml is tripped down to only the rules we are interested in, capturing their text and creating a new graph.

6 Implementation

The final implementation of the tool was broken up into 3 separate parts namely the `Browser Application`, `Code Generator` and `Project Explorer`. They all shared a `Graph Handler` library that helps with manipulating a `Project Graph`.

6.1 Graph Handler

The `Graph Handler` is a set of Java classes that gives the user access to importing, exporting and manipulating `Project Graphs`.

6.1.1 DataReader

For the current iteration of the tool, it is expected that all data to be read in as xml format. This might not always be the case in the future as we would like to have many different wasy in accessing data. An interface is thus created to specify methods that are expected to be implemented to change the data from it's current format into the format supported by the `Graph Handler`.

The following methods are expected to be implemented from any class implementing the `Data Reader` interface:

readFromFile

The method takes a string as a parameter, specifying the location of the file to be read.

getAllConcepts

The method does not take any arguments and returns a collection of `Concepts`.

getAllInteractions

The method does not take any arguments and returns a collection of `Relations`.

buildProjectGraph The method does not take any arguments and returns a single instance of a `Project Graph`. It is expected that the `Project Graph` will be built up using the previously stated methods.

6.1.2 DataWriter

For the current iteration of the tool, it is expected that all data to be written back into xml format. This might not always be the case in the future as we would like to have many different ways of exporting the data. An interface is thus created to specify methods that are expected to be implented to change the data from a format supported by the `Graph Handler` into a specified format.

The following methods are expected to be implemented from any class implementing the `Data Writer` interface.

writeToFile

The method takes a string as a paremeter, specifying the location of the file to be written to.

writeConcepts

The method takes a collection of `Concepts` and writes them to the specified file.

writeRelations

The method takes a collection of `Relations` and writes them to the specified file.

6.1.3 Project Graph

The Project Graph class gives the user access to manipulating a specific `Project Graph`. The user has access to the following methods to do so.

createGraph

addConcept

removeConcept

addRelation

removeRelation

moveConcept

moveConcepts

mergeProjectGraph

6.2 Project Explorer

The `Project Explorer` is a Java Project that analyses other existing Java Projects and transforms them into a `Project Graph` via the `Graph Handler`. It exists of 2 main parts namely the `ByteCodeAnalyser` and the `SourceCodeAnalyser`. Each of these creates a separate `Project Graph` and the 2 are then merged together afterwards.

6.2.1 ByteCodeAnalyser

The `ByteCodeAnalyser` relies on Dr Garbage[2] and the ASM library that it makes use of to analyse existing class files for their interactions. This allows us to easily identify method calls and class inheritance without having to do extra parsing on the source code.

exploreDirectory

The method takes a string as a parameter as the starting directory from which to start exploring the Java Project. This would normally be the bin folder of the project, as the package structure can be derived from the class name using the supplied ASM tools.

exploreClass

The method takes a string as a parameter as the file name of a class to be explored. A `ClassReader` from ASM is initialised to start exploring the class. From the class name the package hierarchy is used to add `Concepts` to the `Project Graph`. From the `ClassReader` a `ClassNode` is obtained and used to get a collection of `MethodNodes`. The `MethodNodes` are then passed on to `exploreMethods`.

exploreMethods

For each `MethodNode` specified in the `MethodNodes`, the `MethodNode` is passed on to `exploreMethod`.

exploreMethod For each `MethodNode` a `Concept` of type `Java Method Group` is created before a `Concept` of type `Java Method` is added as well with the specific signature of the method. A collection of `AbstractInsnNodes` is then obtained and checked for any of type `MethodInsnNode`. The `MetodInsnNode` contains details of the package, class and specific method. A similar process as the start is used to add these details as concepts to the `Project Graph`. Finally a `Relation` of type `Java Method Call` is added for between these methods.

getProjectGraph

The method returns the `Project Graph` that has been built up by the `ByteCodeAnalyser`

6.2.2 SourceCodeAnalyser

exploreDirectory

The method takes a string as a parameter as the starting directory from which to start exploring the Java Project. THis would normally be the src folder of the project. The method finds and keeps the paths of all the `.java` files it finds in the subdirectories.

6.3 Browser Application

The following section describes the layout and workings of the `Browser` Application on which the tool is mainly implemented. The front end consists of HTML pages with javascript libraries doing the visualisations and communication with

the back end. The back end is implemented on the `Django`[11] web framework while `D3.js`[4] is used for the visualisations. Figure 15 shows the conceptual layout of the structure.

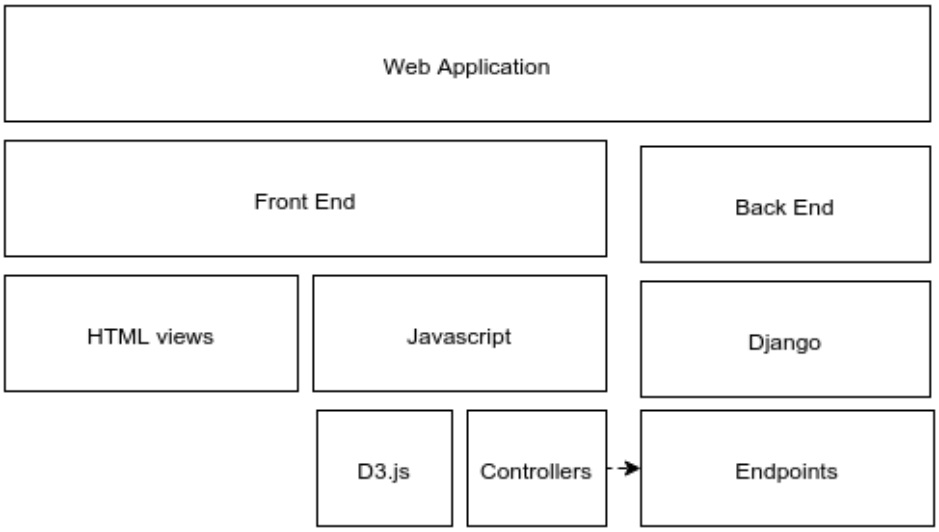


Figure 15: Structure of the Browser Application

6.3.1 Pages

The `Web Application` only consists of 2 pages, namely:

Main

On the main page the user may specify an xml file to be parsed into a `Project Graph` along with a file containing the rules to build the hierarchical structure. After uploading these files the user is redirected to the `Canvas` page.

Canvas

6.3.2 Views

On the `Canvas` page, the following views can be found:

Inverse Sunburst

The `Inverse Sunburst` (Figure 16) represents the hierarchical structure extracted from the `Project Graph`. The view is able to highlight a node and it's parents along the way to the root to highlight where it fits in the structure while greying out the other nodes (Figure 17). Specific functionality is described in Section 6.4.1.

Edge Bundling

The `Edge Bundling` (Figure 16) shows the relations between nodes that are not of a hierarchical nature. When a specific node is highlighted(from the `Inverse SUnburst`) it only shows relations that are related to that highlighted node. The edges are then slightly offset from the centre and coloured green(source of the `relation`) or red(target of the `relation`) to clearly show it's interaction with the rest of the system. Specific functionality is described in Section 6.4.2.

Hierarchical Tree

The `Hierarchical Tree` (Figure 16) represents the hierarchical structure from the `Inverse Sunburst`. It is used to expand and collapse specific `concepts` so that different layers of abstraction may be viewed while also creating an easy way to move an entire grouping to a different parent `concept`. Specific functionality is described in Section 6.4.3.

Figure 16: Inverse Sunburst, Hierarchical Edge Bundling and Tree all together

6.4 Controllers

The following section describes controllers and their individual role in visualising data. Figure 18 shows the overview of how they are expected to interact.

6.4.1 InverseSunburst.js

The `InverseSunburst.js` file is responsible for the functionality related to the `Inverse Sunburst` view(Section 6.3.2). The drawing is handled by the D3.js library with an implementation from [8] used as the basis with some modification to enable the hierarchical structure to be inverted.

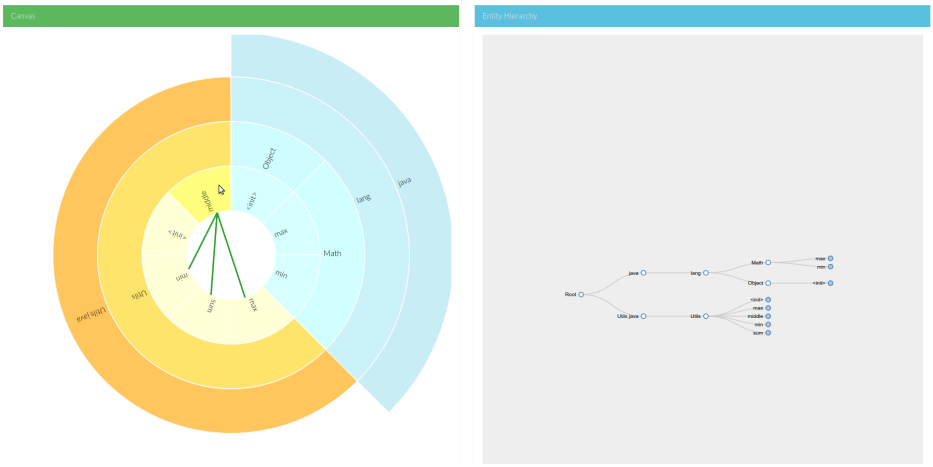


Figure 17: Inverse Sunburst, Hierarchical Edge Bundling and Tree all together with a specific `concept` highlighted

The following methods have been implemented:

setConcepts A list of `concepts` are set using a json format. After being set the `drawAll` method is called to draw them in an appropriate manner.

drawAll

hoverConcept

When a cursor is hovered over a `concept`, all other concepts not on the part from the current `concept` to the root `concept` is greyed out. This functionality was already provided by the base implementation. Furthermore the method `hoverConcept` in the `GraphHandler.js` file is notified so that other changes may also take place.

leaveConcept

When a cursor leaves a specific concept the visualisation must return back to normal. The `drawAll` method is called to draw all `concepts` in their original state.

6.4.2 EdgeBundling.js

The `EdgeBundling.js` file is responsible for the functionality related to the `Edge Bundling` view(Section 6.3.2). The drawing is handled by the D3.js library with an implementation from [5] used as the basis with some modification to control the b-splines a bit more specifically and to change the edges being drawn.

The following methods have been implemented:

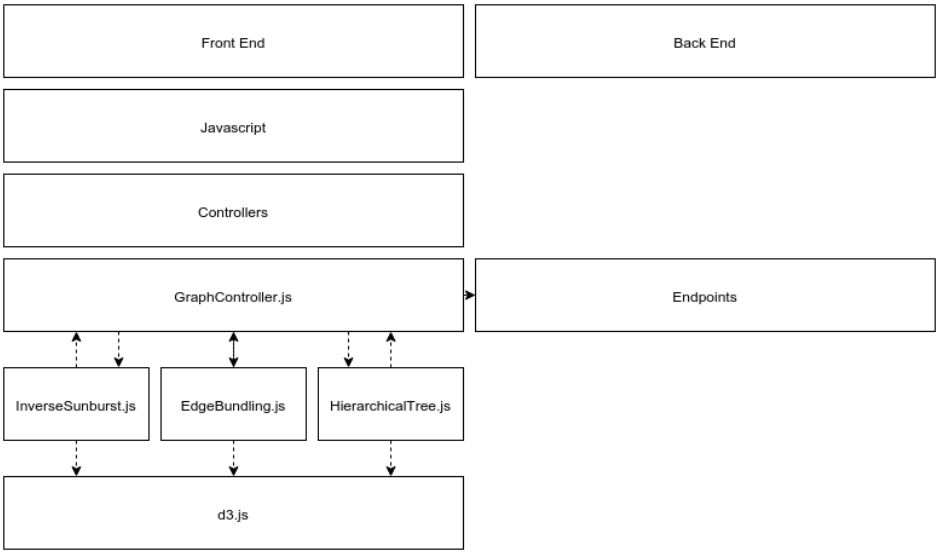


Figure 18: Controller Interactions

setRelations

Sets the `relations` to be displayed.

drawAllRelations

Draws all the relations that have been set by the `setRelations` method.

