Theme: An Approach for Aspect-Oriented Analysis and Design

Elisa Baniassad and Siobhán Clarke
Department of Computer Science
Trinity College, Dublin 2, Ireland
{Elisa.Baniassad, Siobhan.Clarke}@cs.tcd.ie

Abstract

Aspects are behaviours that are tangled and scattered across a system. In requirements documentation, aspects manifest themselves as descriptions of behaviours that are intertwined, and woven throughout. Some aspects may be obvious, as specifications of typical crosscutting behaviour. Others may be more subtle, making them hard to identify. In either case, it is difficult to analyse requirements to locate all points in the system where the aspects should be applied. These issues lead to problems achieving traceability of aspects throughout the development lifecycle. To identify aspects early in the software lifecycle, and establish sufficient traceability, developers need support for aspect identification and analysis in requirements documentation. To address this, we have devised the Theme approach for viewing the relationships between behaviours in a requirements document, identifying and isolating aspects in the requirements, and modelling those aspects using a design language. This paper describes the approach, and illustrates it with a case study and analysis.

1. Introduction

The intent of aspect-orientation is to allow developers to encapsulate system behaviour that does not fit cleanly into the particular programming model in use; it is aimed at breaking the hegemony of the dominant decomposition.

Behaviour that cannot be encapsulated because of its impact across the whole system is called crosscutting behaviour. Before encapsulating crosscutting behaviour into an aspect, the developer must first identify it in the requirements. This is a difficult task to tackle because, by their nature, aspects are tangled with other behaviours, and are likely to be described in multiple parts of the requirements document. Using intuition or even domain knowledge is not necessarily sufficient for identifying the potentially broad range of aspects within a reasonable amount of time. For example, it required significant effort to identify and characterise that prefetching could be modeled as an aspect in the FreeBSD operating system [7].

When identifying aspects early in the lifecycle, developers can currently apply three approaches. The common approach for aspect-identification is to look for the objects in a system first, and then attempt to spot the tangled and scattered behaviour as it becomes evident. This is an ad-hoc approach that is likely to necessitate re-design as aspects are discovered late in the design process. Alternatively, before the design process starts, developers might scan requirements for mentions of typical aspect-style behaviour, such as logging, tracing, or debugging functionality. This only covers a narrow range of potential aspects; it does not help with identifying crosscutting behaviour that does not fall into these categories, or that might be domain specific. To address this need, a developer might start by applying an aspect-oriented requirements engineering technology, and target non-functional requirements as an initial set of aspects [13, 11, 16]. However, there are likely to be many functional requirements in the system that probably break down into complicated and interrelated behaviours. Would any of those be aspects? Where would they crosscut the system?

We believe that in order to identify and model a broad range of aspects early in the lifecycle, and assess where they crosscut the system developers need support for analysis of the relationships between all behaviours described in requirements documentation. They also need support for translation of the results of the analysis into design models which can then be implemented in code.

The model we propose is the Theme approach. Theme provides support for aspect-oriented development at two levels. At the requirements level, Theme/Doc provides views of requirements specification text, exposing the relationship between behaviours in a system. At the design level, Theme/UML [4, 6] allows a developer to model features and aspects of a system, and specify how they should be combined. Our central claim is that Theme/Doc allows the developer to refine views of the requirements in order to reveal which functionality in the system is crosscutting,

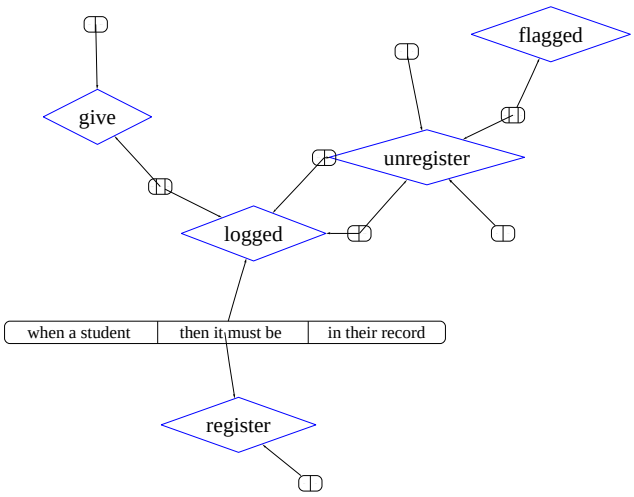

Figure 1. Action View of CMS Requirements

and where in the system it crosscuts. We also claim that the Theme approach assists with maintaining traceability from requirements to design, since requirements map directly to Theme/Doc views which map directly to Theme/UML models. This traceability also provides cues about requirements coverage in the design.

