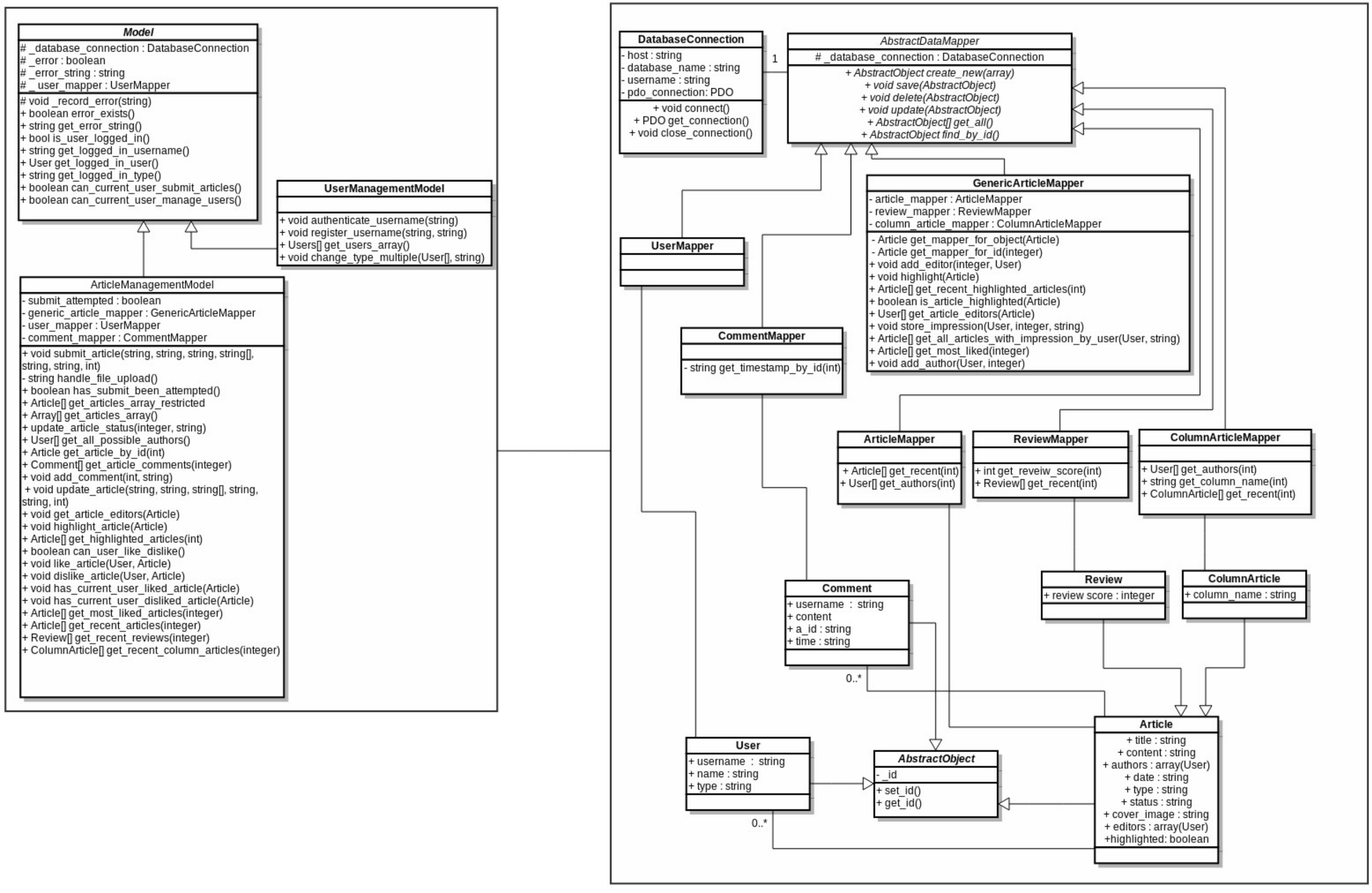