6.4.3 HierarchicalTree.js

The `HierarchicalTree.js` file is responsible for the functionality related to the `Hierarchical Tree` view(Section 6.3.2). The drawing is handled by the D3.js library with an implementation from [3] used as the basis with some modification to handle movements of `concepts` as well as expanding and collapsing the tree.

The following methods have been implemented

setConcepts

Sets the concepts to be drawn.

drawConcepts

Draws the `concepts` that were set during the `setConcepts` method.

moveConcept

Attaches a `concept` to a new parent `concept`. This causes the `drawConcepts` to be called again as well as notifying the `Graph Controller` to notify of the change.

expandConcept

Expands a `concept` in the hierarchy to reveal the children `concepts`. This method also notifies the `Graph Controller` to notify of a change in the structure to be displayed.

collapseConcept

Collapses a `concept` in the hierarchy to group it's children `concepts` together. This method also notifies the `Graph Controller` to notify of a change in the structure to be displayed.

6.4.4 GraphController.js

The `GraphController.js` file acts as an intermediary between the `Endpoints` and the visualisations at the front. Since all the visualisations make use of the same data it makes sense to have a central controller to handle interaction with the `Project Graph` in the back end.

The following methods were implemented

getConcepts

Creates an asynchronous http call to the back end through the `Get Concepts` endpoint to get all the `concepts` of the current system.

getRelations

Creates an asynchronous http call to the back end through the `Get Relations` endpoint to get all the `relations` of the current system.

moveConcept Creates an asynchronous http call to the back end through the `Move Concept` endpoint to update a `concept` being moved to a new parent `concept`. Afterwards the `getConcepts` and `getRelations` methods are called again to update the visualisations.

changeVisibleConcepts

This method is mostly called by either the `HierarchicalTree.js`, `InverseSunburst.js` or `EdgeBundling.js` controllers to notify of a change in the current concepts that are visible. This method updates the `concepts` and `relations` in all 3 of these controllers to display a new visualisation.

addConcept

Creates an asynchronous http call to the back end through the `Add Concept` endpoint to add a new `concept`. Afterwards the `getConcepts` and `getRelations` methods are called again to update the visualisations.

removeConcept

Creates an asynchronous http call to the back end through the `Remove Concept` endpoint to remove an existing `concept`. Afterwards the `getConcepts` and `getRelations` methods are called again to update the visualisations.

addRelation

Creates an asynchronous http call to the back end through the `Add Relation` endpoint to add a new `relation`. Afterwards the `getConcepts` and `getRelations` methods are called again to update the visualisations.

removeRelation

Creates an asynchronous http call to the back end through the `Remove Relation` endpoint to remove an existing `relation`. Afterwards the `getConcepts` and `getRelations` methods are called again to update the visualisations.

setConceptType

Creates an asynchronous http call to the back end through the `Set Relation Type` endpoint to change a `concept` type.

setCocneptValue

Creates an asynchronous http call to the back end through the `Set Relation Type` endpoint to change a `concept` value.

setRelationType

Creates an asynchronous http call to the back end through the `Set Relation Type` endpoint to change a `relation` type.

setRelationValue

Creates an asynchronous http call to the back end through the `Set Relation Type` endpoint to change a `relation` value.

6.4.5 Endpoints

The following endpoints are created to serve as a way for the front end of the browser application to communicate with the back end to manipulate the `Project Graph` of the current system being designed. During this time it is assumed that the user went through the `Home View` which would have created the related `Project Graph` and associated it with the user's current session.

Get Concepts `/getConcepts/`

Returns all the `concepts` from the current `Project Graph`.

Get Relations `/getRelations/`

Returns all the `relations` from the current `Project Graph`.

Add Concept `/addConcept/`

Adds a new `concept` to the `Project Graph`. The method requires a value to be passed along for the concept as well as either the `id` of the parent concept or a string representing the path from the `Root` concept to the parent.

Move Concept `/moveConcept/`

Moves a `concept` to a new parent `concept`. The method either requires the ids of the parent and child `concepts` or 2 strings representing the path from the `Root concept` to the `concept` to be moved and the `concept` that will be the new parent.

Add Relation `/addRelation/`

Adds a new `relation` to the `Project Graph`. The method wither requires the ids of the source and target `concept` or 2 strings representing the path from the `Root concept` to the 2 specified `concepts`.

Set Concept Value `/setConceptValue/`

Sets the value of a `concept`. The method requires the id of the `concept` or the path from the `Root concept` to the required `concept`.

Set Concept Type `/setConceptType/`

Sets the type of a `concept`. The method requires the id of the `concept` or the path from the `Root concept` to the required `concept`.

Set Relation Value `/setRelationValue/`

Sets the value of a `relation`. The method requires the id of the `relation`.

Set Relation Type `/setRelationType/`

Sets the type of a `relation`. The method requires the id of the `relation`.

Download Data `/downloadData/`

The method takes the current `Concept Graph` and exports it as an XML document to be used again later.

6.5 Code Generation

`Code Generation` was implemented as a Java project. To run the code generation the Main class takes 2 command line arguments, namely the xml file containing the information to construct a `Project Graph` and a location to where the code must be generated to. The `Code Generation` project then follows the process listed in Section 5.5.

6.6 Reverse Engineering

`Reverse Engineering` of an existing Java project was implemented as a Java project. To extract a `Project Graph` from an existing Java project the Main class takes 2 command line arguments. The directory which has to be explored as well as the file to output the information to. The `Reverse Engineering` project then follows the process listed in Section 5.6.

6.7 Graph Merger

The `Graph Merger` is a standalone project but is implemented by the `Reverse Engineering` project as well. The Main class takes 3 arguments, namely the locations of 2 xml files to be merged together as well as the location of a 3rd file where the combined graph will be exported to.

The merger works by specifying a primary `Project Graph` into which the secondary `Project Graph` will be merged. The secondary starts by matching it's `Root` to the `Root` of the primary. Each child `concept` of the `Root` of the secondary is then matched to a `concept` in the primary that has the smallest distance to the `Root`. These are then merged together. If no concept was found in the primary it is added directly as a child to the `Root`. This process is repeated with different `concepts` as root as the graphs are being explored and merged.

7 Evaluation and Testing

For evaluation and testing I used the tool to visualise some old Java projects to get an idea of how they worked. Although this is not quantifiable measure, quickly following the flow in the visualisation gave me a good understanding of how the systems work.

The tool was used next on a very small use case (Appendix A) with figures to visualise how the `Project Graph` changed along the way to better visualise what was happening to the user.

Next, the use case that was used during this years CS344 module for constructing UML diagrams was partially constructed with the new methodology and visualisation (Appendix C) to then compare with the tool being used by a student from the group with minimal time to learn the tool and methodology (Appendix D).

The final part of testing took a use case from a Formal Specifications paper and proceeded to apply the methodology on it to see what the structure would like in the end(Appendix B).

8 Conclusion

The initial experience and feedback of the tool looks very promising. The visualisation has been met with enthusiasm and has made understanding a new project a lot easier. The methodology has forced the user to think more critically about every concept added to the graph and it's larger role in the system while the conceptual nature of the graph has made it is to add in pieces without knowing where they fit in exactly at first. The larger picture then allowed the

refocusing and abstracting of certain concepts before committing to types and starting the code generation and development part of the cycle.

For a more in depth conclusion more user would be require to test the tool while also drawing comparisons to their experiences with other tools (example a large portion of the CS244 module group from which only 1 was sampled in the available time).

References

[1] About asm. http://asm.ow2.org/index.html. Accessed: 2015-09-30.

[2] About dr. garbage. http://www.drgarbage.com/about/. Accessed: 2015-09-30.

[3] D3.js drag and drop, zoomable, panning, collapsible tree with auto-sizing. http://bl.ocks.org/robschmuecker/7880033. Accessed: 2015-09-30.

[4] Data driven documents. http://d3js.org/. Accessed: 2015-09-30.

[5] Hierarchical edge bundling. http://bl.ocks.org/mbostock/1044242. Accessed: 2015-09-30.

[6] Mind map guidelines. https://upload.wikimedia.org/wikipedia/commons/thumb/6/66/MindMapGuidlines.svg/2000px-MindMapGuidlines.svg.png. Accessed: 2015-09-30.

[7] Mind map structure. http://cdn-media-1.lifehack.org/wp-content/files/2012/09/Mind-Map-Structure.png. Accessed: 2015-09-30.

[8] Sequences sunburst. http://bl.ocks.org/kerryrodden/7090426. Accessed: 2015-09-30.

[9] Structure complex diagram. https://eclipse.org/amp/documentation/contents/images/structure/StructureComplexDiagram.png. Accessed: 2015-09-30.

[10] The three basic approaches applied to software development methodology frameworks. https://en.wikipedia.org/wiki/Software_development_process#/media/File:Three_software_development_patterns_mashed_together.svg. Accessed: 2015-09-30.

[11] The web framework for perfectionists with deadlines. https://www.djangoproject.com/. Accessed: 2015-09-30.

[12] What is antlr? http://www.antlr.org/. Accessed: 2015-09-30.

[13] T. Buzan and B. Buzan. *The Mind Map Book*. Mind set. BBC Active, 2006.

[14] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):741–748, Sept 2006.

9 Appendices

Appendix : A

This appendix demonstrates how the `Project Graph` is changes throughout the process of adding `concepts`, `relations` and changing their types. Code is then generated, completed and read back into the design to be visualised.

For this appendix the following description will be used to design a system and go through the methodology.

```
Create a library with utility functions to determine the
following from 3 elements:  Max, min, middle.  The middle
function must make use of the following formula :  middle = sum -
max - min.
```

Analysing the Use Case and treating the textual description as an electronic `Document` we obtain the following `Statements` that describe how the system should be designed.

- Create a library with utility functions to determine the following from 3 elements:
  - max
  - min
  - middle
- The middle function must make use of the following formula
  - middle = sum - max - min

Making use of the `Panels`(Figure 3 and 4) to add `Documents` and `Statements` the expected changes in the `Project Graph` can be seen in Figures 1 and 2.

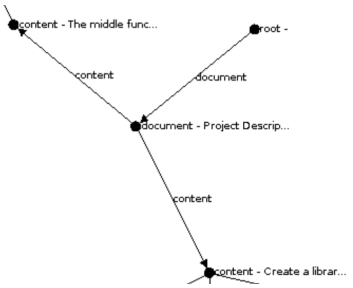


Figure 1: `Document` added to `root` of `Project Graph`

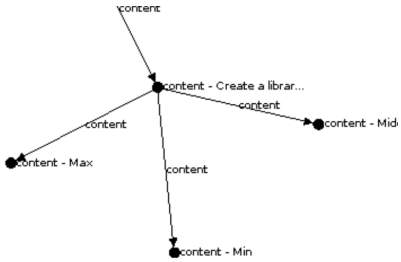


Figure 2: `Statement` added to a `Document` in the `Project Graph`

Figure 3: Adding `Statement` to `Statement`

Figure 4: Desired `Statement` added to the `Document`

Continuing, we now add some `concepts` to our `Project Graph`. First the `Pin Layout is used` to group all the pieces of `Statement` that will be used to create the new `concepts` in the `Project Graph`.

The following is identified as `concepts` that needs to be added based on the information available from the `Statement`.

Statement	`Concept` Value	`Concept` Type	Parent	Relation Type
Create a library...	Utils.java	Java File	root	Java File
Create a library...	Utils	Java Class	Utils.java	Java Class
max	max	Method Group	Utils	Method Group
min	min	Method Group	Utils	Method Group
middle	middle	Method Group	Utils	Method Group
middle = sum...	sum	Method Group	Utils	Method Group

Figure 7 shows how the `Project Graph` would have changed if the `concepts` were added normally while Figure 5 and 6 shows the `Statement Panels` being used to add the `concepts` with Figure 8 showing an extra reference `relation` between the `concepts`. Figure 9 show all the `Method Groups` and `References`.