In this paper we present the approach by means of a small example (Section 2), and provide a case study to assess its effectiveness at aspect identification, the provision of traceability and coverage cues, and scalability of the approach (Section 3). We then discuss other issues and make observations about the approach (Section 4), review related work (Section 5), and conclude (Section 6).

2. Theme

In the Theme approach, a theme is an element of design as defined in [4]. Themes represent features of a system that can be combined to form a functioning whole according to a multi-dimensional model [17]. Each theme is a view of the object model of a system, where only portions of the object model relevant to a particular piece of functionality is shown. The Theme model has two kinds of themes: *base themes*, which may share some structure and behaviour with other base themes, while modelling these from their own perspective, and *crosscutting themes* which have behaviour that overlays the functionality of the base themes. Crosscutting themes are *aspects* [6].

The Theme approach is divided into two segments: Theme/Doc and Theme/UML. These both operate on and refer to the same themes, but depict them at different phases of the lifecycle. Theme/Doc provides views[1] and functional support for identification and depiction at the analysis phase, whereas Theme/UML allows depiction at the design phase.[2]

In this section we work through a small example to illustrate the basic points of how to use Theme/Doc and Theme/UML to support the identification, design, and design checking of aspects in a set of requirements.

2.1. Course Management System

The Course Management System (CMS) is a very small system, with nine requirements:

- R1. Students can register for courses.
- R2. Students can unregister for courses.
- R3. When a student registers then it must be logged in their record.
- R4. When a student unregisters it must also be logged.
- R5. Professors can unregister students.
- R6. When a professor unregisters a student it must be logged.
- R7 When a professor unregisters a student it must be flagged as special.
- R8. Professors can give marks for courses.
- R9. When a professor gives a mark this must be logged in the record.

2.2. Identifying Aspects Using Action Views

Using the Theme/Doc tool, a developer can view the relationship between behaviours described in requirements documentation, and determine which behaviours are base, and which are crosscutting. To help identify crosscutting behaviours we use the *action view* of the requirements document. Two inputs are needed to create the action view: a list of key actions identified by the developer by looking at the requirements document and picking out sensible verbs, and the requirements as written in the original document. Theme/Doc then performs lexical analysis of the text and generates the action view. Each action is potentially a theme to be designed separately in Theme/UML. For the CMS, we have identified five actions: register, unregister, logged, flagged, and give. Figure 1 shows the action view created by the Theme/Doc tool for the CMS. In the view we see the five actions as diamonds. The requirements from the text are shown as rounded boxes (*sentence records*) which contain the words from the original sentence. If the requirement sentence contains a key action, it will be linked to the sentence record. The intent of the action view is to highlight relationships between actions; the text of the individual requirements themselves is not the point at this stage, and so is not intended to be legible. Of course, a user may choose to enlarge any requirement for their information. In this view we have chosen to enlarge

[1] All Theme/Doc views are generated automatically in dot [12] format given the text of the requirements document, a list of key entities, and a list of key actions. The dotty graph visualization package is used for the layout and display of the views.

[2] Theme/UML designs are created by a developer using an appropriate UML editor.

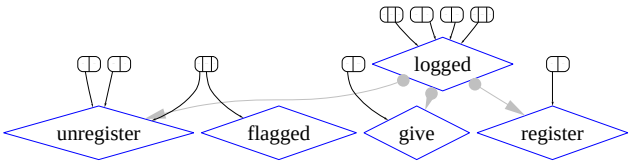

Figure 2. Clipped Action View

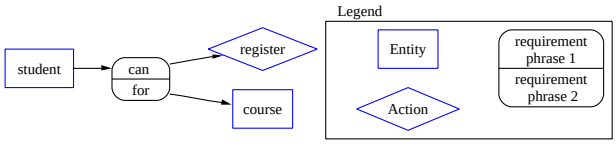

Figure 3. Theme/Doc Theme view: *register*

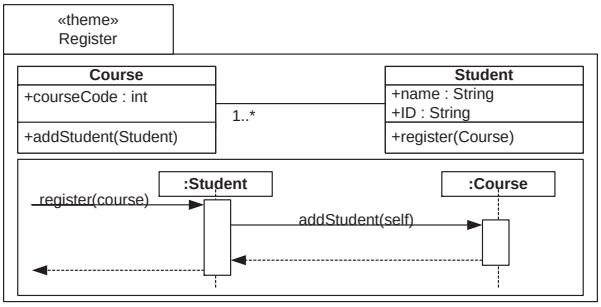

Figure 4. Theme/UML: *register*

R3, which reads "when a student registers then it must be logged in their record".

