VRIJE
UNIVERSITEIT
BRUSSEL

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

EDITABLE AND CUSTOMIZABLE KNOWLEDGE MAPS

Thijs Spinoy

June 2019

Promotor:
Prof. Dr. Olga De Troyer

Advisor:
Jan Maushagen

Sciences and Bio-Engineering Sciences

Abstract

Samenvatting

Acknowledgements

Declaration of Originality

Contents

List of Figures

Listings

List of Tables

1 Introduction

Verwijs naar literatuur voor de eerste paragraaf. De paper van Moody is een goed vertrekpunt.
How to visualize, understand and remember a big amount of information which is written down in a long text? People want to store as much information as they can in their brains because the more they know, the less they have to look up and the faster they can proceed. Students for example, make schemes and summaries of their study material. The reason why they do that is because it is easier to learn and remember nicely visualised stuff in comparison to plain text. Not only the way of learning new subject material, but also teamwork can be enhanced by means of a visualization. If you write down a structure of a computer program in words, it is more difficult to discuss that structure than when the same words are translated to a drawing.

The goal of this thesis is to create a tool in which it is possible to represent linked data and knowledge in a visual manner. It should not only be possible to visualize the data, but also to edit existing data as well as extending the representation with additional data. The solution is based on a visualization tool created by Janssens (2013), called GuideaMaps. This application was mainly built to provide support for the requirement elicitation for serious games. It provides the functionality to enter data in pre-defined maps (trees of nodes). We present a new version of GuideaMaps, which can be used for other purposes as well and which has a bunch of interesting improvements under the hood.

1.1 Problem Statement

For the first version of GuideaMaps, the main goal was to develop the following:

> A tool that allows the different people (and with different background) involved in the development of a serious game (e.g., against cyber bullying) to brood over the goals, characteristics and main principles of a new to develop serious game. The tool should be easy to use and usable in meetings. Therefore, we want to explore the characteristics and capabilities of a tablet (i.e. iPad). (Janssens, 2013)

By specifying the goal in this way, end users of the application are restricted in different ways. First, they need an iPad to be able to use the application. Another type of tablet with a different operating system is not possible, because GuideaMaps was created and designed for iOS only. If someone doesn't have access to an iPad, he cannot use the application, which is a hard restriction.

Initially, the tool was created for the purpose of requirement elicitation for serious games, but it can also be used for the requirement elicitation in other domains (De Troyer & Janssens, 2014). However, because the visualizations are based on pre-defined templates, the nodes can only be edited by the end-user in a limited way: content can be added and the background color can be changed, but the end-user cannot add new nodes. In addition, the visual notation used is fixed: the creators of the visualization templates cannot edit the representation of a node or define their own representation (e.g. change the length and width or use a different shape). Not being able to do this is a limitation in the sense that for some purposes this default visualization may not be very suitable. Furthermore, for defining a template the author had to use XML and no graphical editor was available for this purpose making it harder to define new templates for non-ICT schooled people.

1.2 Research Goals

The problem statement discussed in the previous section indicates that the first version of GuideaMaps comes along with some limitations. Therefore, the following research goals for a new version of GuideaMaps were formulated:

Goal 1
The new version of GuideaMaps should work on all common devices and on different operating systems.

Goal 2
It should be possible to pre-define the maps, i.e. the templates, in a graphical way.

Goal 3
The new version of GuideaMaps should allow the end-user to extend and modify the pre-defined map in some restricted way.

Goal 4
The application should be generic in such a way that it can be customized to be usable for different purposes, so that the user can define its own graphical representation for the visualization.

This thesis presents a solution, called *GuidaMaps 2.0*, for the problem statement with the research goals taken into account. How the tool achieved the research goals formulated is explained into detail in the rest of this thesis.

2
Related Work

There exist lots of ways to visualize data and the relations between data. In this chapter, we discuss some of the possibilities related to what we need for GuideaMaps.

2.1 Mind Maps

The most well-known technique to visualize related data is to create a mind map (a.k.a. idea map). This technique is mainly used to show the relation between portions of information and for brainstorming purposes. Other applications where this technique is used are note-taking, problem solving, etc. (Balaid et al., 2016)

Mind maps are created by writing the main idea in the middle of the drawing, while all sub-ideas are placed around that center node. Each sub-idea is connected with its parent by means of a line. Hence, this kind of visualization is not difficult to create or understand. Its simplicity is one of the reasons why it is used a lot in practice.

Because mind maps is not a new concept, but one that most people already know quite well, we will not discuss it into further detail.

2.2 Knowledge Maps

According to Balaid et al. (2016), *knowledge maps* is an umbrella term for tools and techniques like mind maps. O'Donnell et al. (2002) defined the concept as follows:

> Knowledge maps are node-link representations in which ideas are located in nodes and connected to other related ideas through a series of labeled links.

This way of representing information has for example a positive impact on students. The paper of O'Donnell et al. (2002) teaches us that students using knowledge maps are better in remembering the main ideas of the subject in comparison to the ones that study from the text without the visualization. As GuideaMaps's context is more focused on representing large amounts of knowledge in an easy to grasp way, it is less important that the users remember the content of the visualization, but the node-link representation with the main idea in the center also showed to be useful for this purpose, as illustrated by the popularity of mind maps.

Next to mind maps, concept maps is a second technique included under the umbrella of knowledge maps. Concept maps are similar to mind maps in some sense but they do have some different characteristics. First, the purpose of a mind map is to associate ideas, topics or things, while concept maps illustrate relations between concepts. Further, the structure of a concept map is mostly hierarchical and visualized like a tree. On the other hand, mind maps sometimes have a radial layout and not hierarchical. (Davies, 2010)

Hence, we can state that GuideaMaps makes use of a knowledge map visualization and more specifically some kind of combination of mind maps and concept maps.

3 Requirements

The application should meet a lot of requirements in order to deliver some quality to the users. End users of a system often qualify a system as *usable* if it provides the right kind of functionality, if it is easy to learn and easy to use. In the bachelor course *User Interfaces* two definitions of usability were provided:

Definition 1
Usability is a measure of the ease with which a system can be learned and used, its safety, effectiveness and efficiency, and attitude of its users towards it. (Preece et al., 1994)

Definition 2
Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. (ISO, 1998)

Hence, there are functional and usability requirements the system should meet in order to be evaluated as *usable* by the end users.