Figure 5: Collection of `pinned Content Panels`

Figure 6: `Entity` being added from `Content Panel`

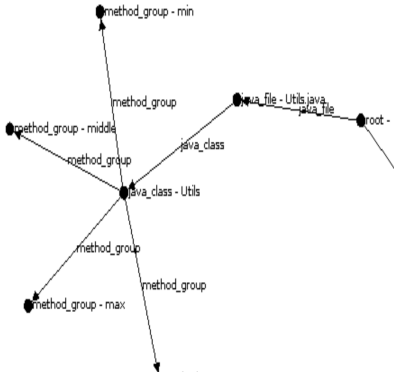


Figure 7: `Method Groups` in `Project Graph` with no `References`

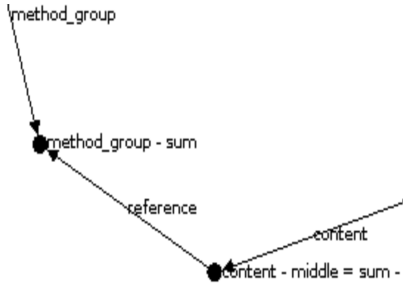


Figure 8: Reference between `Content` and `Method Group`

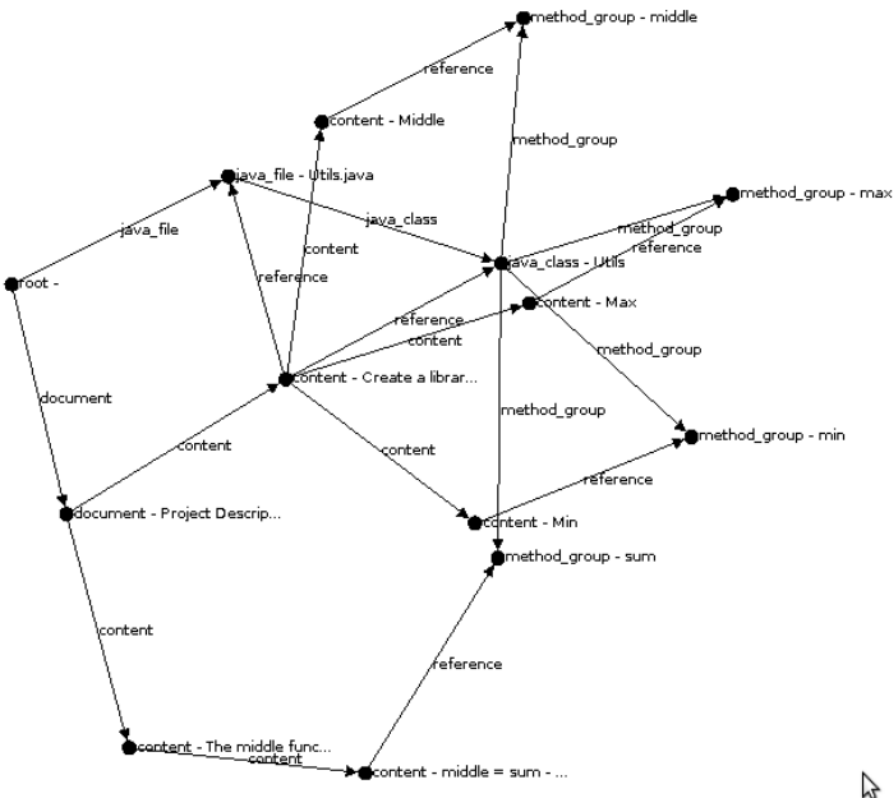


Figure 9: `Method Groups` with all the `References` in the `Project Graph`

We also now want to show during this layer of abstraction that we want to call `sum, min` and `max` from `middle` as stated in our `Statement`. Even though we do not know the exact `Method` we can assign an `Expected Call` between the `Method Groups` to inform us during code generation later that we want to call certain methods. During code generation these expected calls that have not been implemented yet will appear as TODO's in the Java File. Figure 10 shows a `Relationship` being added from a `Panel` and Figure 11 shows the `Edge` after it has been added to the `Project Graph`. Figure 12 shows all the `Expected Calls` in the `Project graph` with Figure 13 showing the complete `Project Graph` after the initial design phase (with added `Gen Headers` etc.).

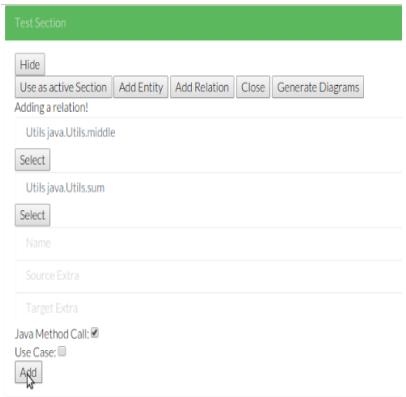


Figure 10: Adding a `Relationship` from a `Panel`

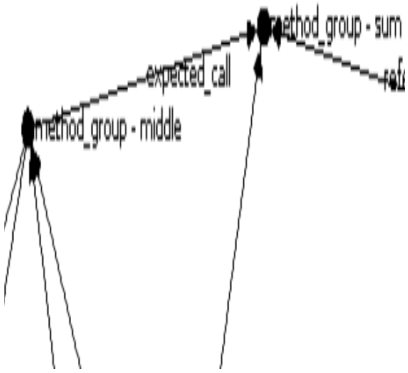


Figure 11: `Expected Call Relationship` between 2 `Method Groups`

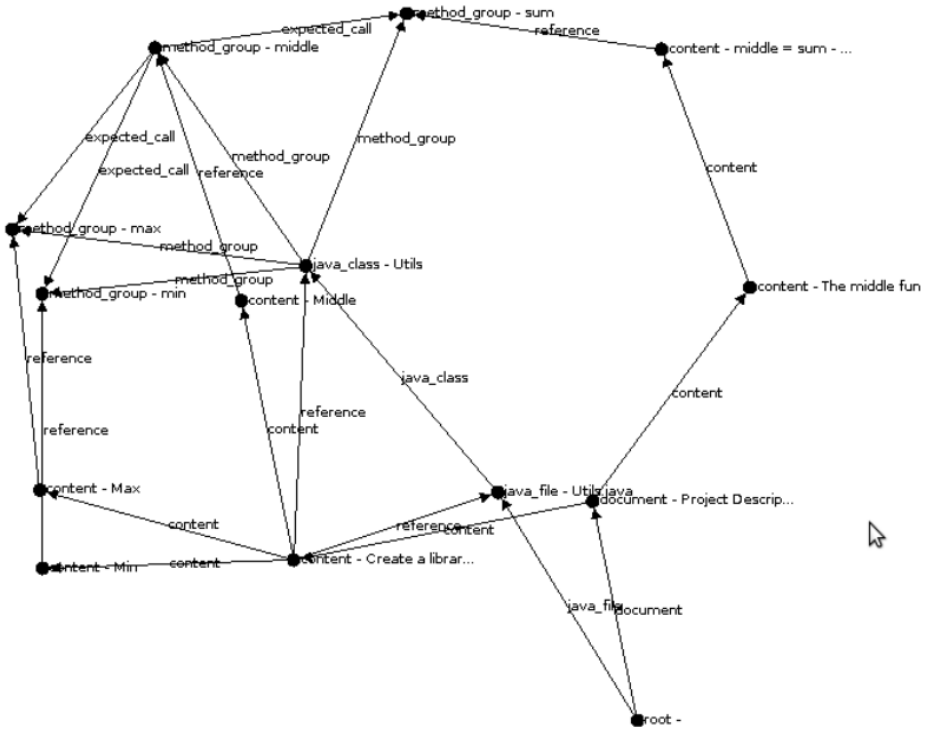


Figure 12: `Expected Call` added between all related `Nodes` in `Project Graph`

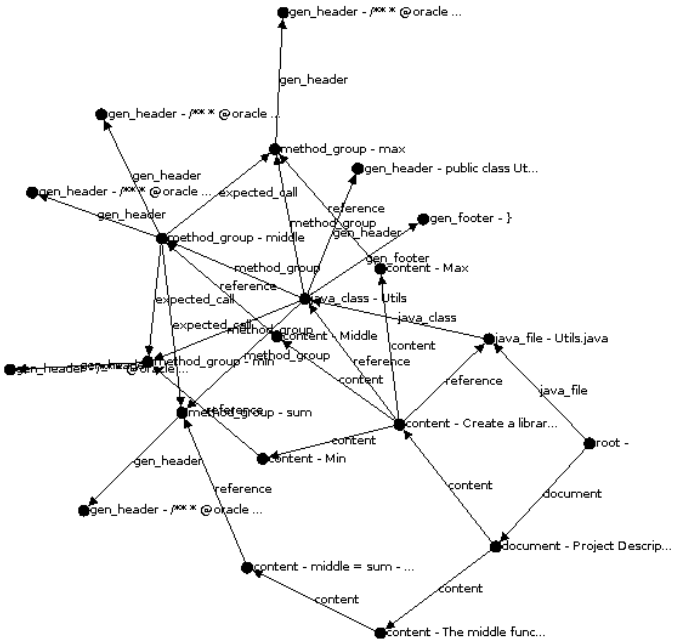


Figure 13: `Project Graph` with code gen `Nodes` added

At this point the graph is already getting cluttered and confusing even for such a small system.

1 Initial Generated Code

Continuing with the `Use Case`, running our `Project Graph` through the code generator we get a Java File that can be seen in Section 1.

```
public class Utils {
   /**
    * @oracle method max
    *
    */

   /**
    * @oracle method min
    *
    */

   /**
    * @oracle method middle
    *
    * Calls:
    *    TODO: method sum
    *    TODO: method max
    *    TODO: method min
    */

   /**
    * @oracle method sum
    *
    */
}
```

2 Completed Code

```
import java.lang.Math;

/**
 * This class consists exclusively of static methods that provide utility
 * functions to the developer.
 *
 * @author Jean
 *
 */

public class Utils {

   /**
    * @oracle method max
    *
    */

   /**
    * Determine the largest of 2 values
    *
    * @param i
    * @param j
    * @return largest value
    */
   public static int max(int i, int j) {
      return Math.max(i, j);
   }

   /**
    * Determine the largest of 3 values
    *
    * @param i
    * @param j
    * @param k
    * @return largest value
    */
   public static int max(int i, int j, int k) {
      return max(i, max(j, k));
   }

   /**
    * @oracle method min
    *
    */

   /**
    * Determine the smallest of 2 values
```

```
 *
 * @param i
 * @param j
 * @return smallest value
 */
public static int min(int i, int j) {
   return Math.min(i, j);
}

/**
 * Determine the smallest of 3 values
 *
 * @param i
 * @param j
 * @param k
 * @return smallest value
 */
public static int min(int i, int j, int k) {
   return min(i, min(j, k));
}

/**
 * @oracle method sum
 *
 */

/**
 * Calculates the sum of 2 values
 *
 * @param i
 * @param j
 * @return sum
 */
public static int sum(int i, int j) {
   return i + j;
}

/**
 * Calculates the sum of 3 values
 *
 * @param i
 * @param j
 * @param k
 * @return sum
 */
public static int sum(int i, int j, int k) {
   return sum(i, sum(j, k));
}

/**
```

```
     * @oracle method middle
     *
     *        Calls: TODO: method sum TODO: method max TODO: method min
     */

    /**
     * Determines the value between the smallest and largest of the values
     * supplied
     *
     * @param i
     * @param j
     * @param k
     * @return middle value
     */
    public static int middle(int i, int j, int k) {
      return sum(i, j, k) - max(i, j, k) - min(i, j, k);
    }
}
```

Reverse Engineering

Now that the code is completed we would like to reverse engineer it and extract the respective `Project Graphs`.

Byte Code Analysis

Running the Byte Code Analysis on the completed Java File generates the graph in Figure 14.

Source Code Analysis

Running the Source Code Analysis on the completed Java File generates the graph in Figure 15.

Project Graph Merging

The previous sections each focused on extracting a specific part of the current project, but to fully utilise all the information the 2 of them must be combined. Combining the Java Byte Code Analysis and Java Source Code Analysis yields the graph in Figure 16.