As we might expect, none of the actions in the requirements is isolated from the rest. They all relate in some way to the rest of the actions. The fact that they are linked shows us that there is crosscutting and tangling of behaviours within the requirements. For instance, we can see from this figure that the *logged* action is mentioned in four requirements, and that the *register* action is mentioned in two requirements. Requirements often refer to more than one action. For instance, R3 refers to both the *register* and *logged* actions.

We now look more closely at the actions and the shared requirements to determine which action is a base action, and which is crosscutting. Our aim is to separate and isolate groups of actions and requirements in the action view, arriving at two kinds of action/requirement groupings. The first are self-contained and have no requirements that refer to action in other groups, which we determine to be "base". The second kind are crosscutting and have requirements referring to actions in other groups.

We use the *clipping* functionality of the tool to achieve this separation and isolation. First, we examine each shared requirement to see which action it is more appropriately coupled with. For example, the requirement that links *logged* to *register* is R3, which describes logged behaviour that is added to registration behaviour. We intuit that logging is the primary behaviour of this requirement, and hence that R3 should be coupled with the *logged* theme. In deciding this, we have determined that *logging* is crosscutting and *register* is base. Second, we clip the arrow from R3 to *register* so that it is only linked to *logging*. That arrow is replaced by a grey arrow with a dot at its head which points from *logged* to *register*, indicating that *logged* crosscuts *register*.

We then visit each requirement that *logged* shares with other actions, to determine whether they belong with *logged* or the other action. Since we have determined that *logged* crosscuts *register* it is likely that it also crosscuts the other actions with which it is linked. We continue to snip the links between the shared requirements and the base actions, and leave them with the *logged* action. Finally, we arrive at our goal as shown in the *clipped action view* displayed in Figure 2: four actions with requirements that refer only to themselves and are therefore base, and one action that mentions others and is therefore crosscutting. In clipped action views, crosscutting themes are placed above the themes that they crosscut.

We now make one final observation from the action groupings. In this view we see that the *flagged* action is linked to the *unregister* action. We examine the requirement they share and make the decision that the flagging functionality should be included in the professor's behaviour in the *unregister* theme. An alternative might have been to consider that flagging behaviour crosscut unregistering behaviour. Now, each grouping becomes a theme we wish to model using Theme/UML.

2.3. Planning for Design Using the Theme View

Theme/Doc's *theme view* is used to plan the design and modelling of the themes identified in the previous step. Theme views differ from action views in that they do not only show requirements and actions, but also show key elements of the system that will need to be considered for each theme design in Theme/UML. To construct a theme view, a developer must supply an additional set of keywords: key entities. Like the action view, the theme view is created through lexical analysis of the text of the requirements document.

Figure 3 shows the theme view of the *register* theme clipped of crosscutting behaviour. We can see that it has only one requirement, which mentions nothing other than registration behaviour. When reading sentences in Theme/Doc, first read the element that points into the sentence record, then read the first phrase in the record, and then read the element pointed to by that phrase. If the phrase is empty, it just means there was only a space between the first and second element, with no connecting phrase. Then read the subsequent connecting phrase, and then the element to which it points. Continue back and forth between the record and its attached elements until the record ends. To read a sentence, do not read more than one element out from the sentence record. This sentence reads "Student(s) can Register for Course(s)".

We can use the theme view to plan which classes and methods appear in our Theme/UML for *register*. When modelling using Theme/UML (described in more detail in [4]), good object-