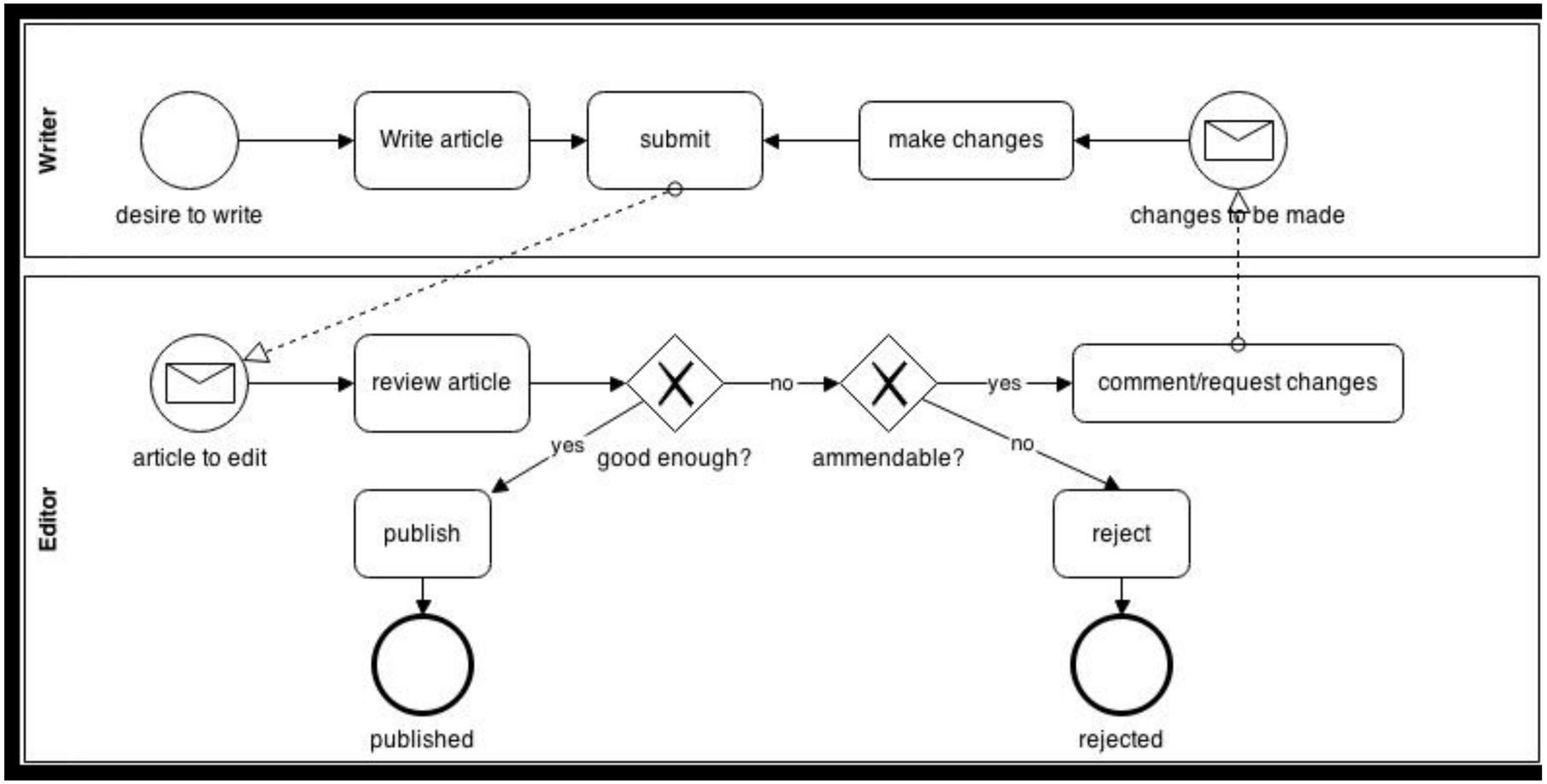