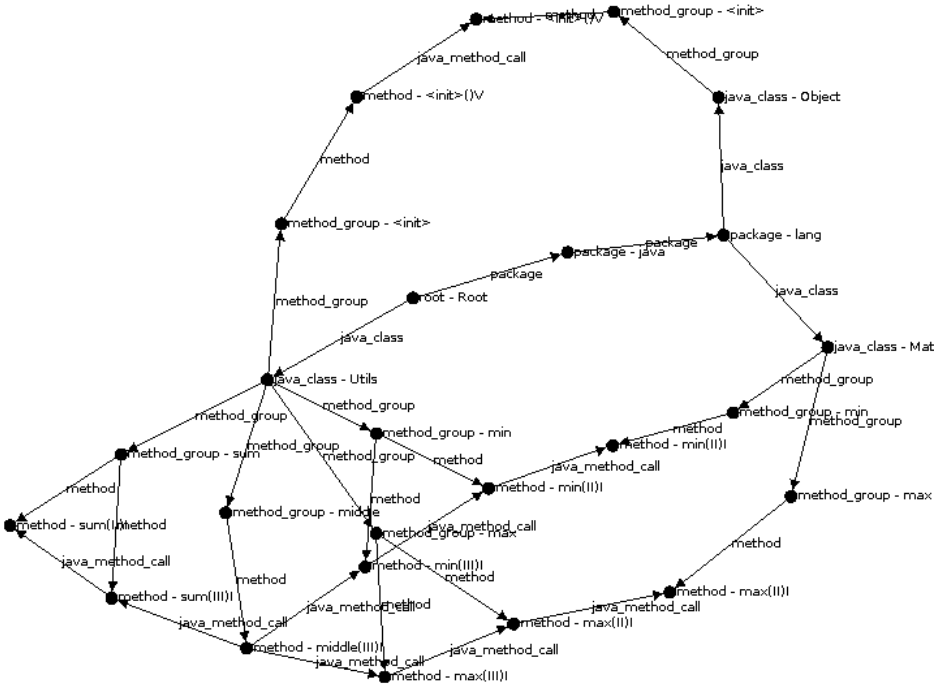


Figure 14: Graph Generated from Byte Code Analysis

Figure 17 through 25 shows the visualisation for the initial design and the design after the code has been developed. The figures also show different layers of the system by having some nodes expanded or collapsed.