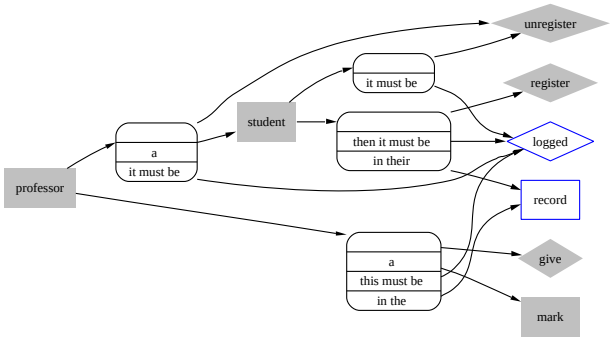

Figure 5. Theme/Doc Theme View: *logged*

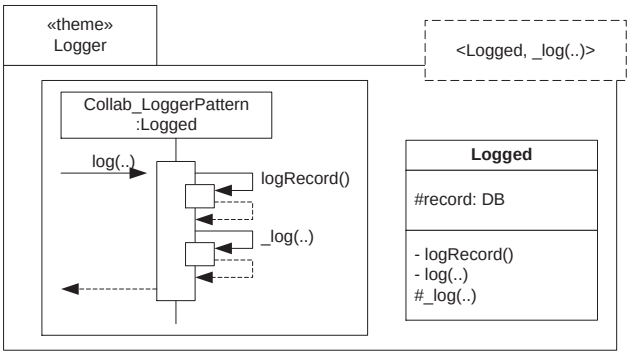

Figure 6. Theme/UML: *logged*

oriented design practices should be used to help determine which classes and methods are described. As is evident in Figure 4, in this case, we simply decide that each action in the theme view is a method, and each entity is a class. We also make some additional design decisions to make course registration work.

When modelling a crosscutting theme we want to model the crosscutting behaviour in an abstract and potentially re-usable way. We do not want explicit references to any base actions or entities. The theme view for crosscutting themes helps with identification of such elements by greying out actions and entities found in other themes. The remaining white actions and entities can then be used to guide the design of the abstract crosscutting behaviour. The grey actions are also used to determine the joining, or binding of the crosscutting behaviour to the greyed-out base.

An example of this is shown in Figure 5. We can see that only elements that are unique to the *logged* theme are `logged` and `record`. We can now model the abstract behaviour for logging records without refering directly to registration, unregistration, students, professors or giving marks. The model for the *logged* theme is shown in Figure 6 in which we can see that the `record` element of the *logged* theme has become a database in this design, and that the `logged` action has been loosely translated into the `logRecord` method. Theme/UML allows reasoning about elements from a base by using templates that will be bound to real base elements at a later stage.

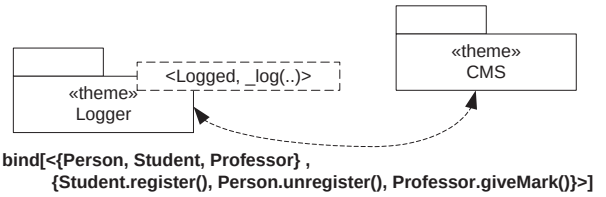

Figure 7. Composition of *logged* **and other themes**

To determine how the crosscutting theme should hook into the base features, we look back at the theme view for *logged*. All of the grey actions are behaviours that are crosscut by the *logged* behaviour. The Theme/UML for *logged* provides the `_log()` template method as the handle method for the base behaviour. To resolve which method the `_log()` method actually is, we use the *bind* feature of Theme/UML to bind it to a concrete method from another feature of the system. So, we bind the `_log` method to the grey actions in the *logged* theme view: register, unregister, and give(mark). To determine which classes the methods belong to we can look at Figure 5 and the relevant Theme/UML models for each action. We see that the `register` method is associated with the `Student` class (also illustrated in Figure 4) and the `giveMark` method belongs to the `Professor` class. Both the `Student` and `Professor` classes are linked to the `unregister` method, so we use their parent class, `Person`, specified in a Theme/UML model not shown here, for the binding. Figure 7 depicts the bind statement.

2.4. Re-checking Themes Using the Augmented Theme View

After the themes have been designed using Theme/UML, we revisit the Theme/Doc theme views for help in verifying that the design choices we made align with the requirements. To do this, we augment the Theme/Doc theme view with representations of those design decisions. To augment the *register* theme view, for instance, we need to add one method, and three associations. The result is shown in Figure 8. In the augmented view, “has” relationships are shown using an inverted arrow at the container element, pointing to the contained element. The “calls” relationship is shown with a dashed arrow. We’ve added a method to the view, which is shown as an action. To distinguish it as an augmentation, the corners are marked. These augmentations are specified by the user and fed into the Theme/Doc tool which generates the augmented graph.