3.1 Functional Requirements

1. **Customization**
   End users should be able to customize the visualization a bit to their needs. It should be possible to change the background color of nodes, to add child nodes and to remove nodes they don't need anymore. But end users should only be able to delete nodes they added themselves and not the nodes initialized by the map creator. Otherwise it would be possible for end users to change the complete visualization.

2. **Modes And Rights**
   There should be two possible modes in the system: an end user mode and a map creator mode. A map creator has the rights to change the structure of the visualization, while the end user is restricted in which data he can edit. These requirements can be seen as a security mechanism to not let a particular end user mess with the visualization.

3. **Zoom the visualization**
   As a user, you sometimes want to zoom in or out such that you get less or more information at the same time on the screen. For example, a feature to zoom is very useful in situations where you want to compare different parts of the visualization. The scrolling gesture is probably the best gesture for this action, because this is a well-known way to zoom in applications (e.g. Google Maps). Users don't like it when every application has a different gesture for the same action. Hence, it won't help them if we choose for another gesture than scrolling in GuideaMaps. Also, using the same gesture as in other applications improves the memorability and learnability of our tool.

4. **Zoom to fit**
   A variation on the zooming feature is zoom to fit (a.k.a. zoom to bounding box). With custom zooming, the user can set the zooming level to meet its needs. On the other hand, zoom to fit adapts the zooming level and moves the content of the application until everything fits on the screen.

5. **Genericity**
   The functional requirements defined until here are specific for GuideaMaps. We want our tool to be useful in many other cases than GuideaMaps. Therefore, some requirements concerning the genericity of the application can be expressed.

   (a) The tool should be generic in such a way that it is possible to create a different design for the nodes and the links.

   (b) The end user should not be restricted to the predefined design of GuideaMaps.

   (c) A developer can create its own implementation for the nodes and the links, which then can be *plugged in* into the system.

   (d) The core of the application should be completely separated from the parts that are customizable for the end-users.

3.2 Usability Requirements

Next to the functional requirements, the application should meet a lot of other requirements, e.g. in terms of usability.

In order to obtain a high usability, we should formulate several usability requirements for the application. Later in this thesis, when we explain the

implementation details, we will come back to these requirements and discuss how we managed to meet them.

1. **Intuitiveness**
   It is important to keep in mind that the tool should not only be used by people with experience in Computer Science. It doesn't matter whether or not the user has a background in Computer Science, he should be able to easily learn to use the system in short time. Therefore, some design choices are crucial for the level of usability.

   (a) It is important to choose for clear, not misunderstandable icons on buttons where a click on this button invokes a certain action. Some examples of these actions are (1) adding a child node, (2) explore and edit a node and (3) expand/collapse a node. Good icons[1] for each of the mentioned actions are shown in figure 3.1. The icons are not ambiguous, they can only be linked to one particular action and thus the user knows exactly what to expect when clicking on the button. Well chosen icons are one of the factors in the design that help users to remember how to use the system. If they recall the meaning of the icon immediately when they see it, the users will experience the system as easy to learn and easy to remember.

   If the user knows it is not possible to add child nodes, we would state that a plus- and minus-icon are also possible to indicate the possibility to expand or collapse a node. If the visualization allows to add child nodes, the plus-icon should be used for this action and not for the expand- or collapse-action.

(a) Plus (b) Explore/Edit (c) Expand (d) Collapse

Figure 3.1: Good icons for the following actions: (a) add child node, (b) explore and edit node, (c) expand node and (d) collapse node.

   (b) As already mentioned in the functional requirement 3, gestures for trivial actions should not differ from the gesture for the same action in other applications.

2. **Learnability**
   Intuitiveness and learnability are somehow related to each other. It is easier to learn how a system works if the possibilities are intuitive and actions are not hidden.

3. **Efficiency**
   As a user, it should be easy to achieve your goals. The application should be created in such a way that user errors are (almost) impossible.

[1] `https://fontawesome.com/`

4. **Accessibility**
   The target audience of the application should be limited as little as possible.

   (a) We don't expect the user to have a background in Computer Science or visualization techniques.

   (b) Color blind people should still be able to use the system. We can make sure this is possible by allowing map creators to choose for colors that are distinguishable by color blind people.

3.3 Other Requirements

1. **Device- and OS-independent**
   The application should run on different kinds of devices (e.g. tablets, laptops and desktops) and operating systems (Android, iOS, MacOS and Windows). While version 1.0 was only designed for an iPad, our version should provide a solution for this restriction. One of the possibilities to solve the problem is to create the application in the browser. Hence, the user should not be limited to a particular device; the tool can be used on any machine. The only restriction on the used device is that it needs to have a screen that is large enough because it is not very convenient to work with the visualization on small screen areas, e.g. on smartphones. The application could run on smartphones but it is not recommended nor required to use it on devices with relatively small screens.

4 Implementation

The previous sections explained *what* the application should do, which goals should be achieved and which requirements we surely want the application meets. In this section, more details are provided about *how* all this is translated to the implementation.

4.1 Design Choice

Before we started with the actual implementation, we had to decide how to achieve Research Goal 1, i.e. how will we make sure the application is devide- and OS-independent? A first technique could have been to make use of the Java Virtual Machine. This approach would work but at least one big problem comes along with it. We want to make it possible to let multiple people work on the same visualization, eventually at the same time. Therefore, it is more convenient to have a browser-based application.

Nowadays, a lot of browsers exist, each with its own characteristics and differences. Because the focus of this thesis was not to have the application perfectly working on *all* browsers, we made sure that the tool works without any issues in Google Chrome. We chose for Google Chrome because this is one of the most widely used browsers in the world.

4.2 Technologies

A browser is device- and OS-independent and lots of technologies exist to create applications and visualizations for the web. A number of technologies assisted in the developing process of the application. The used technologies are browser-based

because we chose for a solution optimized to work in the browser. In this section, each of these technologies is discussed to explain its importance.

4.2.1 ReactJS

ReactJS is an open source library to create user interfaces[1]. One of its main goals is to provide the best possible rendering performances[2]. Performance is high because ReactJS allows developers to break down the user interface into different components. Each component has its own *state*, which contains information about the content of the component. This state can be updated while the application is running and if such an update is made, only this component is re-rendered instead of re-rendering the complete UI[3]. Hence, this involves a huge benefit for the performance. Next to that, it is also not very hard to learn to code in React when comparing to other frameworks (e.g. AngularJS). If the developer knows HTML and JavaScript, he will be able to code in ReactJS quickly.