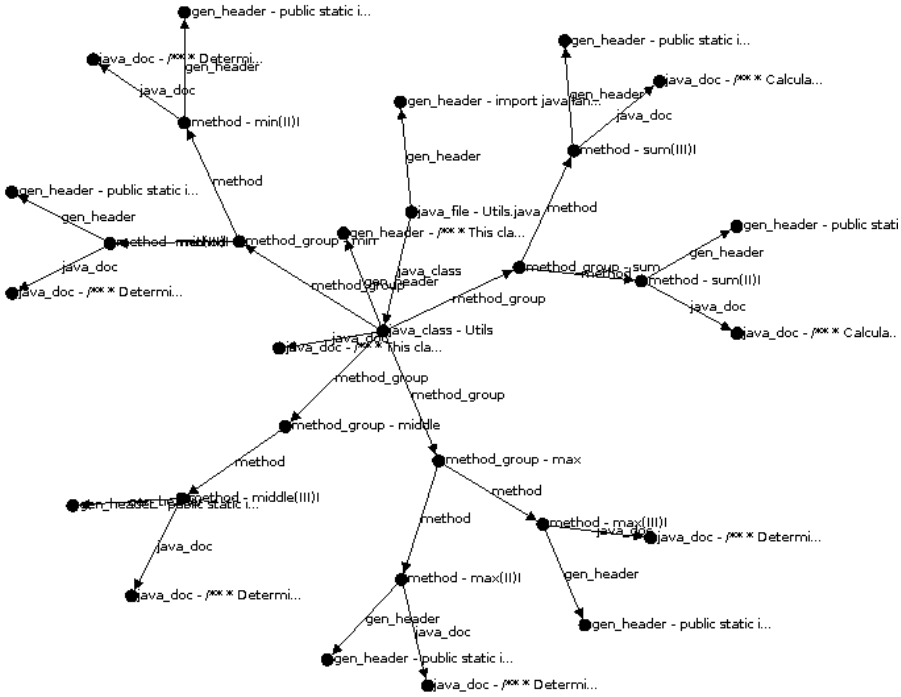


Figure 15: Graph generated from `Java Source Code Analysis`

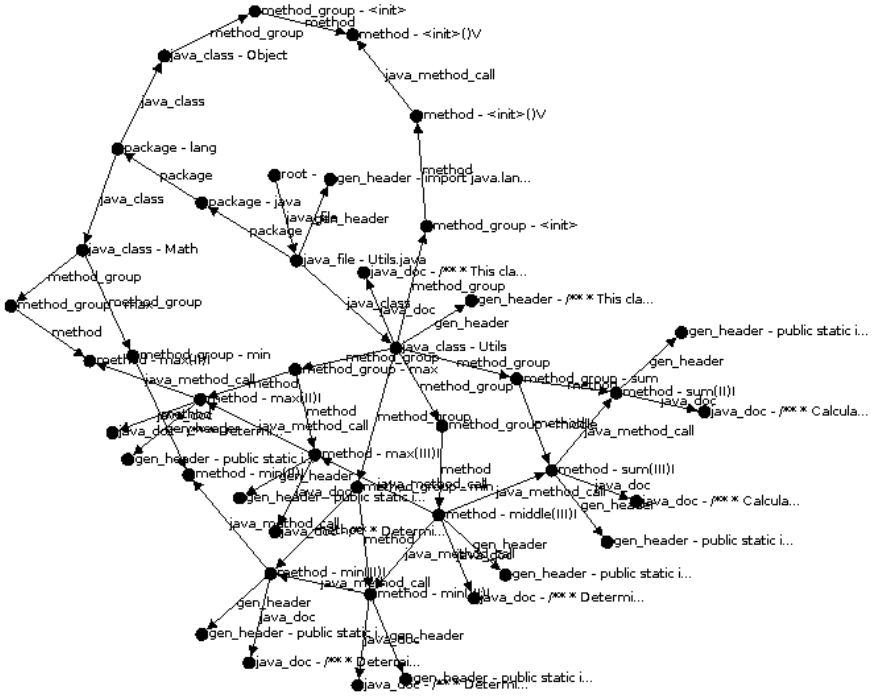


Figure 16: Combination of Java Byte Code and Java Source Code

Figure 17: Visualisation of initial design phase

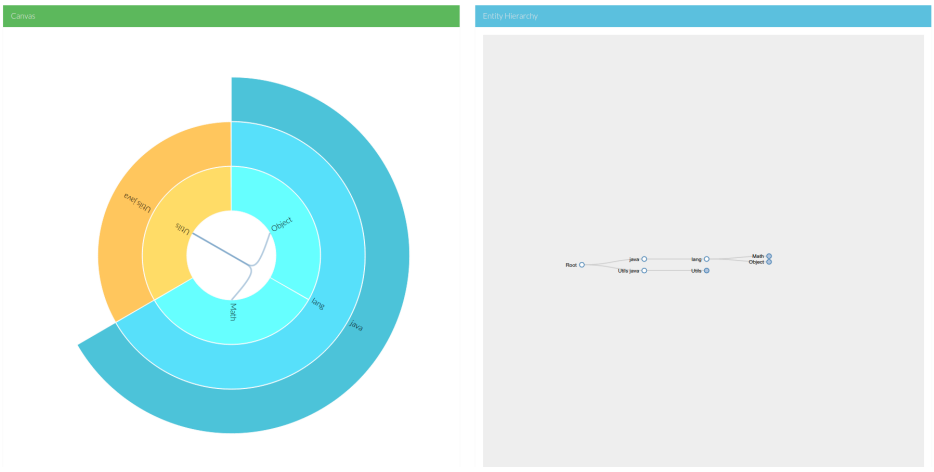


Figure 18: Visualisation of project after merging analysis into Project Graph

Figure 19: Visualisation of project after merging analysis into Project Graph

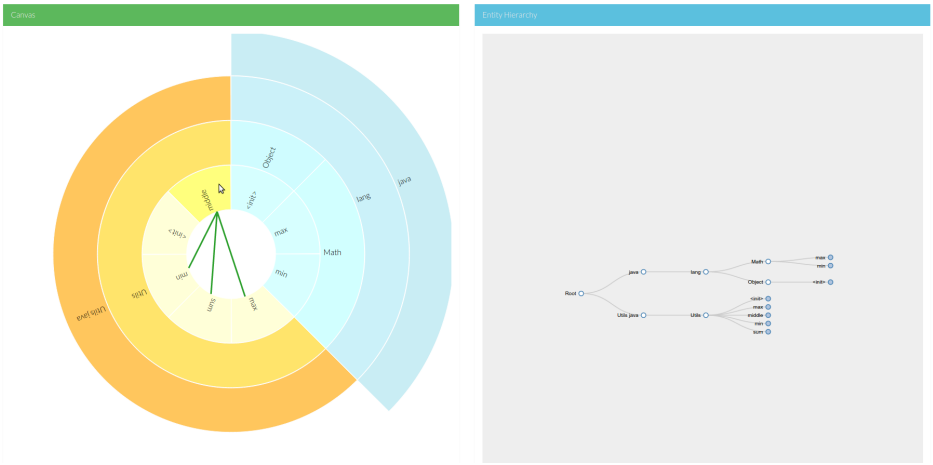


Figure 20: Visualisation of project after merging analysis into Project Graph

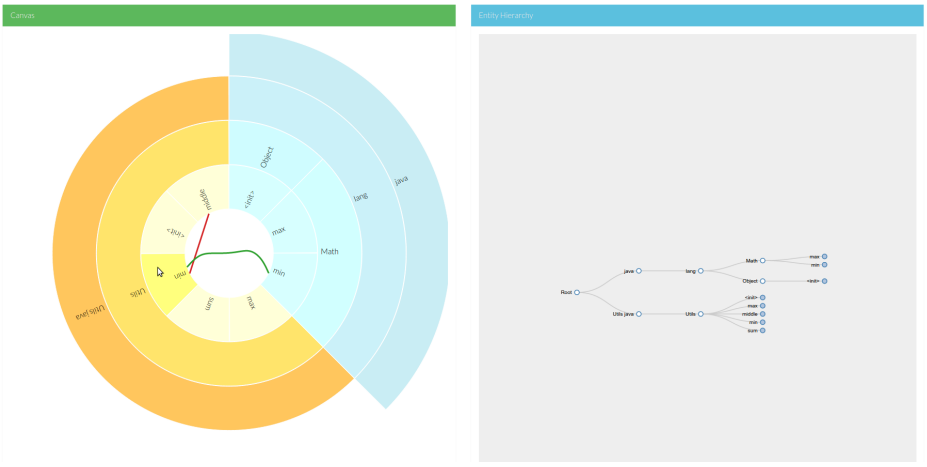


Figure 21: Visualisation of project after merging analysis into Project Graph

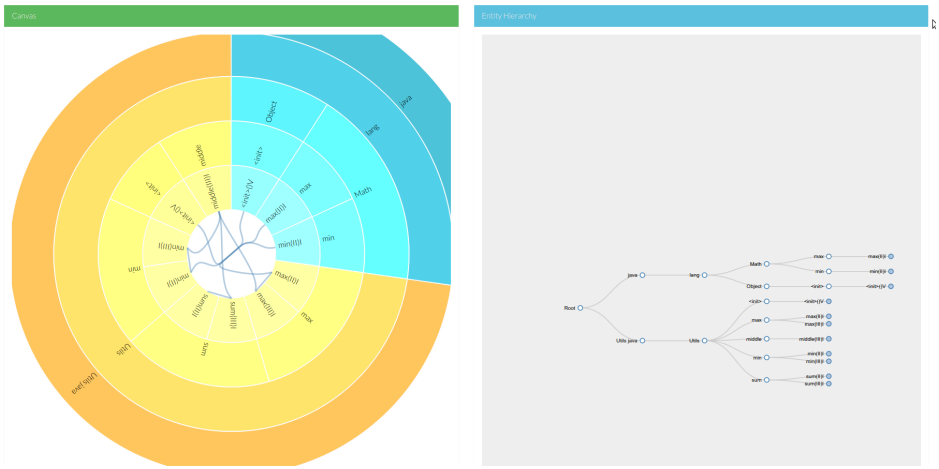


Figure 22: Visualisation of project after merging analysis into Project Graph

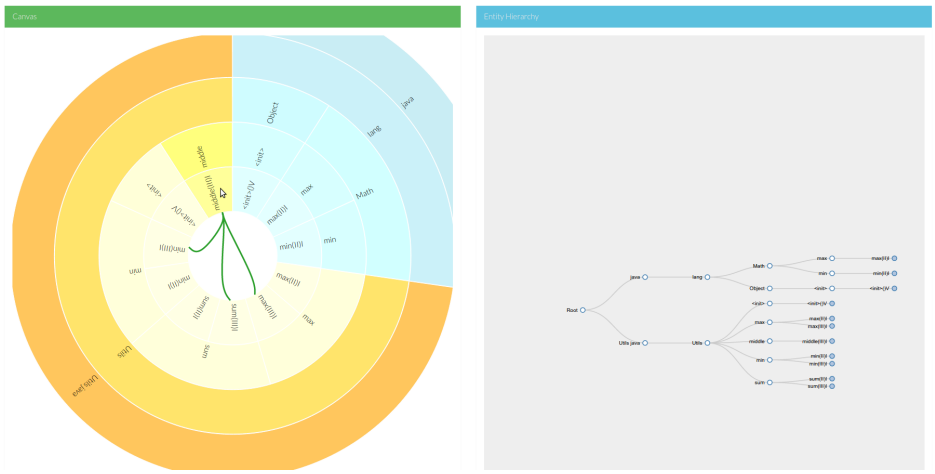


Figure 23: Visualisation of project after merging analysis into Project Graph

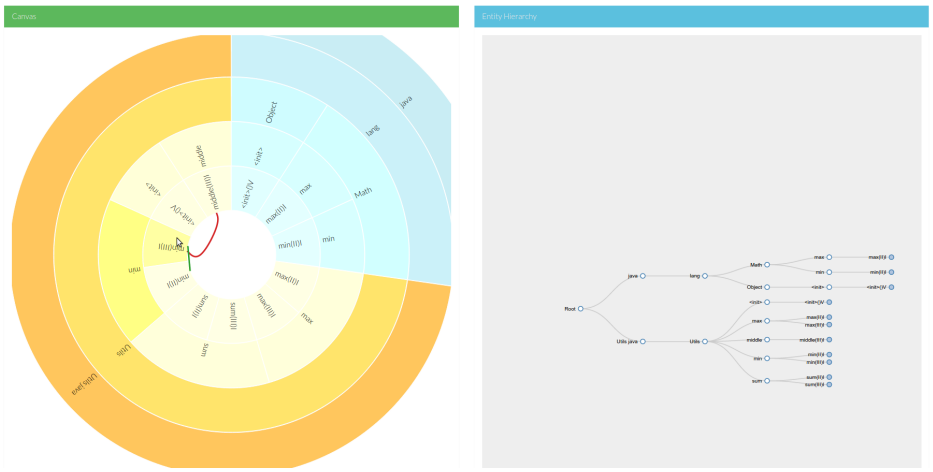


Figure 24: Visualisation of project after merging analysis into Project Graph

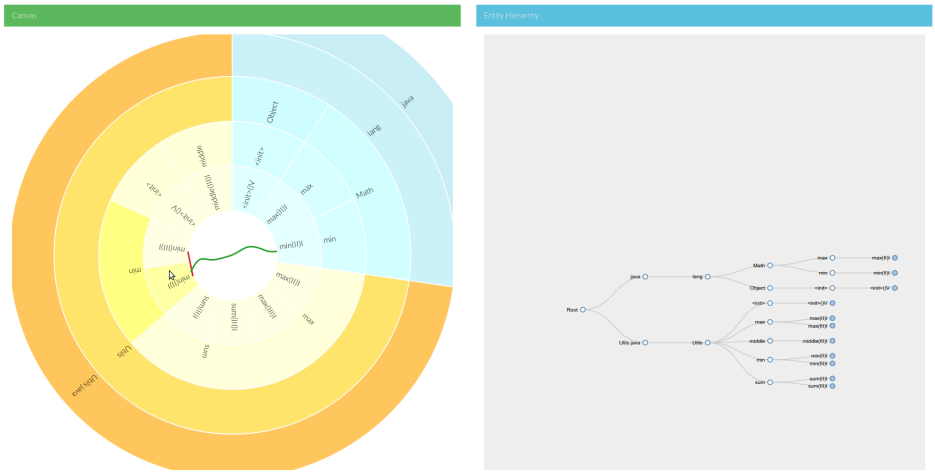


Figure 25: Visualisation of project after merging analysis into Project Graph

Appendix : B

This appendix demonstrates how the `Methodology` is used to derive an initial conceptual view of the system to be designed before committing to types.

The following is an extract from a Natural Language document. The content is from a currently unpublished paper by Kobayashi, Ishikawa and Honiden. Each of the following bullet points is considered a `statement` from the document.

- Either # cars going left on the bridge, or # cars going right on the bridge is zero.
- If the traffic light on the mainland is green, # cars going right on the bridge is zero.
- The sum of # cars going left on the bridge, # cars going right on the bridge, and # cars on the island is less than or equal to the capacity of outside.
- If the traffic light on the mainland is green, the sum of # cars going left on the bridge, # cars going right on the bridge, and # cars on the island is strictly less than zero.
- If the traffic light on the island is green, # cars going left on the bridge is zero.
- If the traffic light on the island is green, # cars on the island is greater than zero.
- Either the traffic light on the island or the traffic light on the mainland is red.

`Either # cars going left on the bridge, or # cars going right on the bridge is zero.`

The following `concepts` and their `hierarchical relations` were identified:

- Root → car
- Root → bridge
- Root → bridge → cars going left
- Root → bridge → cars going right

The following `general relations` were identified from these `concepts`:

- Root → bridge → cars going left $\Longrightarrow$ Root → car
- Root → bridge → cars going right $\Longrightarrow$ Root → car

The following `constraints` were identified:

- Root → bridge → is cars going left or right zero

Refinement for: Root → bridge → is cars going left or right zero

- (when?) Root → bridge → cars going left change $\Longrightarrow$ Root → bridge → is cars going left or right zero
- (when?) Root → bridge → cars going right change $\Longrightarrow$ Root → bridge → is cars going left or right zero
- (how?) Root → bridge → is cars going left zero $\Longrightarrow$ Root → bridge → is cars going left or right zero
- (how?) Root → bridge → is cars going right zero $\Longrightarrow$ Root → bridge → is cars going left or right zero

Refinement for: Root → bridge → cars going left change

- (when?) Root → bridge → set cars going left $\Longrightarrow$ Root → bridge → cars going left change

Refinement for: Root → bridge → cars going right change

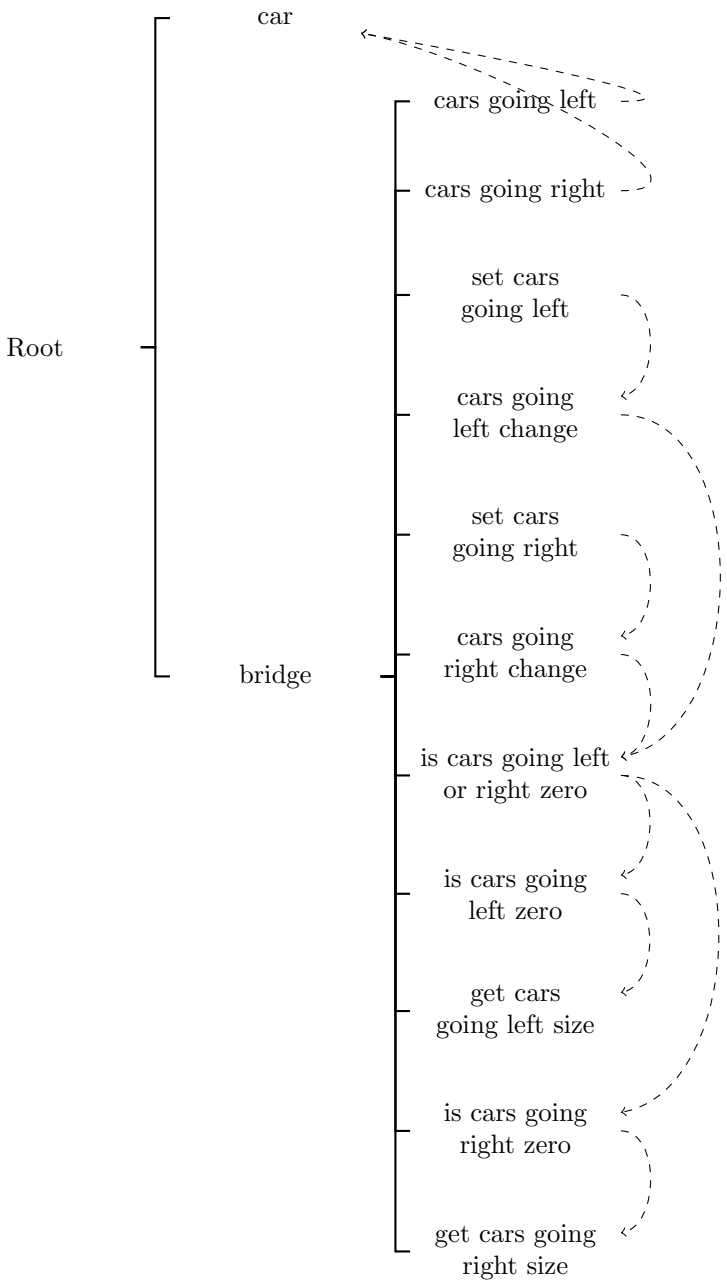
- (when?) Root → bridge → set cars going right $\Longrightarrow$ Root → bridge → cars going right change

Refinement for: Root → bridge → is cars going left zero

- Root → bridge → is cars going left zero $\Longrightarrow$ Root → bridge → get cars going left size (how?)