By looking at this graph, we can see that the Theme/UML design seems to align with the requirement shown in Theme/Doc. If there were other design elements (other classes or methods) included in the Theme/UML design, we would see them here, and would be able to assess their correctness with a view to the requirements. This view is not intended to provide any formal basis for analysis, but rather to place, in the context of the requirements, the design decisions that were made. This context then gives the

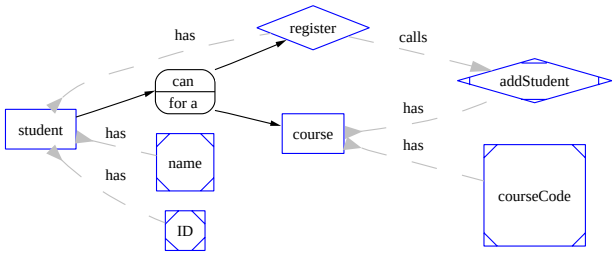

Figure 8. Augmented View: *register*

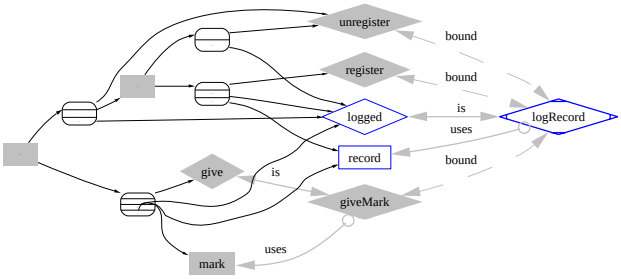

Figure 9. Augmented View: *logged*

developer a view for verifying the design decisions themselves.

Since the Theme/UML for a crosscutting theme only includes information about the abstract crosscutting behaviour, only the structures related to the behaviour are added to the theme view. This view is shown in Figure 9, in which all augmented elements are enlarged. As before, all elements not unique to the crosscutting behaviour are shown in grey. This view has two relationships the *register* augmented view did not: "is-a" relationships, shown with a bi-directional arrow, and "bound" relationships which are shown with bi-directional dashed arrows, as they are in the Theme/UML bind notation. Elements that are involved in the binding but that are not shown in the view are also added. For instance, the action `giveMark` is in grey because it is not in the Theme/UML for the *logged* theme, but is included because the `giveMark` method is bound to the `logRecord` method. To assess the completeness of the design decisions for this theme we look at whether all the greyed actions are bound to the action representing the crosscutting behaviour. We also scan to see that all the structures mentioned are dealt with in some way. We can see that `mark`, for instance, is included in the crosscutting behaviour because it is used by the `giveMark` method which is bound to `logRecord`.

3. Case Study

The goals of this case study were to test the Theme approach on a larger example, and perform preliminary assessment of it in terms of effectiveness for finding aspects, support for assessment of requirements coverage, facilitation of requirements traceability,

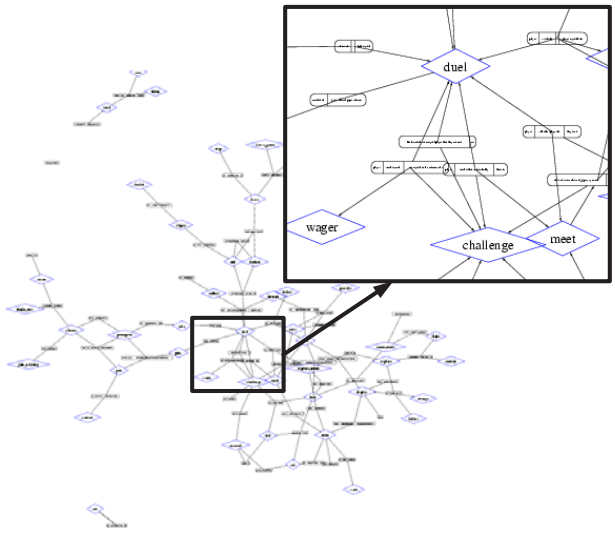

Figure 10. Game Action View: All Actions

and scalability. We will first give a general description of the location aware game that was the basis for the case study, and then provide results and analysis.

3.1. Location Aware Game

The set of requirements used in this case study are those for a location aware game called the Crystal Game, which was developed in an independent research project. The game has 89 requirements, so is roughly ten times larger than the example provided in Section 2. The game involves any number of human players, each of whom is provided hand-held devices. The game can be set in any location, as long as the location has the necessary infrastructure. The object of the game is to collect crystals that are deposited throughout the location. As a player moves around the game space, their hand-held device will alert them when they have encountered a crystal. Computer-generated characters also take part in the game. When a player encounters one of them, they will interact, and perhaps duel. When a player encounters another player, they will duel, and the loser of the duel will turn all of their crystals over to the winner. The game ends after a specified time period. The winner is decided by how many crystals each player has. There are other constraints and requirements in this game which will be of interest and will be described in later sections.