Because of these benefits, ReactJS is the framework in which the GuideaMaps application is implemented. Each node and each link is considered as a separate and unique React Component. The most important reason for implementing the nodes like this is performance: if the state of the node is updated, only this node is re-rendered and not the complete UI.

Alternatives to ReactJS are for example AngularJS and VueJS[4]. ReactJS is by far the most used of these three. Further, it is said to be easier to learn because you only have one structure ("Component") to remember, while in AngularJS this is not the case. Also, VueJS lacks resources and is not used a lot in practice, which makes it more difficult to discuss problems with other developers. Maybe the most important benefit of ReactJS is its performance; React is really fast and our tool should have fast rendering times as well. Hence, because the alternatives seem to have some important drawbacks, we chose for ReactJS as the framework in which we created the application.

4.2.2 d3

Another helpful tool is d3. With d3-hierarchy[5], it is possible to transform JSON-data into hierarchical data. Having this kind of data makes it much easier to create a tree- or cluster-structure. In the case of GuideaMaps, a clustered visualization is very useful. The *main*-node (a.k.a. the root node) is then positioned at the center, such that its child nodes can be placed around it. Hence, the farther a node is away from the center, the lower it is in the hierarchy. Also, the visualization will not be messed up by positioning the nodes in this way,

[1] `https://reactjs.org/index.html`

[2] `https://medium.com/@thinkwik/why-reactjs-is-gaining-so-much-popularity-these-days-c3aa686ec0b3`

[3] `https://facebook.github.io/react/docs/why-react.html`

[4] https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d

[5] `https://github.com/d3/d3-hierarchy`

because every node has exactly one parent. This means there won't be a spaghetti of links where you cannot see which node the link comes from and which node it is pointing to.

Further, d3 makes it easier to create a zoomable layout. The tool is able to update the positions of the nodes each time the zooming level is changed. Hence, as a developer, you don't have to take care of the updated positions if you make use of d3. By using this technique, the functional requirements about zooming (section 3.1) are achieved.

The alternatives for d3 are most of the time they come along with a problem d3 does not have. For example ChartJS and ChartistJS are limited in the number of features: you can create nice charts with it, but with d3 provides more functionality than only charts. Further, some of the alternatives are commercial (Highcharts, Webix). d3 is open-source and provides functionality to zoom your visualization, to create a cluster of nodes based on JSON-data describing these nodes, etc. Because of the wide range of possibilities with d3, the fact that it is widely used and supported by all modern browsers and that it is able to act together with ReactJS, we believe that d3 was a better choice than its alternatives.

4.2.3 Tailwind CSS

The style of the application is very important for the end user. Everything should look pretty and, as already mentioned in section 3.2, the possibilities should be straightforward and visible. While developing and creating a beautiful style for applications and websites, the code for these styles can become a big part of the implementation. Hence, a good framework is necessary to reduce the lines of style code to a reasonable number. It also helps to improve the readability of the code.

Tailwind CSS[6] is a framework that assists developers to style their application. The difference with more famous frameworks, like Bootstrap, is that Tailwind CSS has no default theme. If you want to use a Bootstrap-feature, this eventually comes along with other features you don't always wanted and it can be quite hard to undo the part you didn't ask for. Furthermore, you have to write additional lines of style code to undo the unwanted parts. With tailwind on the other hand, you can grab only the features you want, without side-effects. Figure 4.1 shows an example with two small listings. The first uses inline style while the second makes use of tailwind CSS.

[6] `https://tailwindcss.com/`

```
<div
  style={{
    position: absolute,
    border: 1px solid black,
    borderRadius: 0.25rem,
    padding:: 0.5rem,
  }}
/>
```

Listing 4.1: Normal CSS, no tailwind.

```
<div
    className={
      'absolute border
      ↪ border-solid
      ↪ border-black rounded
      ↪ p-2'
    }
/>
```

Listing 4.2: With tailwind CSS.

Figure 4.1: Difference when using tailwind CSS or not.

The figure illustrates the difference to implement four CSS property-value pairs in normal CSS and implementing the same four with tailwind. In the case of normal CSS, we need four lines of code to retrieve the intended result. On the other hand, with tailwind CSS, we add some classes providing the same style. The classnames can be placed on a single line instead of four. This example proves that the number of lines can be decreased.

Bootstrap can be very interesting to use in applications and websites that should run on devices with small screens (e.g. smartphones). But we already decided to not optimize our application for such devices (3.3) and hence Bootstrap is probably not the best framework to use in this case. Given the example and the fact that Tailwind CSS does not have a default theme, we prefer Tailwind over Bootstrap. Keep in mind that it is certainly possible to create the same application with Bootstrap, but with Tailwind CSS, the code is easier to understand.

4.3 Main Code Structure

With the technologies mentioned in the previous section, the most important pillars the application relies on are discussed. In this section, we will explain the main structure of the code, such that it is clear how all elements work together. Figure 4.2 shows a visualization of the structure of the code.

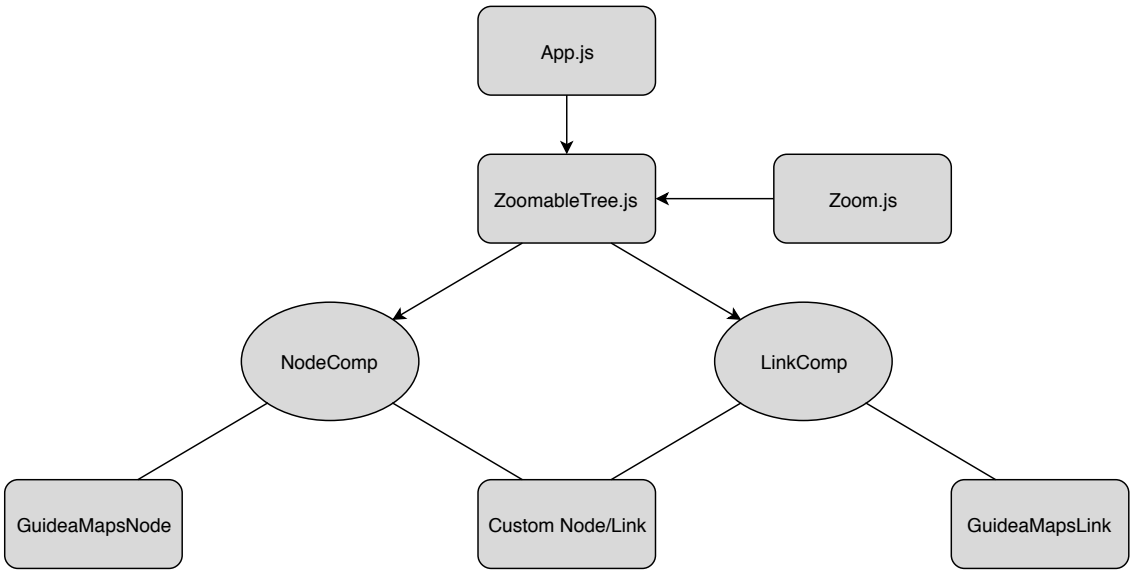

Figure 4.2: Structure of the application.