Refinement for: Root → bridge → is cars going right zero

- Root → bridge → is cars going right zero ⟹ Root → bridge → get cars going right size (how?)



```
If the traffic light on the mainland is green, # cars going right
on the bridge is zero.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → traffic light
- Root → mainland
- Root → mainland → traffic light
- Root → traffic light → green

The following `general relations` were identified from these `concepts`:

- Root → mainland → traffic light $\Longrightarrow$ Root → traffic light

The following `constraints` were identified:

- Root → constraint checker[1] → validate when mainland traffic light is green

Refinement for: Root → constraint checker → validate when mainland traffic light is green

- (when?) Root → mainland → on traffic light green $\Longrightarrow$ Root → constraint checker → validate when mainland traffic light is green
- (how?) Root → constraint checker → validate when mainland traffic light is green $\Longrightarrow$ Root → bridge → is cars going right zero

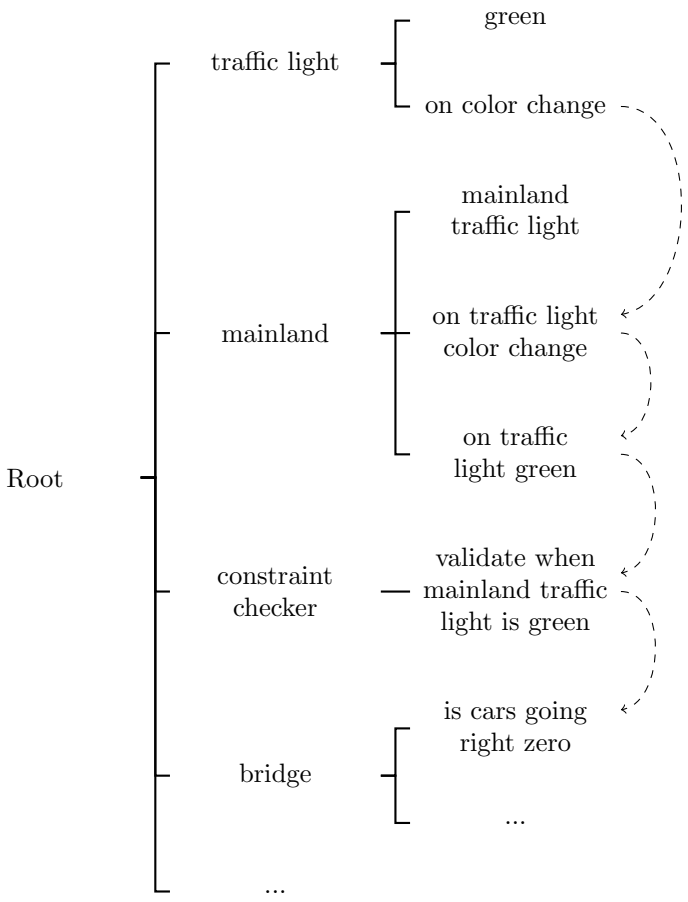
Refinement for: Root → mainland → on traffic light green

- (when?) Root → mainland → on traffic light color change $\Longrightarrow$ Root → mainland → on traffic light green

Refinement for: Root → mainland → on traffic light color change

- (when?) Root → traffic light → on color change $\Longrightarrow$ Root → mainland → on traffic light color change

[1] there is no logical local place to add the checking of the constraint so we create a new concept that will be doing some of the checking for us, assuming that this concept will eventually be a class with references to the other classes that will be needed for the check to be done.


Root
traffic light
green
on color change
mainland
mainland traffic light
on traffic light color change
on traffic light green
constraint checker
validate when mainland traffic light is green
bridge
is cars going right zero
...
...

`The sum of # cars going left on the bridge, # cars going right on the bridge, and # cars on the island is less than or equal to the capacity of outside.`

The following `concepts` and their `hierarchical relations` were identified:

- Root → island
- Root → island → cars
- Root → outside
- Root → outside → capacity

The following `general relations` were identified from these `concepts`:

- Root → island → cars $\Longrightarrow$ Root → car

The following `constraints` were identified:

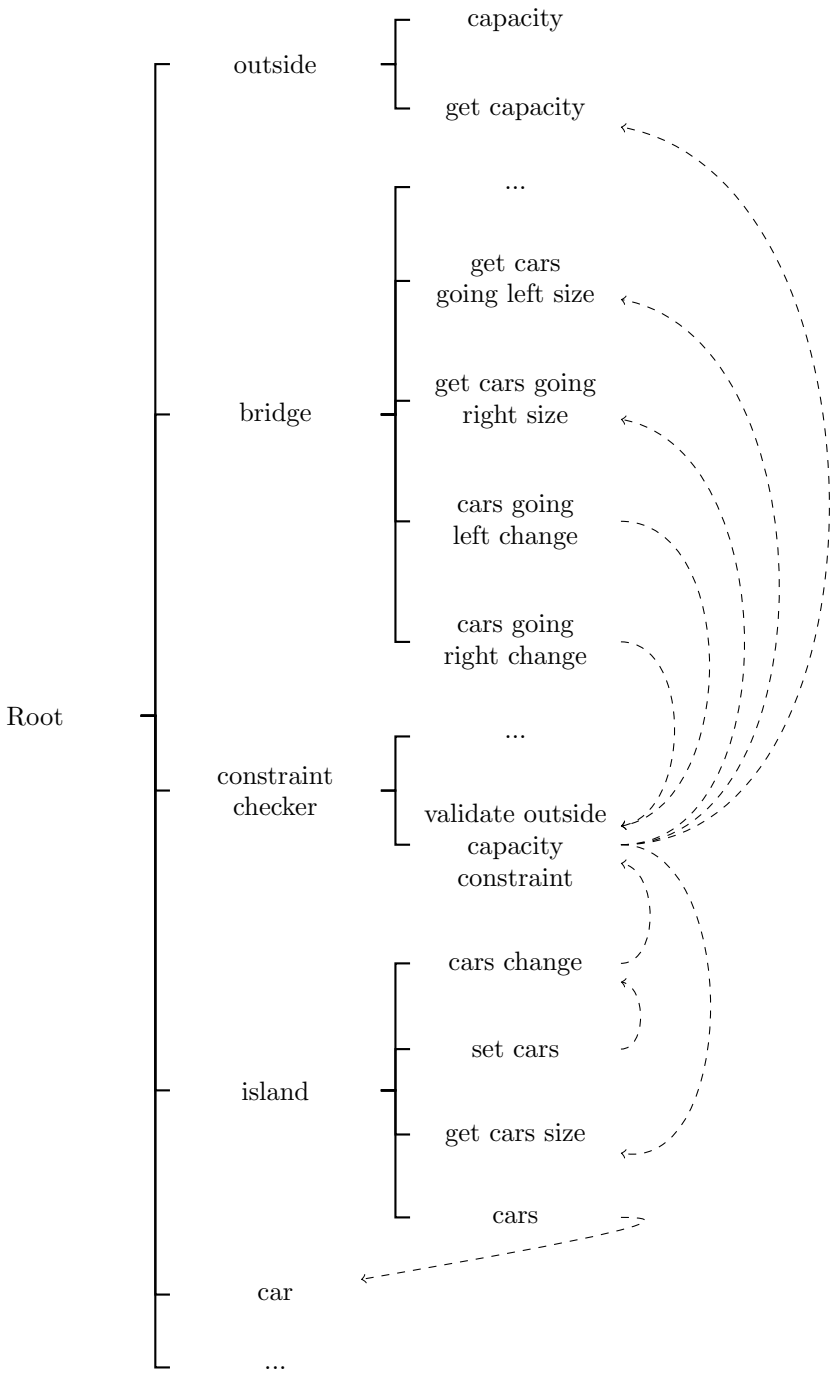
- Root → constraint checker → validate outside capacity constraint

Refinement for: Root → constraint checker → validate outside capacity

- (when?) Root → bridge → cars going left change $\Longrightarrow$ Root → constraint checker → validate outside capacity
- (when?) Root → bridge → cars going right change $\Longrightarrow$ Root → constraint checker → validate outside capacity
- (when?) Root → island → cars change $\Longrightarrow$ Root → constraint checker → validate outside capacity
- (how?) Root → constraint checker → validate outside capacity $\Longrightarrow$ Root → bridge → get cars going left size
- (how?) Root → constraint checker → validate outside capacity $\Longrightarrow$ Root → bridge → get cars going right size
- (how?) Root → constraint checker → validate outside capacity $\Longrightarrow$ Root → island → get cars size
- (how?) Root → constraint checker → validate outside capacity $\Longrightarrow$ Root → outside → get capacity

Refinement for : Root → island → cars change

- Root → island → set cars $\Longrightarrow$ Root → island → cars change


Root
outside
capacity
get capacity
bridge
...
get cars going left size
get cars going right size
cars going left change
cars going right change
constraint checker
...
validate outside capacity constraint
island
cars change
set cars
get cars size
cars
car
...

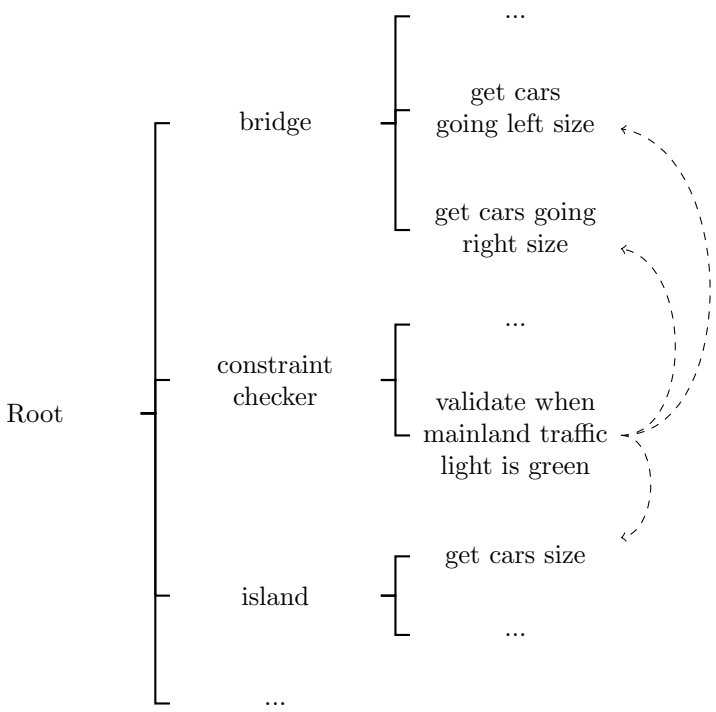
`If the traffic light on the mainland is green, the sum of # cars going left on the bridge, # cars going right on the bridge, and # cars on the island is strictly less than zero.`

The following `constraints` were identified:

- Root → constraint checker → validate when mainland traffic light is green

Refinement for: Root → constraint checker → validate when mainland traffic light is green

- (how?) Root → constraint checker → validate when mainland traffic light is green ⟹ Root → bridge → get cars going left size
- (how?) Root → constraint checker → validate when mainland traffic light is green ⟹ Root → bridge → get cars going right size
- (how?) Root → constraint checker → validate when mainland traffic light is green ⟹ Root → island → get cars size



```
If the traffic light on the island is green, # cars going left on
the bridge is zero.
```

The following `constraints` were identified:

- Root → constraint checker → validate when island traffic light is green

Refinement for: Root → constraint checker → validate when island traffic light is green

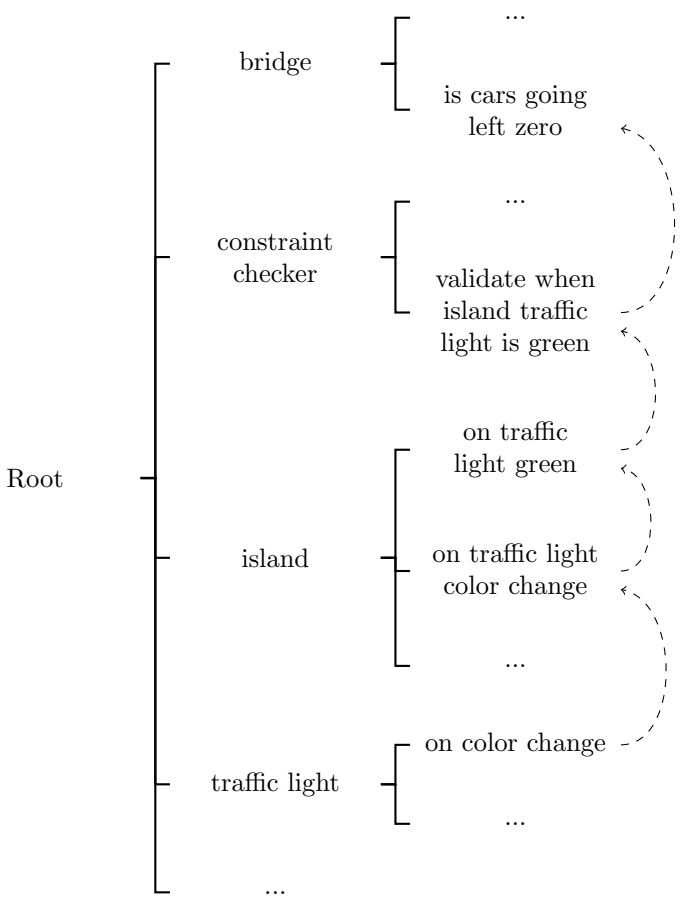
- (when?) Root → island → on traffic light green $\Longrightarrow$ Root → constraint checker → validate when island traffic is green
- (how?) Root → constraint checker → validate when island traffic light is green $\Longrightarrow$ Root → bridge → get cars going left size

Refinement for: Root → island → on traffic light green

- (when?) Root → island → on traffic light color change $\Longrightarrow$ Root → island → on traffic light green

Refinement for: Root → island → on traffic light color change

- (when?) Root → traffic light → on color change $\Longrightarrow$ Root → island → on traffic light color change


Root
bridge
...
is cars going left zero
constraint checker
...
validate when island traffic light is green
island
on traffic light green
on traffic light color change
...
traffic light
on color change
...
...