3.2. Results

In this section we review the steps we took to apply the Theme approach to the Crystal Game requirements.

3.2.1. Finding Themes

We identified 59 actions in the game requirements, and generated an action view to examine their relationships. Based on intuition and some cursory analysis of the view, we determined that all of these actions should not be modeled as separate themes.

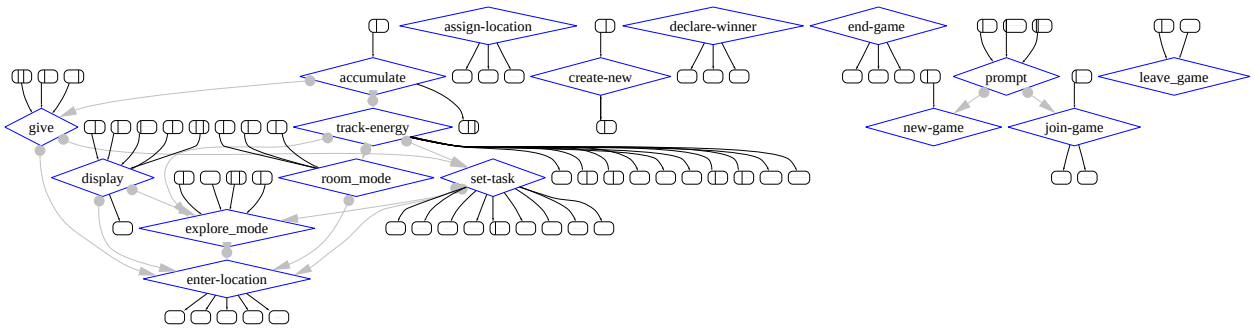

Figure 11. Clipped Action View of Major Game Actions

Instead, we examined the view to determine the relationships between the actions, to decide how to group the actions into larger themes.

This was a mainly analyticial process, but it was supported by the action view. Because actions that share requirements are displayed close to one another in the view, we were able to examine closely located actions to assess whether they should be grouped into a common theme.

We used the view shown in Figure 10 to perform such an assessment. This figure shows the initial action view for the game, with the centre portion of the view enlarged. The enlarged view shows four actions, *duel, wager, challenge* and *meet*. We examined the requirements they shared, considered the meaning of the actions, and determined that *duel*, *wager*, and *challenge* should all be grouped under the general heading of *duel*, since players challenge one another to duel, and wager crystals on the outcome of a duel. In that case, we classified *duel* as being more major than *wager* and *challenge*, which we saw as sub-actions of *duel*. The *meet* action was connected to *duel* because when players encounter one another they duel. We examined requirements shared by *meeting* and *duelling* and determined that since they were not synonymous, they should not be grouped into one theme. Later, we determined that *duel* and its sub-actions should be grouped under the more major action, *set-task*. In the end, we arrived at the view shown in Figure 11, which displays the 16 major actions which became our themes. Of those actions, five are independent, while others share requirements, and hence crosscut one another in some way.

The clipping functionality of the tool helped us investigate the major action view to determine which themes are crosscutting and which are base. In the case of the *prompt* theme this was straightforward. The *prompt* theme (shown to the right of Figure 11) shared requirements with two other themes, *new-game* and *join-game*. By examining the shared requirements it could be seen that the prompting behaviour crosscut these two themes.

As is visible on the left side of Figure 11, there are several related themes. To determine which of those was crosscutting, we began by assessing the requirements between the *explore-mode* and *enter-location* themes. We determined that *explore-mode* crosscut *enter-location*. By continuing to examine themes that shared requirements with *enter-location* we further determined that *room-mode* was crosscutting, as was *give*, *accumulate*, *set-task*, and *display*. We then examined the remaining shared requirements, and encountered themes that crosscut other crosscutting themes. For instance, the *track-energy* theme was determined to crosscut *set-task*, *room-mode* and *explore-mode*, all of which crosscut *enter-location*. There are five themes that crosscut other crosscutting themes: *display, set-task, give, track-energy* and *accumulate*.