The application is developed in such a way that it can be used as a library for other purposes than GuideaMaps as well. Hence, the research goal to have a generic solution is achieved. The general, always-returning part of the code can be found in App.js, ZoomableTree.js and Zoom.js, where the layout of the nodes is defined as well as the implementation to allow the user to zoom the visualization in and out. When talking about the library, the layer of the NodeComp and LinkComp is very interesting and important. For GuideaMaps (GM), an implementation for a GMNode and GMLink is provided. Each implementation describes what every node and every link should look like in the visualization.

The strength of the library is its genericity, i.e. when a particular user would like to have a different representation for the nodes or the links or both. In that case, new components (e.g. MyCustomNode and MyCustomLink) should be implemented. In the code, only one line should be adapted: in App.js, ZoomableTree is called with a certain number of props. Two of these props are NodeComp and LinkComp, which are set to GMNode and GMLink, respectively, by default. Hence, the only action that is required to *plug in* an other component is replacing GMNode by MyCustomNode and GMLink by MyCustomLink. Hence, the visualization can be customized to the needs of the user. Figure 4.3 shows the part of the code in App.js that should be adapted as explained. Note that the shown props are not the only props that are passed to ZoomableTree. The others are omitted because they should not be changed by the user and to improve the readability.

```
<ZoomableTree
    NodeComp={GMNode}
    LinkComp={GMLink}
/>
```

Listing 4.3: Default components.

```
<ZoomableTree
    NodeComp={MyCustomNode}
    LinkComp={MyCustomLink}
/>
```

Listing 4.4: Custom components.

Figure 4.3: Two listings showing how to use the library.

The reason why we chose for this approach is that now the user does not have to change anything of the default implementation. You only have to create your own components and plug them in as props for ZoomableTree and leave the implementation for GuideaMaps as it is.

Next to custom components, you can also implement your own functions to handle changes in the visualization. For example, to add a new child node, GuideaMaps calls the function passed to the *onAddNode*-prop (i.e. addGMChildNode). In a custom implementation, you can pass another function to this prop to make sure that other work is done than in the default implementation. If you don't want to allow users to add child nodes, you don't have to remove this line (because the library is created in a way to avoid changes in the default code structure) but you just replace addGMChildNode by *() -> null*, a function that does not do anything.

4.4 Default Implementation: GuideaMaps

Now the overview of the structure of the application is discussed, we will consider the GuideaMaps visualization and its implementation details in this section. The resulting visualization can be seen in figure 4.4.

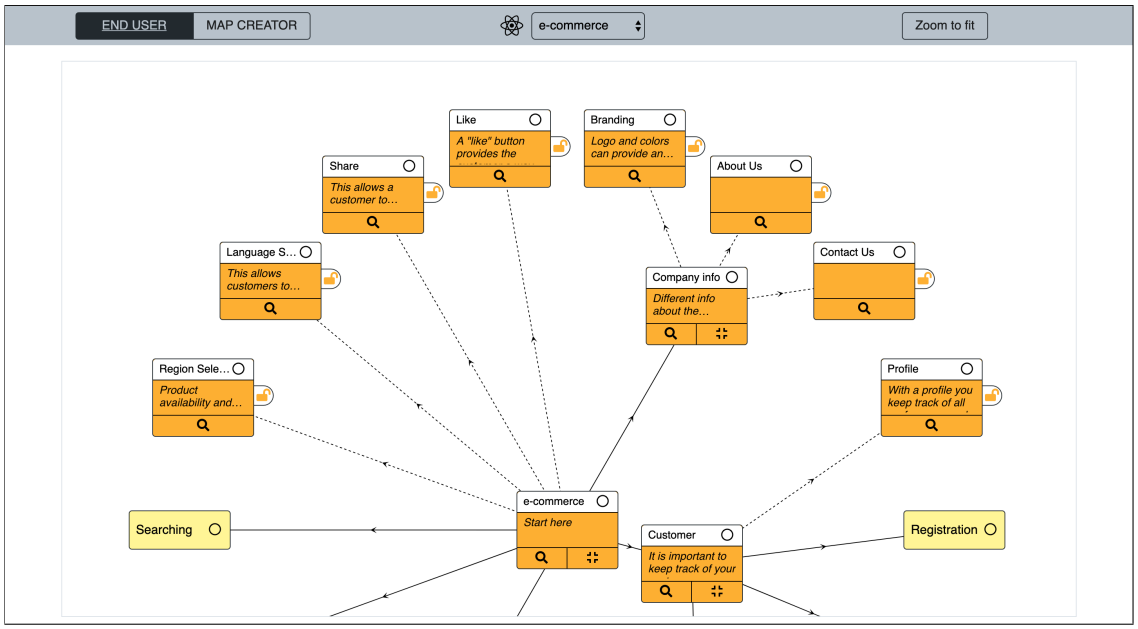

Figure 4.4: GuideaMaps Layout.

The nodes represent a specific part of the data and the links illustrate the relation between the data (i.e. the nodes). Before we discuss the nodes into detail, we start with the *navigation bar* above the visualization itself. This navigation bar is divided in three equal parts. In the center, the user can select which visualization he wants to use via an option menu. The name of the currently selected option is always visible. There are several reasons why this is possible. The first is because the user should be able to switch between templates. A user should not be limited to one possible visualization. As a map creator, you can create a new template, which then can be filled with data by the end users. As an end user, you should be able to switch between different visualizations because it is possible that you are working on more than one visualization at the same time.