Model
# _database_connection : DatabaseConnection
# _error : boolean
# _error_string : string
# _user_mapper : UserMapper
# void _record_error(string)
+ boolean error_exists()
+ string get_error_string()
+ bool is_user_logged_in()
+ string get_logged_in_username()
+ User get_logged_in_user()
+ string get_logged_in_type()
+ boolean can_current_user_submit_articles()
+ boolean can_current_user_manage_users()
UserManagementModel
+ void authenticate_username(string)
+ void register_username(string, string)
+ Users[] get_users_array()
+ void change_type_multiple(User[], string)
ArticleManagementModel
- submit_attempted : boolean
- generic_article_mapper : GenericArticleMapper
- user_mapper : UserMapper
- comment_mapper : CommentMapper
+ void submit_article(string, string, string, string[], string, string, int)
- string handle_file_upload()
+ boolean has_submit_been_attempted()
+ Article[] get_articles_array_restricted
+ Array[] get_articles_array()
+ update_article_status(integer, string)
+ User[] get_all_possible_authors()
+ Article get_article_by_id(int)
+ Comment[] get_article_comments(integer)
+ void add_comment(int, string)
+ void update_article(string, string, string[], string, string, int)
+ void get_article_editors(Article)
+ void highlight_article(Article)
+ Article[] get_highlighted_articles(int)
+ boolean can_user_like_dislike()
+ void like_article(User, Article)
+ void dislike_article(User, Article)
+ void has_current_user_liked_article(Article)
+ void has_current_user_disliked_article(Article)
+ Article[] get_most_liked_articles(integer)
+ Article[] get_recent_articles(integer)
+ Review[] get_recent_reviews(integer)
+ ColumnArticle[] get_recent_column_articles(integer)
DatabaseConnection
- host : string
- database_name : string
- username : string
- pdo_connection: PDO
+ void connect()
+ PDO get_connection()
+ void close_connection()
1
AbstractDataMapper
# _database_connection : DatabaseConnection
+ AbstractObject create_new(array)
+ void save(AbstractObject)
+ void delete(AbstractObject)
+ void update(AbstractObject)
+ AbstractObject[] get_all()
+ AbstractObject find_by_id()
GenericArticleMapper
- article_mapper : ArticleMapper
- review_mapper : ReviewMapper
- column_article_mapper : ColumnArticleMapper
- Article get_mapper_for_object(Article)
- Article get_mapper_for_id(integer)
+ void add_editor(integer, User)
+ void highlight(Article)
+ Article[] get_recent_highlighted_articles(int)
+ boolean is_article_highlighted(Article)
+ User[] get_article_editors(Article)
+ void store_impression(User, integer, string)
+ Article[] get_all_articles_with_impression_by_user(User, string)
+ Article[] get_most_liked(integer)
+ void add_author(User, integer)
UserMapper
CommentMapper
- string get_timestamp_by_id(int)
ArticleMapper
+ Article[] get_recent(int)
+ User[] get_authors(int)
ReviewMapper
+ int get_reveiw_score(int)
+ Review[] get_recent(int)
ColumnArticleMapper
+ User[] get_authors(int)
+ string get_column_name(int)
+ ColumnArticle[] get_recent(int)
Comment
+ username : string
+ content
+ a_id : string
+ time : string
0..*
Review
+ review score : integer
ColumnArticle
+ column_name : string
User
+ username : string
+ name : string
+ type : string
0..*
AbstractObject
- _id
+ set_id()
+ get_id()
Article
+ title : string
+ content : string
+ authors : array(User)
+ date : string
+ type : string
+ status : string
+ cover_image : string
+ editors : array(User)
+highlighted: boolean