```
If the traffic light on the island is green, # cars on the island
is greater than zero.
```

The following `constraints` were identified:

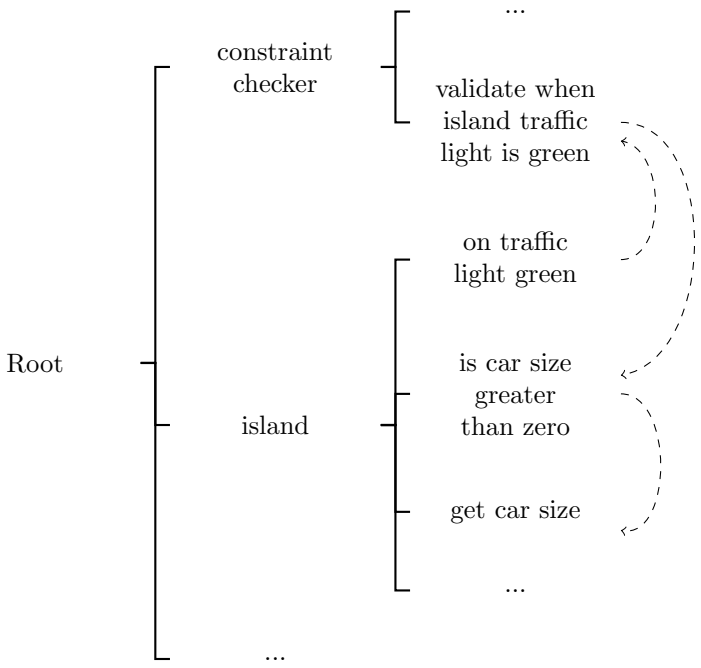
- Root → constraint checker → validate when island traffic is green

Refinement for: Root → constraint checker → validate when island traffic is green

- (how?) Root → constraint checker → validate when island traffic is green ⟹ Root → island → is car size greater than zero

Refinement for: Root → island → is car size greater than zero

- (how?) Root → island → is car size greater than zero ⟹ Root → island → get cars size



```
Either the traffic light on the island or the traffic light on
the mainland is red.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → traffic light → red

The following `constraints` were identified:

- Root → constraint checker → is island traffic or mainland traffic red

Refinement for: Root → constraint checker → is island traffic or mainland traffic red

- (when?) Root → traffic light → on color change $\Longrightarrow$ Root → constraint checker → is island traffic or mainland traffic red
- (how?) Root → constraint checker → is island traffic or mainland traffic red $\Longrightarrow$ Root → island → is traffic light red
- (how) Root → constraint checker → is island traffic or mainland traffic red $\Longrightarrow$ Root → mainland → is traffic light red

Refine for: Root → island → is traffic light red

- (how?) Root → island → is traffic light red $\Longrightarrow$ Root → island → get traffic light color

Refine for: Root → island → get traffic light color

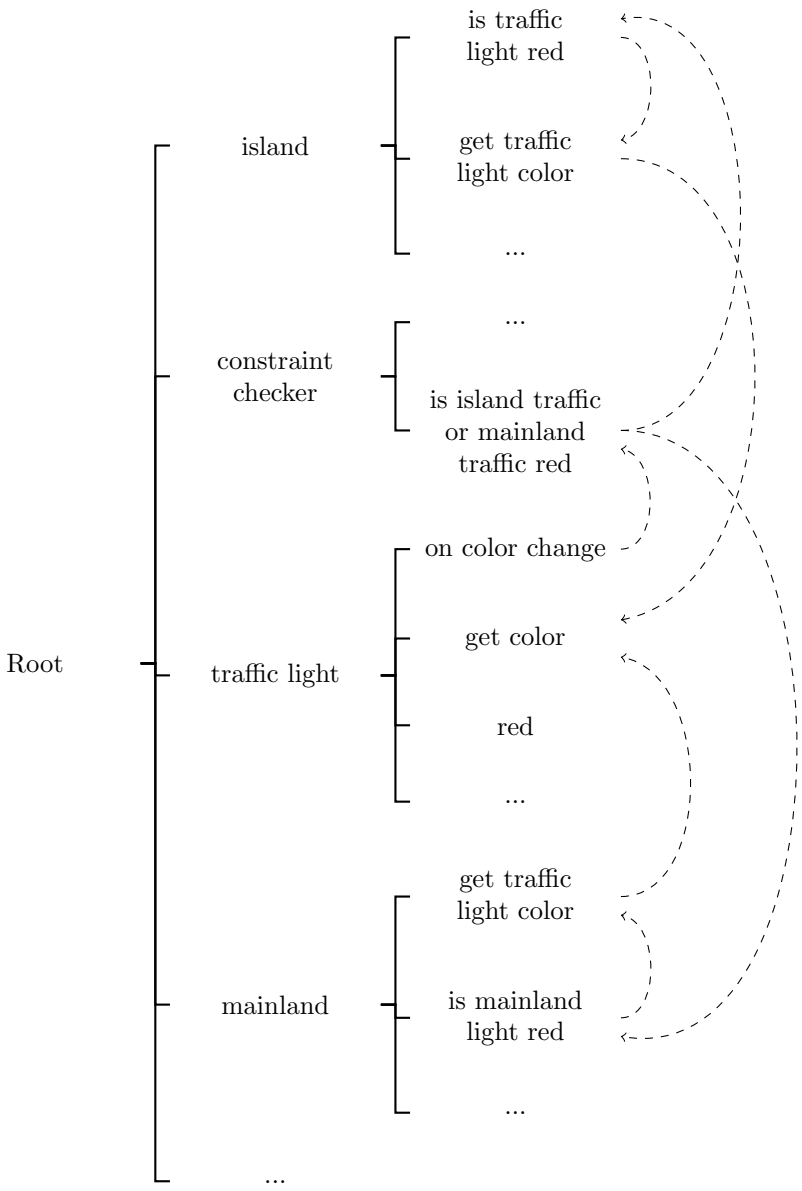
- (how?) Root → island → get traffic light color $\Longrightarrow$ Root → traffic light → get color

Refine for: Root → mainland → is traffic light red

- (how?) Root → mainland → is traffic light red $\Longrightarrow$ Root → mainland → get traffic light color

Refine for: Root → mainland → get traffic light color

- (how?) Root → mainland → get traffic light color $\Longrightarrow$ Root → traffic light → get color


Root
island
constraint checker
traffic light
mainland
...
is traffic light red
get traffic light color
...
...
is island traffic or mainland traffic red
on color change
get color
red
...
get traffic light color
is mainland light red
...

Appendix : C

This appendix demonstrates how the `Methodology` is used to derive an initial conceptual view of the system to be designed before committing to types.

The following is an extract from a Natural Language document. Each of the following bullet points is considered a `statement` from the document.

- E-Z-Move offers local, national, and international relocations.
- All three types of relocations have a pick-up and a delivery address; local relocations must have pick-up and delivery in the same town, while international relocations also require two customs clearance centres (one of them must be in SA).
- For each relocation type we record the name and phone number of the contact person.
- When customers book a move they need to give the pick-up date and time, the addresses and the volume (in m^3) they need.
- There are customers that often book with E-Z-Move (e.g., UCT and SU), so the customer data needs to be saved in the system.
- New customers need to be added only when they book their first relocation.
- Each truck has a given capacity (in m^3). If a move is too big for the biggest available truck it needs to be split into two orders. (Adding support for booking multiple trucks is left for V2.0).
- For each relocation order we need to reserve the truck, the driver, and the packers. If one of those isn't available on the scheduled date we cannot accept the order.
- E-Z-Move has two types of trucks, panel vans and tractor trailers. Panel vans can only be used for local relocations.
- Customers typically order an entire truck for their exclusive use, but E-Z-Move also offers truck sharing: if a truck has still space available, then the system should place an offer on gumtree. Customers can then book into this shared truck by quoting the move's reference number.

- If a customer books into an existing shared truck, the ad needs to be removed and an updated ad needs to be placed.
- If trucks are available, E-Z-Move office staff sometimes puts shared trucks up on gumtree even if there's no customer yet.
- When a driver gets scheduled for a tour he needs to be informed immediately by the system; the driver then phones the packers (which don't have computer access). Each driver only works with one team of packers.
- Since the trucks are used a lot, it is important to schedule repairs and services for them.
- During the time a truck is in the shop for repair or service it is unavailable for a relocation.
- Customers can change their order as long as there's still space on the truck. They cannot cancel their order if the truck was ordered for their exclusive use.
- Every staff member needs to login before using the system and can then only see the necessary information for his area of work.
- The system must give informative error messages.
- The office staff are responsible for putting all necessary data into the system, e.g. truck data,driver data...

```
E-Z-Move offers local, national, and international relocations.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → relocation
- Root → relocation → local
- Root → relocation → national
- Root → relocation → international

```
All three types of relocations have a pick-up and a delivery
address; local relocations must have pick-up and delivery in
the same town, while international relocations also require two
customs clearance centres (one of them must be in SA).
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → relocation → pick-up address
- Root → relocation → delivery address
- Root → relocation → customs clearance centre 1
- Root → relocation → customs clearance centre 2

The following `constraints` were identified:

- Root → relocation → validate local relocation same town
- Root → relocation → validate international customs clearance

Refinement for: Root → relocation → validate local relocation same town

- (related?) Root → relocation → type → local
- (related?) Root → relocation → type → national
- (related?) Root → relocation → type → international
- (when?) Root → relocation → on local relocation $\Longrightarrow$ Root → relocation → validate local relocation same town
- (how?) Root → relocation → validate local relocation same town $\Longrightarrow$ Root → relocation → validate pick-up and delivery address same town

Refinement for: Root → relocation → on local relocation

- (when?) Root → relocation → on relocation type change $\Longrightarrow$ Root → relocation → on local relocation

Refinement for: Root → relocation → on relocation type change

- (when?) Root → relocation → set type ⟹ Root → relocation → on relocation type change

Refinement for: Root → relocation → validate pick-up and delivery address same town

- Root → address
- Root → address → town
- Root → relocation → pick-up address ⟹ Root → address
- Root → relocation → delivery address ⟹ Root → address
- (how?) Root → relocation → validate pick-up and delivery address same town ⟹ Root → relocation →get pick-up address
- (how?) Root → relocation → validate pick-up and delivery address same town ⟹ Root → relocation →get delivery address

```
For each relocation type we record the name and phone number of
the contact person.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → relocation → name
- Root → relocation → phone number

```
When customers book a move they need to give the pick-up date and
time, the addresses and the volume (in m^3) they need.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → customer
- Root → customer → book move
- Root → relocation → book
- Root → relocation → volume
- Root → relocation → pick-up date time

```
There are customers that often book with E-Z-Move (e.g., UCT and
SU), so the customer data needs to be saved in the system.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → customer → save

```
New customers need to be added only when they book their first
relocation.
```

The following `constraints` were identified:

- Root → relocation → save customer if new

Refinement for: Root → relocation → save customer if new

- (when?) Root → relocation → book ⟹ Root → relocation → save customer if new
- (how?) Root → relocation → save customer if new ⟹ Root → customer → does customer exist
- (how?) Root → relocation → save customer if new ⟹ Root → customer → save

Refinement for: Root → Customer → does customer exist

- (how?) Root → Customer → does customer exist ⟹ Root → customer → get customer

```
Each truck has a given capacity (in m^3).  If a move is too big
for the biggest available truck it needs to be split into two
orders.  (Adding support for booking multiple trucks is left for
V2.0).
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → truck
- Root → truck → capacity
- Root → relocation → truck

The following `general relations` were identified from these `concepts`:

- Root → relocation → truck ⟹ Root → truck

The following `constraints` were identified:

- Root → relocation → is truck available for capacity

Refinement for: Root → relocation → is truck available for capacity

- (when?) Root → relocation → set capacity ⟹ Root → relocation → is truck available for capacity
- (how?) Root → relocation → is truck available for capacity ⟹ Root → truck → get largest truck
- (how?) Root → truck → get capacity

Refinement for: Root → truck → get largest truck

- (how?) Root → truck → get largest truck ⟹ Root → truck → get largest truck

Refinement for: Root → truck → get largest truck

- (how?) Root → truck → get largest truck ⟹ Root → truck → get all available trucks
- (how?) Root → truck → get largest truck ⟹ Root → truck → sort trucks by capacity

```
For each relocation order we need to reserve the truck, the
driver, and the packers.  If one of those isn't available on the
scheduled date we cannot accept the order.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → driver
- Root → packer
- Root → relocation → driver
- Root → relocation → packer
- Root → driver → reserved dates
- Root → packer → reserved dates

The following `general relations` were identified from these `concepts`:

- Root → relocation → driver ⟹ Root → driver
- Root → relocation → packer ⟹ Root → packer

The following `constraints` were identified:

- Root → relocation → is driver available for date time
- Root → relocation → is packer available for date time

Refinement for: Root → relocation → is driver available for date time

- (when?) Root → relocation → book ⟹ Root → relocation → is driver available for date time
- (how?) Root → relocation → is driver available for date time ⟹ Root → DB → driver → get driver available on date time
- (how?) Root → relocation → is driver available for date time ⟹ Root → relocation → set driver
- (how?) Root → relocation → is driver available for date time ⟹ Root → driver → add reserved date

Refinement for: Root → driver → get driver available on date time

- (how?) Root → driver → get driver available on date time ⟹ Root → driver → get all drivers
- (how?) Root → driver → get driver available on date time ⟹ Root → driver → is driver available on date time

Refinement for: Root → driver → is driver available on date time

- (how?) Root → driver → is driver available on date time ⟹ Root → driver → get reserved dates

Refinement for: Root → relocation → is packer available for date time

- (when?) Root → relocation → book ⟹ Root → relocation → is packer available for date time
- (how?) Root → relocation → is packer available for date time ⟹ Root → packer → get driver available on date time
- (how?) Root → relocation → is packer available for date time ⟹ Root → relocation → set packer
- (how?) Root → relocation → is packer available for date time ⟹ Root → packer → add reserved date

Refinement for: Root → packer → get packer available on date time

- (how?) Root → packer → get packer available on date time ⟹ Root → packer → get all packers
- (how?) Root → driver → get packer available on date time ⟹ Root → packer → is packer available on date time

Refinement for: Root → packer → is packer available on date time

- (how?) Root → packer → is packer available on date time ⟹ Root → packer → get reserved dates