The right part of the navigation bar contains a single button. A click on this button makes sure the visualization is zoomed to a certain level where all nodes fit on the screen. This is one of the reasons why we mentioned in the requirements (section 3.3) that the screen of the used device should not be too small.

The left part is created to be able to switch between the user modes (i.e. end user or map creator). A map creator has more rights and hence he can perform more actions than an end user. In the following subsections, the differences between these modes is elaborated.

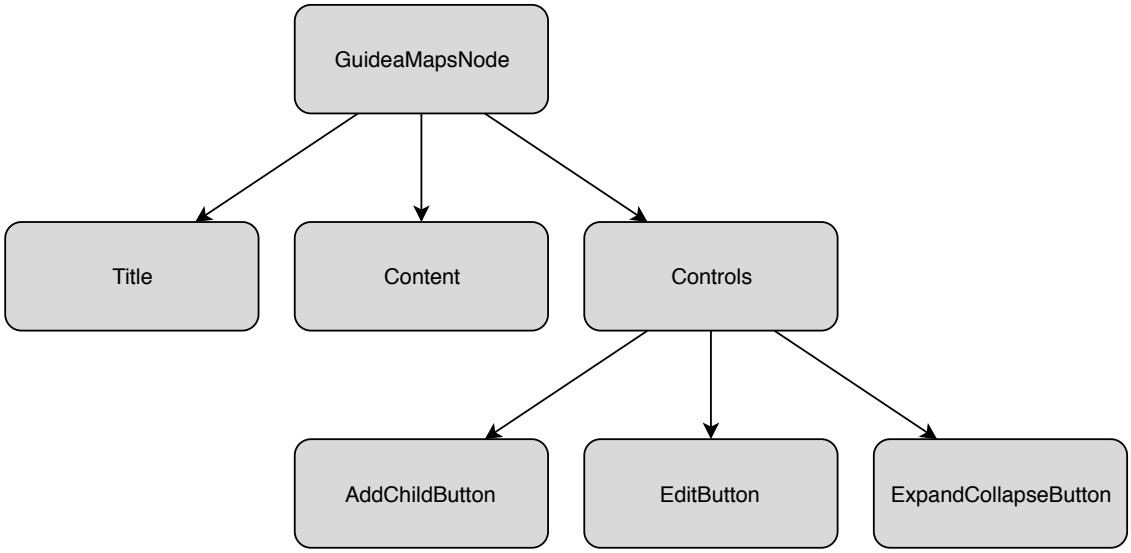

Figure 4.5: Structure of GuideaMapsNode.

As you can see in figure 4.5, the structure of a GuideaMapsNode node is quite easy. Each node consists of three html *div*-elements: a title-div, a content-div and a controls-div.

4.4.1 Title

The title-div is positioned at the top of the node. It consists of the title text when this is given by the map creator. Otherwise, it shows *Insert title* in italics to remind the user that he still had to set a title for this node. An end user can set the title if no title exists yet, but he can never change it. The reason for this is because we distinguish two situations. In the first situation, the end user adds a child node for a particular parent. This node has not title by default and then he should be able to insert a title. In the second situation, the user opens an existing node created by the map creator. In that case an end user is not allowed to change the title of the node, because otherwise he would be able to change the complete layout of the visualization.

Next to the title-text, the title-div contains an icon placed near the right border. This icon indicates whether all information, i.e. some content, is provided or not. The icon that is shown also depends on whether the information in its child-nodes is filled in or not. We distinguish three possible situations:

1. The data of the node itself and of all of its children are correctly filled in. In this case the icon will be a completely filled circle.

2. The data of the node itself and of all of its children are all empty. In this case the icon will be an empty circle.

3. In all other cases, the data of the node itself or of at least one of its nodes is filled in. Then the icon will be a semi-filled circle.

This icon can assist the user in determining whether all information is filled in. For example, if he checks the root node and sees that the circle is completely filled, he does not have to check every child-node to know that all information is filled in. On the other hand, suppose only one node is not yet provided of some content. If the user then starts from the root node and always follows the child-node with a semi-filled circle, he will find the incomplete node much faster than in the case he has to check all the nodes.

4.4.2 Content

The content of each node is just some text describing the information corresponding to the title. As long as no content is provided by the user, a description is shown in italics to instruct the user which content to provide.

4.4.3 Controls

The last part of a node is the controls-div. With *controls* we mean the different actions that a user can take concerning the particular node. The number of actions a user can perform depends on the mode. An end user has two buttons: one to "open" the node to view and edit the data and one to expand or collapse the node to show or hide the child nodes, respectively. When a node is collapsed, all child nodes on all lower levels in the hierarchy are hidden. On the other hand, when a node is expanded, only the child-nodes of the next level in the hierarchy are shown.

Because a map creator is allowed to add child nodes, a third button can be found in its controls-div of the node. A click on this button opens a small modal, where some information about the new node should be provided before the node can be created. Figure 4.6 illustrates what the modal looks like when the user wants to add a default node. He has to start by selecting the option "Default" at the top, after which two input fields appear to insert a title and a description for the node. A click on the button at the bottom will add a child node of the selected type and with the provided title and description.

It is possible to add an *empty* node if the user does not provide a title and/or a description. Even though it is not really useful to add nodes without any context, we only require the user to select the type of the child node. If the type is unknown, the node will not be added. An example case where a map creator would add an empty node is when he wants to remind himself a child node is necessary at that place, but currently he does not know exactly what data the node should contain.