Just as a warning: I am adamant that **nobody** could complete this assessment in 30 hours, let alone to the best of their abilities. Unfortunately, cramming 130 hours into three weeks alongside five other modules and another assessment is difficult, and as a result this design document has suffered. Apologies in advance.

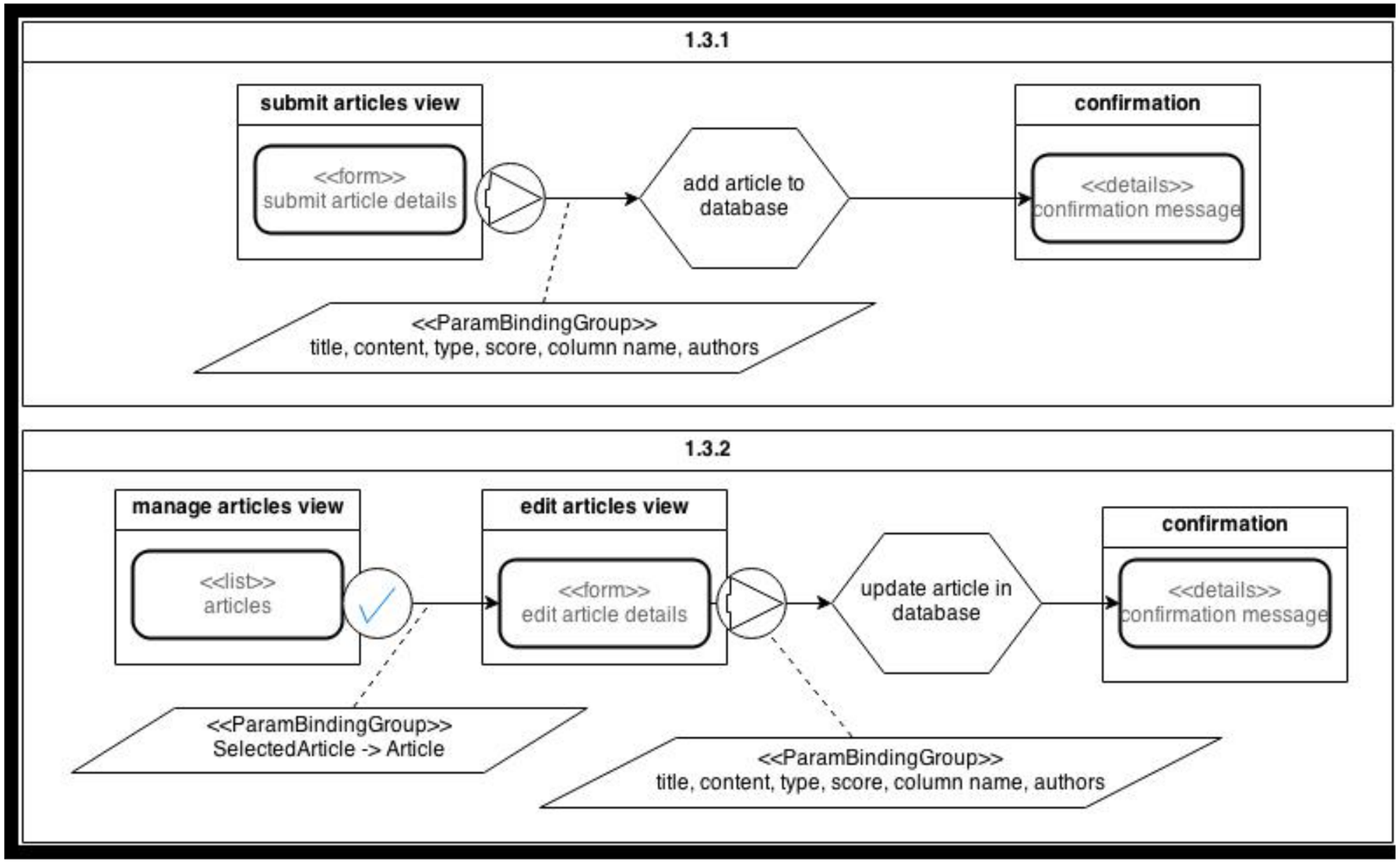
**1.1** The most obvious decoupling in this system is due to the data mappers. By handling all of the database-specific logic, the object mappers allow the model logic to ignore the database completely. I have a mapper for each main model element (review, column article, article, user, comments), but have also expanded on this by creating a GenericArticleMapper which can act as an intermediary: based on the object or data passed to it, it calls to the relevant article-based mapper. This is a revised design, that came about because I ended up with a lot of repeated logic in the model: it would need to store, find, or update, a content piece - regardless of which type it was - and so I would duplicate the “check which type, if review use the review mapper, otherwise ..” logic. The GenericArticleMapper consolidates this logic concisely, and still allows for specific handling of each type of content piece. Note that the GenericArticleMapper implements the same inherited methods that each of the specific mappers also inherits, and so forms a kind of interface. The GenericArticleMapper also handles operations that are common to all types of content piece: the logic for adding authors, for example, is not necessarily repeated across all of the article mappers, but consolidated into one.