3.2.2. Modelling and Composing Themes

The theme view was used to drive the modelling and composition semantics for design of the game using Theme/UML. Figure 12 shows the Theme/UML for the *give* theme. The *give* theme handles the situation in which a player or a character gives a crystal to another player. This happens in three situations: when a player meets a character, the character gives them a crystal; when a player loses to another player, the loser gives the winner a crystal; and when a player completes a task for a character, the character gives the player a crystal.

To ensure that the developer carefully considers the order in which crosscutting themes are composed with base themes, Theme/UML allows only one crosscutting theme per composition. We therefore needed to inspect the crosscutting relationships to determine the order of binding. For this we used the clipped action view shown in Figure 11. In this view, the themes are positioned hierarchically, based on whether they crosscut one another. The grey arrows indicate which themes crosscut other themes. We can see, for instance, that there are no grey arrows extending from *enter-location*, which indicates that it is base functionality. This is also suggested by the fact that *enter-location* is positioned lowest in the graph. To determine the binding order, we begin with the lowest themes, and work to the highest, incrementally binding in one crosscutting theme at a time. To determine what is first in the binding we identify the themes that crosscut only that theme (*room-mode* and *explore-mode*), and placed those first, and second in the binding order. The final bindings were done with the "more" crosscutting themes: *display, give* and *track-energy*. The very last binding is of *accumulate*, since it crosscuts the *give* and *track-energy* themes.

The *give* theme bindings are also shown in Figure 12. *Give* is bound fifth in the general order, and is bound to an already composed theme, *enterLocation-exploreMode-roomMode-display-setTask*. As the name suggests, the composed theme is the product of binding *enter-location* to its closest four crosscutting themes.

3.2.3. Checking Theme/UML

Finally, we used the augmented theme view to check the validity of the design choices we made. An example of this is shown in Figure 13. In this figure we can see that four elements have been added to the view: `giver, receiver, receive`, and `item`. The relationships from the Theme/UML are also shown. We can see that all the grey actions have been bound to `give`, except for `wager` and `duel`, which are drawn in through `lose`. We can see that both players and characters are `giver`s, but only a player is bound to `receiver`.

3.3. Analysis

In this section we discuss how the results of the application of the Theme approach reflect on the four areas of interest: effectiveness at support for aspect identification, requirements coverage, traceability, and scalability.

3.3.1. Effectiveness of Support for Aspect Identification

Through the application of the Theme approach, we were able to identify eight aspects: *explore-mode, room-mode, accumulate, track-energy, give, set-task, display* and *prompt*. Had we carefully read the requirements document we may have identified seven of these behaviours as aspect behaviours since they provide tracking or logging style functionality. However, it is unlikely that we would have identified the *give* functionality as an aspect because mentions of the *give* action are spread throughout the document, and it might have been difficult to recall that the same abstract behaviour is occurring with relation to different system features. Also, since in the document text it is described as a consequence of other actions, such as meeting, and duelling, it is possible that we would have automatically thought of *give* as a method in those actions. It wouldn't have been until we were modelling or implementing it that we would have noticed its crosscutting nature. We can see from Figure 12 that the *give* functionality works as an aspect, as it can be expressed in an abstract way, and can be overlayed on the appropriate behaviour through bindings.

We also found our approach effective support for determining the binding order for multiple crosscutting themes. This may be otherwise difficult to determine.

3.3.2. Requirements Coverage

We were initially concerned that it may also be difficult to assess whether all the requirements have been associated with a theme. We noted that the action view can be used to monitor requirements coverage, because if a requirement is not associated with a theme it is orphaned in the view. By orphaned, we mean that only the sentence record for the requirement appears, without being linked to a diamond-shaped action. We found that a requirement could be orphaned in two ways. Orphans can appear in the initial action view if the requirement contains no key actions. This may happen if it refers to another action, but does not mention it explicitly. By inspecting the requirement we can identify the requirement's original location in the text, and can read the requirement in context to determine to which action it refers. The other