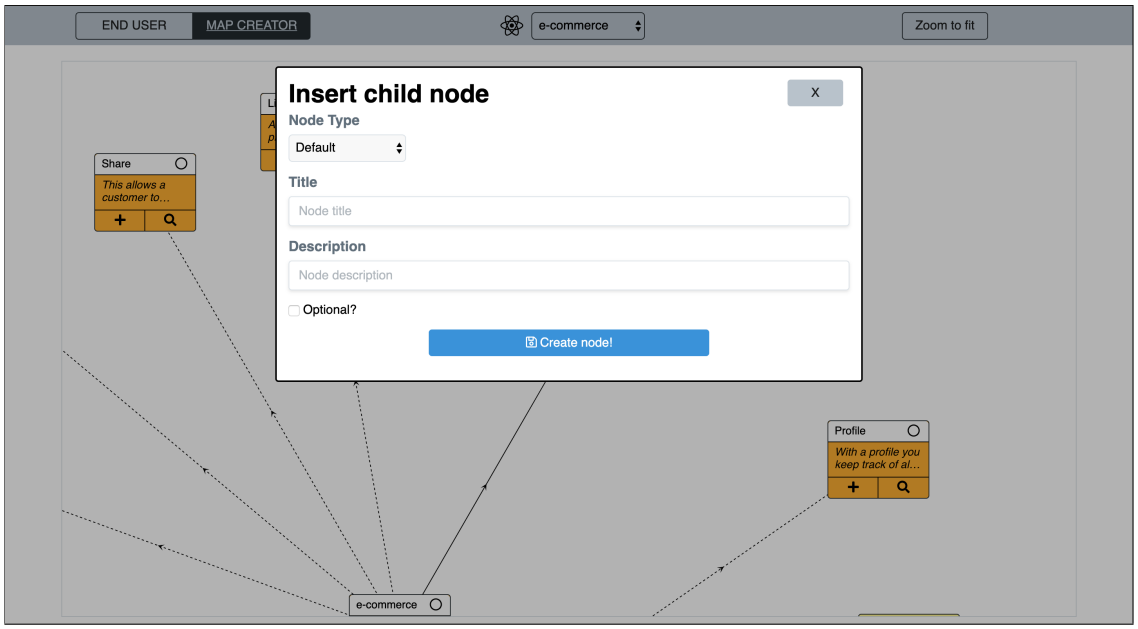

Figure 4.6: The edit modal in map creator mode.

TODO: describe addition of choice nodes

4.4.4 EditModal

A click on the button to open the node opens a modal representing the data. What this modal looks like depends on the mode. If we are in map creator mode, a form will be shown to allow to change the title, the description, the content itself and the background color (Figure 4.7). In the case of the end user mode (Figure 4.8), only the content and the background color can be changed. Only the first time a child node is added and no title and description is filled in yet, the end user is able to fill in these data as well.

When changing the background color, the user can choose to include all children or not. If the checkbox "include children" is checked, the background color of all child-nodes on all sublevels in the hierarchy will be changed to the new color. Otherwise, only the background color of the current node will be changed.

The text color is black by default. This can become a problem when the background color is changed to a dark color and certainly when it is black as well because then the text is not readable anymore. To solve this issue, we wrote some CSS rules that change the text-color depending on the background color. A black (or dark) background will result in white text and a white (or light) background results in black text.

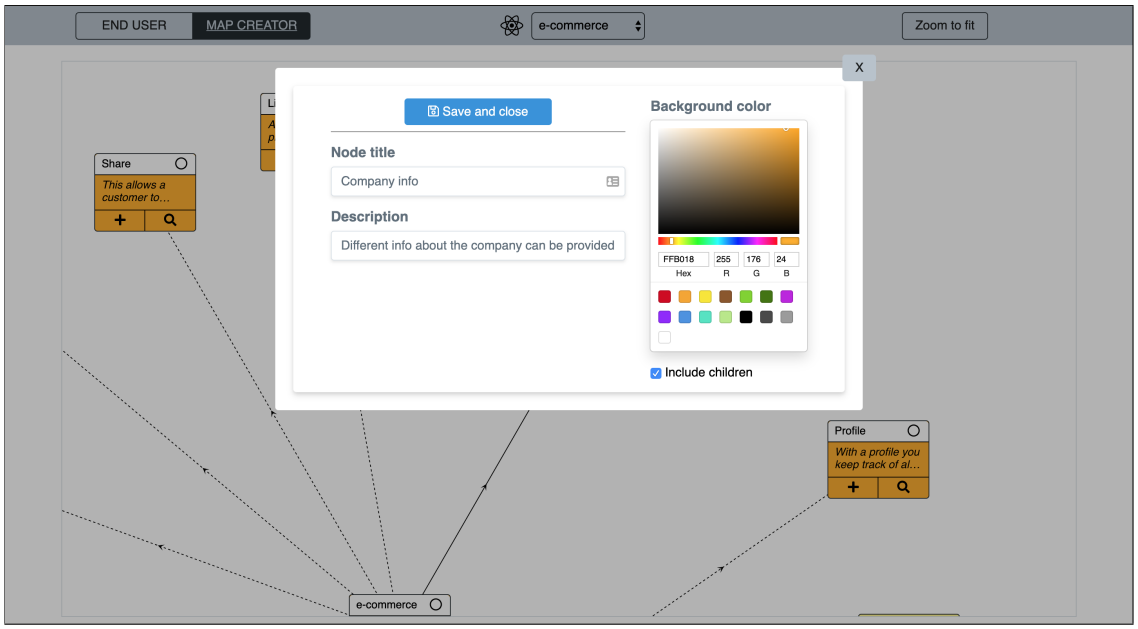

Figure 4.7: The edit modal in map creator mode.

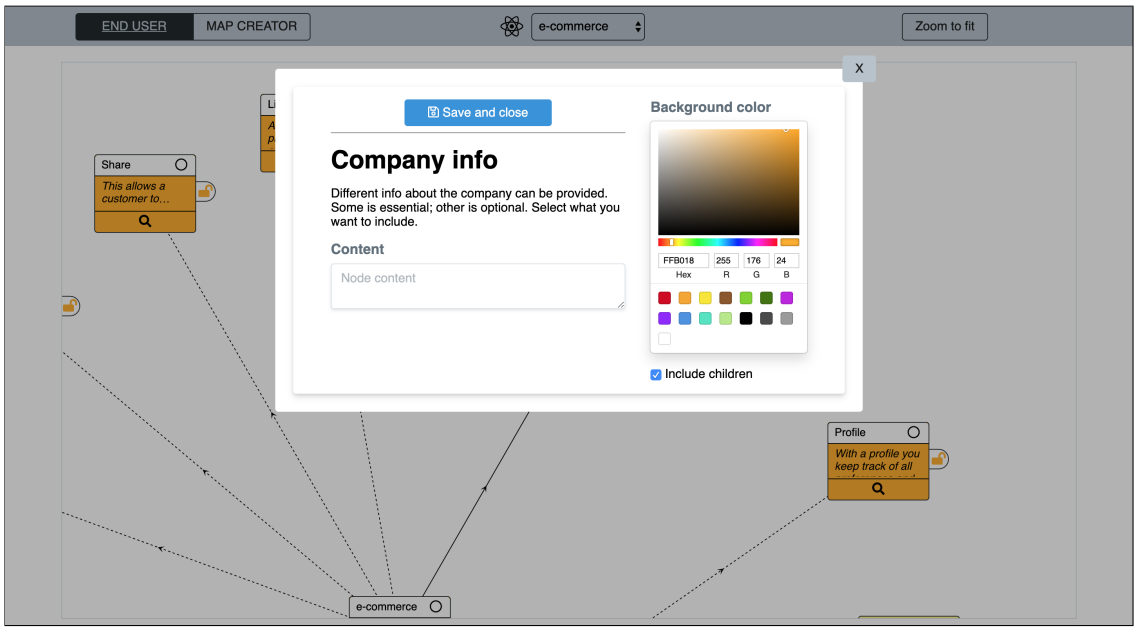

Figure 4.8: The edit modal in end user mode.