```
E-Z-Move has two types of trucks, panel vans and tractor
trailers.  Panel vans can only be used for local relocations.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → truck → type
- Root → truck → type → panel van
- Root → truck → type → tractor trailer

The following `constraints` were identified:

- Root → truck → validate local for panel van

Refinement for: Root → truck → validate local for panel van

- (when?) Root → relocation → on local relocation ⟹ Root → truck → validate local for panel van
- (how?) Root → truck → validate local for panel van ⟹ Root → truck → get type

```
Customers typically order an entire truck for their exclusive
use, but E-Z-Move also offers truck sharing:  if a truck has
still space available, then the system should place an offer
on gumtree.  Customers can then book into this shared truck by
quoting the move's reference number.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → relocation → book with reference number
- Root → truck → book
- Root → gumtree
- Root → gumtree → place add

The following `general relations` were identified from these `concepts`:

- Root → relocation → book with reference number ⟹ Root → relocation → book
- Root → relocation → book ⟹ Root → truck → book

The following `constraints` were identified:

- Root → relocation → does reference number exist
- Root → relocation → if shared truck place ad on gumtree

Refinement for: Root → relocation → does reference number exist

- (when?) Root → relocation → book with reference number ⟹ Root → relocation → does reference number exist

Refinement for: Root → relocation → if shared truck place ad on gumtree

- (when?) Root → relocation → book ⟹ Root → relocation → if shared truck place ad on gumtree
- (how?) Root → relocation → if shared truck place ad on gumtree⟹ Root → gumtree → place add

```
If a customer books into an existing shared truck, the ad needs
to be removed and an updated ad needs to be placed.
```

The following `concepts` and their `hierarchical relations` were identified:

- Root → gumtree → place add

- Root → gumtree → remove add

The following `general relations` were identified from these `concepts`:

- Root → relocation → if shared truck place ad on gumtree ⟹ Root → gumtree → remove add
- Root → relocation → if shared truck place ad on gumtree ⟹ Root → gumtree → place add

For demonstration purposes the rest of the use case is not expanded further for now to show the visualisation at this stage.

Figure 1:

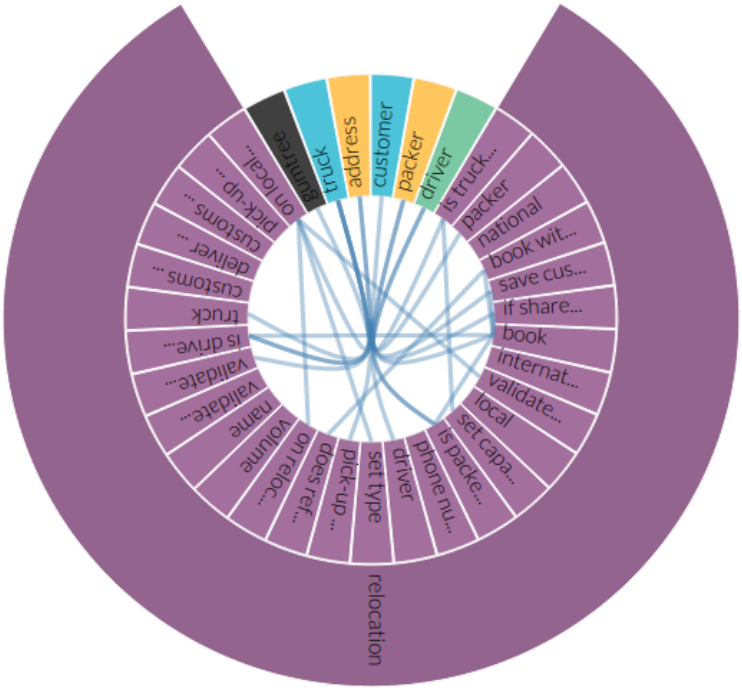

relocation
is truck...
packer
national
book wit...
save cus...
if share...
book
internat...
validate...
local
set capa...
is packe...
phone nu...
driver
set type
pick-up ...
does ref...
onReloc...
volume
name
validate...
validate...
is drive...
truck
customs ...
deliver ...
customs ...
pick-up ...
on local...
quintree
truck
address
customer
packer
driver

Figure 2:

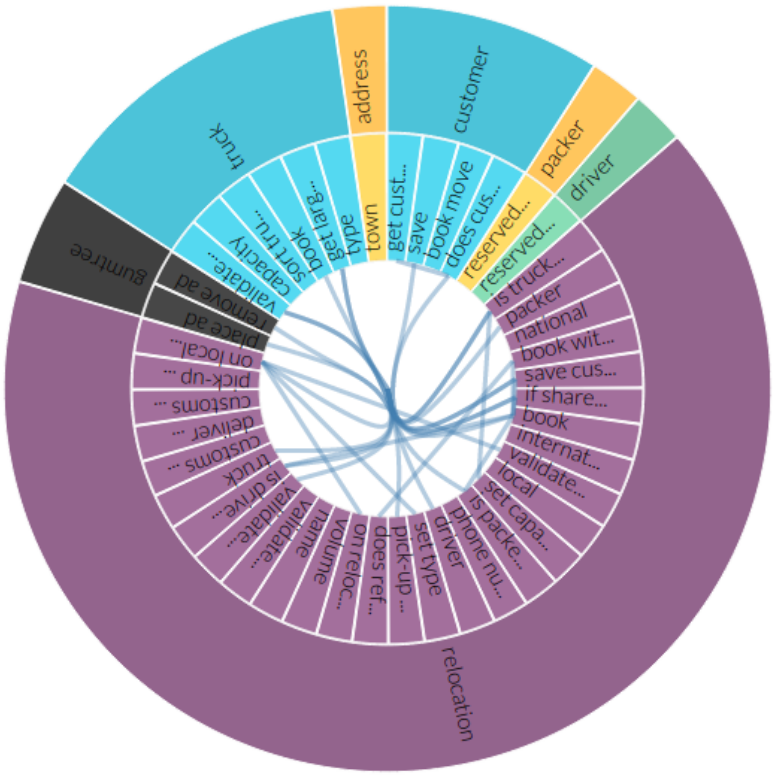

truck
address
customer
packer
driver
gumtree
relocation
type
town
get cust...
save
book move
does cus...
reserved...
reserved...
validate...
sort truc...
book
get tar g...
is truck...
packer
national
book wit...
save cus...
if share...
book
internat...
validate...
local
set capa...
is packe...
phone nu...
driver
set type
pick-up ...
does ref...
on reloc...
volume
name
validate...
validate...
is drive...
truck
customs ...
deliver ...
customs ...
pick-up ...
on local...
place ad
remove ad
validate ad

Figure 3:

Appendix : D

This appendix demonstrates how the `Methodology` is used to derive an initial conceptual view of the system to be designed before committing to types. For this specific one, a 3rd year student (Mat Milne) was given the same set of `statements` as is in Appendix C as the specific use case was used during the CS344 module that he took this semester (2nd semester 2015). The module (Software Engineering) teaches the basics of constructing requirements documents, design documents and the different layers of UML when designing a system. Mat was only given an hour to complete this task to see if the methodology gave him any more insight compared to designing the same system with UML over multiple weeks.

The following was derived during the time he used the tool and methodology:

- Relocations $\rightarrow$ Type $\rightarrow$ Local
- Relocations $\rightarrow$ Type $\rightarrow$ National
- Relocations $\rightarrow$ Address $\rightarrow$ Pick-Up
- Relocations $\rightarrow$ Address $\rightarrow$ Delivery
- Relocations $\rightarrow$ Custom1 $\rightarrow$ Address
- Relocations $\rightarrow$ Custom2 $\rightarrow$ Address
- Relocations $\rightarrow$ LocalAddressesSameTown
- Relocations $\rightarrow$ InternationCustomsClearance
- Relocations $\rightarrow$ setType
- Relocations $\rightarrow$ setCustoms
- Relocations $\rightarrow$ setCustom1
- Relocations $\rightarrow$ setCustom2
- Relocations $\rightarrow$ setType $\Longrightarrow$ Relocations $\rightarrow$ LocalAddressesSameTown
- Relocations $\rightarrow$ setCustoms $\Longrightarrow$ Relocations $\rightarrow$ setCustoms1
- Relocations $\rightarrow$ setCustoms $\Longrightarrow$ Relocations $\rightarrow$ setCustoms2

- Relocations → setCustoms ⟹ Relocations → InternationCustomsClearance
- Address
- Relocations → Custom1 → Address ⟹ Address
- Relocations → Custom2 → Address ⟹ Address
- Relocations → Address → Pick-Up ⟹ Address
- Relocations → Address → Delivery ⟹ Address

Although not much information was derived as Mat did have some questions about the methodology, he clearly understood the way in which he wanted to expand the system but was not always clear how to proceed in doing so in a correct manner as to be able to reuse some ideas. For example when adding the same `Address` information under `Customs1` and `Customs2` he immediately realised this is duplication and needs to be moves somewhere.

With a proper introduction to the methodology and tool I would suspect that these problems and questions can be ironed out and the `methodology` and `tool` used to good effect for similar projects. This would have to be explored with more students though.

Mat also had the following to say about the `methodology` and `tool`

> *"Software was much easier to use and understand than UML, even after only having used it for an hour. The visualization was a big help in trying to extract the information from the Spec. Definitely a better tool in constructing a diagram from the original information provided. I picked up on aspects that I missed in my UML diagram, like the inclusion of of customs and the validation of them. As well as validation of local relocations having pick-up and and delivery addresses in the same town."*

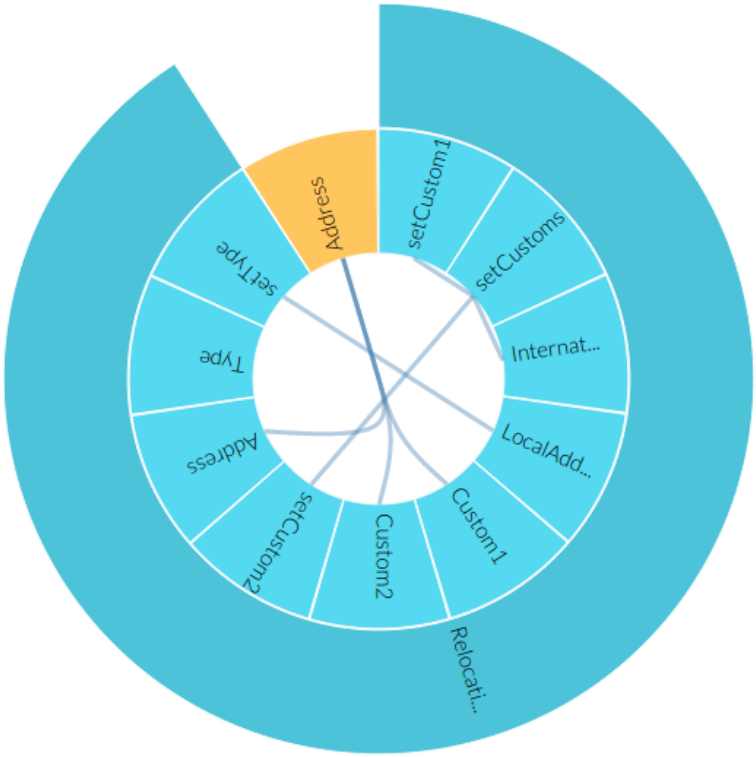


Figure 1: Representation built up by Mat