I have split the majority of the model logic into UserManagementModel and ArticleManagementModel. This is because all of the operations of the CMS can be categorised as either managing users (login, registration, user types ...) or managing articles (submitting, publishing, highlighting etc), and as such separating it further feels arbitrary. This distinction is also pertinent because the User and Article mappers can be assigned to their respective parts of the model. There is some amount of crossover: for example the article management page, while practically only concerned with the management of articles, still requires some amount of interaction with users in the form of permissions checking. All pages on the website need to know which user is logged in, and many have some form of permissions-based check. For this reasons both of the model classes inherit from Model, which contains these crossover aspects.

I have designed a DatabaseConnection to keep a simple and consistent representation of this across all of the data mappers that require one.




**1.2** Above is the BPMN representation describing the process of moving and article from the initial writing through to publication. There are elements that are not described here, for example highlighting of said article. The specifics of commenting on an article are also not detailed here. I do not know what “provide information regarding how the requirements of the process are met the specified process" (sic) means. However, it is clear from this diagram that a writer may write a new article and submit it. The writer then can only wait for a response from an editor. If an editor has reviewed an article and has decided that a) it is not good enough (in this case ‘good enough’ is down to the editor to define) but b) that it is good enough to improve on, she/he can request changes from the writer. Upon which a writer may make those changes and resubmit it. It is also clear that only an editor has the ability to publish or reject an article.




**1.3**

In this diagram I have shorthanded the <<ParamBindingGroup>> parameter binding such that by “title, content, type, score, column name, authors”, I mean to say that all of those values are bound on that submit. In 1.3.1 we can see how a writer will enter the details into the submit articles view form, and submit it. On the submit event, all the details are bound, and further inserted into the database. The user is directed to a confirmation screen with a message to the effect that the article was successfully submitted.

In 1.3.2 there is a selection event on the articles list, which redirects the user according to their SelectedArticle to the ‘edit articles view’, from which point the interaction flow is almost identical to 1.3.1, except that the ‘edit articles view’ differs from the ‘submit articles view’ in that it is already populated with information.

**1.4**

**The ‘manage articles page’** embodies the ability to do almost everything involved in article management. The ‘selectors’ navigation on the left hand side operates as a simple filtering mechanism. A user, depending on their type, may filter articles not just by their status, but by whether they are an author or they are an editor. These ‘selectors’ are additive, and when selected all of the articles of the respective type is displayed in the main area.

In the context of article management, once displayed, the variable that best categorises the articles is an article’s status. For this reason, each article listing is also colour-coded according to its status. The colour coding is not an exact 1:1 mapping with the selectors on the left: for example, ‘my authored’ will show all of the articles by the current user, irrespective of status. The colours, however, are matched by the buttons that are available once an article is selected, in that the colour of a button is the colour the selected article will turn if clicked. This allows a user to perceive the button’s affordances (in the correct sense). The buttons also display ‘clickability’ (inner shadow) on mouseover and so have been designed with affordances in the incorrect sense, too. When a button is pressed, jQuery dynamically posts the request and displays a green success text above the button, unless there was an error in which case it is red.

Users are able to distinguish between the selectors on the left and the articles on the right by the visual barrier formed of three things: some small whitespace separation, a border, and the visual dissimilarity between selectors and article listings. The selectors both change shade when clicked, and display a subtle in-pressed-type shadow that helps the user understand they have been selected. Similarly, when selecting an article, a bold dotted border is placed on the article to make it clear it has been selected. When articles appear and disappear, they do so in an animated fashion such that they appear to be shrinking off to the left, where the selectors are. This gives the user a sense that the articles have not just ‘disappeared’ but are in fact only hidden.

When an article is selected, buttons are displayed to the user based on what the user has permission to do, and are able to do with that article. For example, an editor cannot highlight an article until it has been published. This avoids confusing clutter, and shortens the time a user takes to ‘form a plan of action’ about how they can undertake something because they are not scanning many controls that are not relevant.