4.4.5 Optional nodes

The visualization also provides optional nodes. They can be recognized by the link between this node and its parent. A regular node is connected to its parent via a solid link, while an optional node is connected via a dashed link. A special characteristic is that these nodes can be disabled by the end user. Optional nodes contain an additional button at their right side with a lock. As long as the node is enabled, the button is represented by an open lock. A click on this button disables

the node and changes the icon to a closed lock. A second click re-enables the node. Next to the lock icon, a disabled node can be recognized by its gray background and a blur so that you cannot see the content anymore.

5 Use case

To illustrate the simplicity of the library we implemented a use case, called *Plateforme DD*. The goal was to visualize the links between the content of the website[1]. The visualization and functionality is completely different than the one for GuideaMaps and can be seen in figure 5.1.

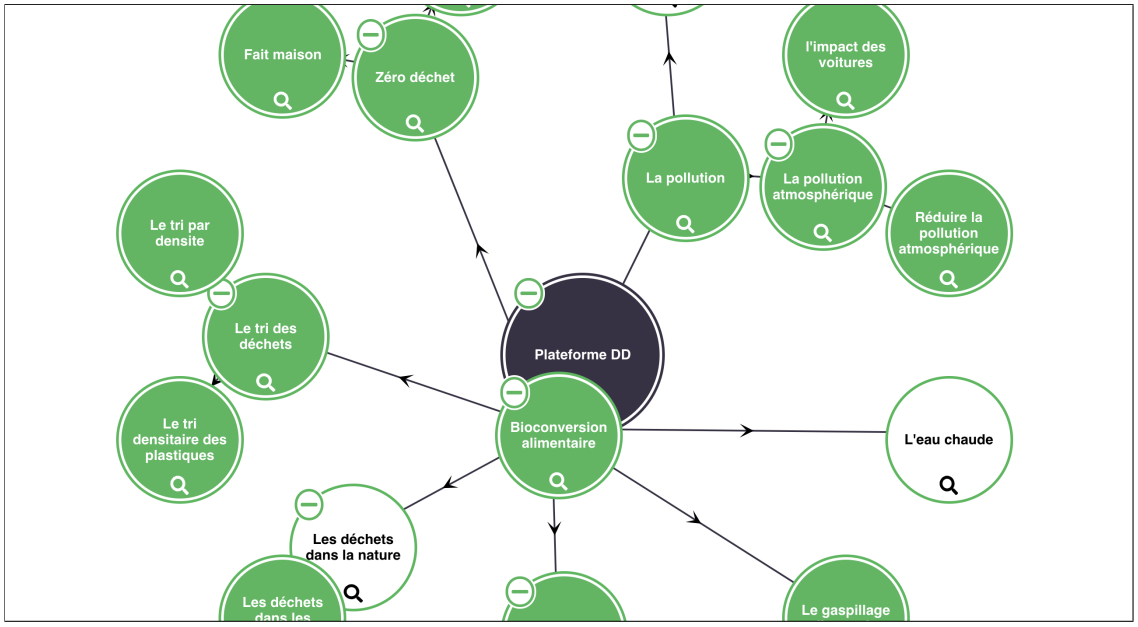

Figure 5.1: Plateforme DD Layout.

[1]https://sciences.brussels/dd/

5.1 Configuration

In Figure 4.3, we showed how we could alter the visualization of the data without affecting the default code. Remember that the props in that figure are not the only customizable props of the library. Figure 5.2 shows the differences in configuration when we compare the visualization of GuideaMaps with the one of Plateforme DD.

```
<ZoomableTree
    NodeComp={GMNode}
    LinkComp={GMLink}
    EditModalComp={
    ↪ GMEditModal}
    onAddNode={addGMNode}
    onDeleteNode={
    ↪ deleteGMNode}
    onNodeUpdate={args ->
    ↪ updateGMNode(args)}
    onVisibleChildrenUpdate={
    ↪ nodeId =>
    ↪ updateGMVisibleChildren(
    ↪ nodeId)}
/>
```

Listing 5.1: GM Configuration.

```
<ZoomableTree
    NodeComp={PDDNode}
    LinkComp={PDDLink}
    EditModalComp={
    ↪ PDDEditModal}
    onAddNode={() -> null}
    onDeleteNode={
    ↪ deleteGMNode}
    onNodeUpdate={args ->
    ↪ updateGMNode(args)}
    onVisibleChildrenUpdate={
    ↪ nodeId =>
    ↪ updateGMVisibleChildren(
    ↪ nodeId)}
/>
```

Listing 5.2: PDD Configuration.

Figure 5.2: The configuration differences between GuideaMaps and Plateforme DD.

5.1.1 NodeComp

The most obvious difference, which can immediately be seen when comparing both visualizations, is the layout of the nodes. In GuideaMaps (GM), we had rectangular nodes, while in Plateforme DD (PDD) the nodes are circular. To achieve this result, we implemented a special component, called *PDDNode*, and plugged it in the library, i.e. we replaced *GMNode* by *PDDNode*. From then on, every node in the data will be visualized by the code of PDDNode instead of the code of GMNode. Hence, GMNode still exists; it is not deleted or overwritten, but simply not used in this configuration.

5.1.2 LinkComp

The second prop is responsible for the representation of the links. You won't observe big differences between a GMLink and a PDDLink, except for the thickness: a PDDLink is thicker than a GMNode. As a consequence, the implementation of PDDLink is a copy of the GMLink with the value for *strokeWidth* as only difference.

If the user does not need a different style for the links, he does not have to copy the implementation of GMLink and create a PDDLink component. In that case, it is possible to use GMLink as LinkComp, while the other props are eventually specially created for Plateforme DD.

5.1.3 EditModalComp

The edit modal of Plateforme DD is completely different in comparison to the one of GuideaMaps. First of all, it is not a real *edit*-modal because a user of the PDD visualization is not allowed to update the data of the nodes. Hence, he can only view the content of the nodes, which is shown like in Figure 5.3.

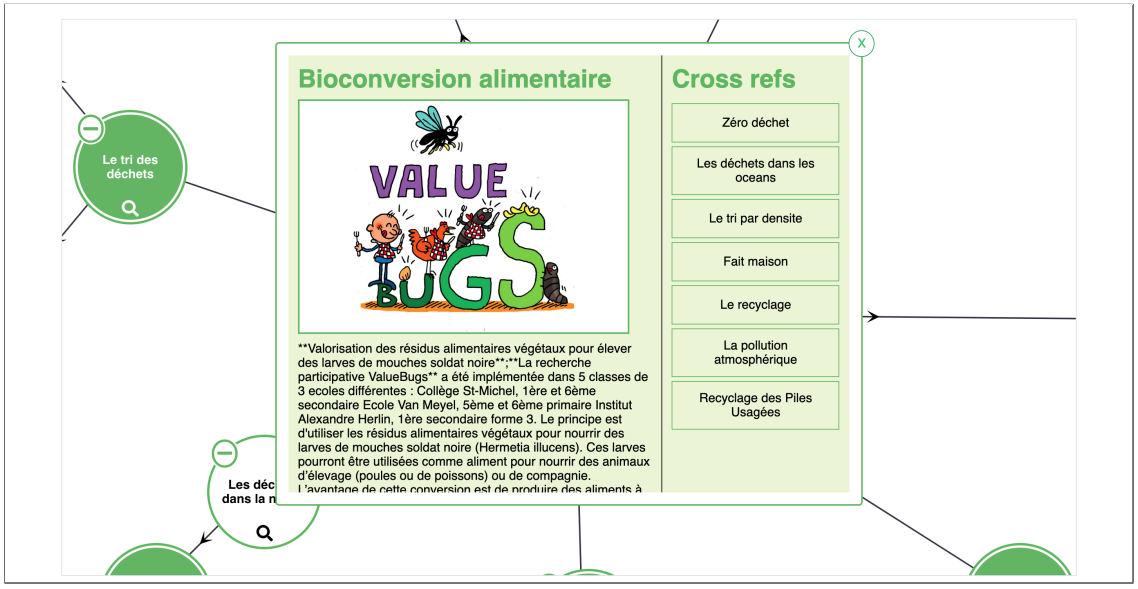

Figure 5.3: Plateforme DD Edit Modal.

At the left part of the edit modal, the actual content is shown. It starts with the title of the node, followed by the picture corresponding to it. If no such picture exists, the text providing more information about the topic of the node is shown immediately under the title. Otherwise the title and the text are separated by the accompanying picture.

The size of the edit modal is fixed, but in case the information does not fit the modal, the user can scroll in the left part to continue reading until the end. The modal will not grow in size to make sure all data fits in it.

For some nodes, the edit modal will contain a list of cross references of the node. Cross references are links from a certain node n to another node, which is not a direct child of n. Such a link is not shown in the visualization, but by providing such a list, the user can still discover the cross reference via the modal. A click on a cross reference will close the modal and *move* the visualization until the node corresponding cross reference is centered. This allows the user to navigate to linked nodes without the need to create a spaghetti of links in the visualization.

5.1.4 Action Listeners

In GuideaMaps, we have a controls-div (section 4.4.3) containing a number of buttons to allow the user to perform some actions. On the other hand, in Plateforme DD, we chose for another approach. Because we have circular nodes instead of rectangular, it is less convenient to position multiple buttons next to each other at the bottom of the node. Further, we don't need a button to add child nodes because this is not allowed in Plateforme DD. Hence, we only have to buttons: one to expand and collapse the child nodes and one to open the edit modal.

The button to open the node is positioned in the center at the bottom of the node. A click on that button opens the modal and centers the node such that the node is in the middle of the screen when the user closes the modal. Specific to this implementation is that there is a second way to open the modal. If the user clicks once on the node, it is centered. If he clicks once more on the node (not necessarily on the button), the modal will open and the content will be shown. This is because a click on a centered node results in the same action as a click on the button to open the node.

The second button is one to expand or collapse the child nodes. As already said, we cannot position it next to the other button at the bottom of the node. Therefore we position it at the top-left of the boundary of the node. There, a circle is visible with a minus inside if the children are visible. A click on the minus will collapse the node such that the children are not visible anymore and replace the minus by a plus. When the user clicks on that plus again, the exact opposite action will happen: the node will be expanded and hence the children of the node will be visible again and the plus is replaced by a minus.

Note that a plus and minus can be used here as symbols to expand and collapse a node. This is possible because there can be no confusion with the action of adding a child node in Plateforme DD. If this would be possible, other symbols should be used to avoid such confusion and to not reduce the intuitiveness (Usability Requirement 1).

6
Evaluation

7 Conclusion & Future Work

7.1 Conclusion

7.2 Future Work

Describe future work here.

References

Balaid, A., Rozan, M. Z. A., Hikmi, S. N., & Memon, J. (2016). *Knowledge maps: A systematic literature review and directions for future research.*

Davies, M. (2010). *Concept mapping, mind mapping and argument mapping: what are the differences and do they matter?*

De Troyer, O. (2016). *Computer Science, Lecture Notes: User Interfaces.* Vrije Universiteit Brussel.

De Troyer, O., & Janssens, E. (2014, August). A feature modeling approach for domain-specific requirement elicitation. In *Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on* (p. 17-24). IEEE. doi: 10.1109/RePa.2014.6894839

ISO. (1998). *ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability.* Author. Retrieved from `https://books.google.be/books?id=TzXYZwEACAAJ`

Janssens, E. (2013). *Supporting requirement elicitation for serious games using a guided ideation tool on ipad.*

O'Donnell, A. M., Dansereau, D. F., & Hall, R. H. (2002). *Knowledge maps as scaffolds for cognitive processing.*

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., & Carey, T. (1994). *Human-computer interaction.* Essex, UK, UK: Addison-Wesley Longman Ltd.