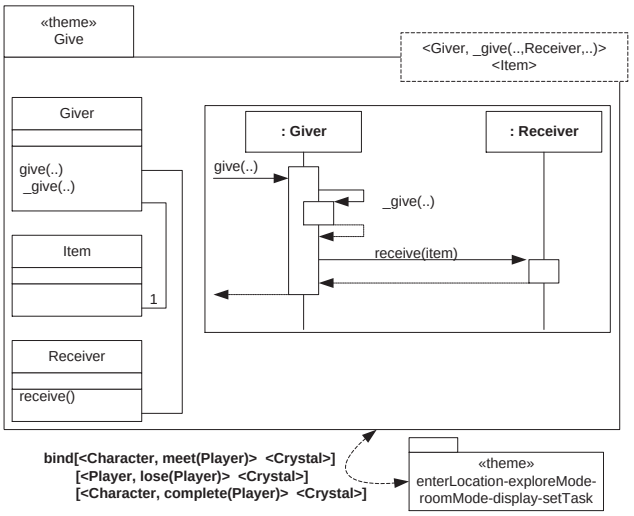

Figure 12. Theme/UML: *give*

way orphans can appear is when forming the major action view. As major actions are identified, they are added to a new list of keywords. The minor actions that have been grouped under the major action will be annotated so that they will be linked to the major action in the view. Minor actions that are not grouped with major actions will disapear, and their requirements will appear to be orphaned. We systematically visited the orphaned requirements to determine whether any of their minor actions should be promoted to major, or whether to group those requirements under other major actions.

3.3.3. Traceability

We found that the Theme approach provides traceability by explicitly linking portions of a requirements document to their outcome in a design model. Action views directly represent the requirements documentation, and are traceably refined into major action views. These maintain both the original content of the requirements, and also a developer's choices about how the requirement should be grouped into features. Major action views can then be transformed into theme views, which have a 1-to-1 mapping to their modelling in Theme/UML. As a final step, the design choices made at the modelling level are then added into the theme view to form the augmented view, which places those design choices in context with the requirements. The developer can examine any Theme artifact and view backward and forward traceability links. For instance, when looking at a theme view we obtain backwards mappings by inspecting the sentences displayed to find their location in the original text, and by observing which actions were deemed to be minor. We can also obtain forward mappings identifying the Theme/UML that models the view based on their common name.

For example, in Figure 13 we see the augmented view for the *give* theme. If we inspect the sentences shown in this view we would find that they are sentences 4, 18 and 69 from the original text. We can see that actions *duel, lose, wager, meet*, and

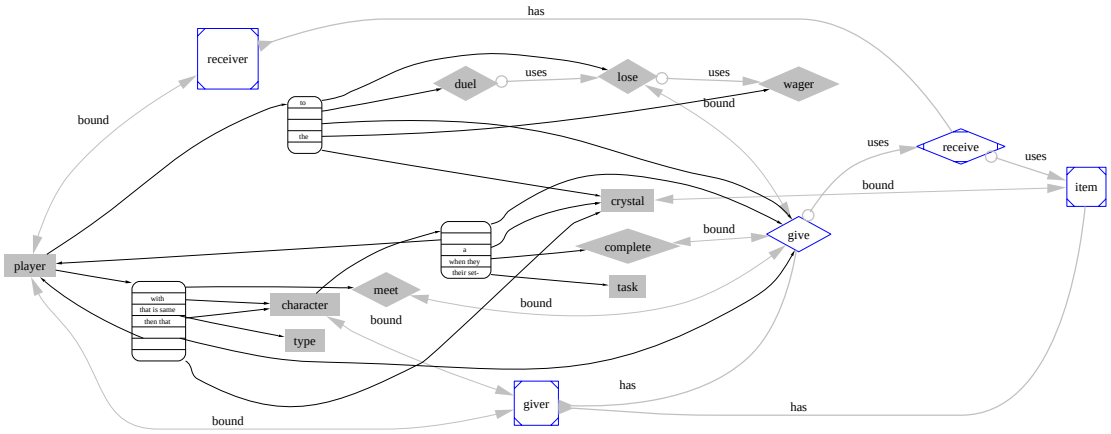

Figure 13. Augmented View: *give*

complete are all contained in other themes. By visiting the action view we can infer that they were deemed minor actions since there is no major theme named after any of them, and that three were grouped under the *set-task* theme, and one was grouped under the *enter-location* theme. This is further confirmed by viewing the *give* Theme/UML shown in Figure 12.

3.3.4. Scalability of Action Views

In the small CMS example, almost every action became a theme. This allowed for easy association of requirements with themes, and for easy viewing. However, in the larger game example, we classified actions into two types: major and minor. Major actions became themes, while minor actions were slotted to become methods within a theme. When forming the action views for this larger example, we chose only to use the major actions to form the view, rather than viewing all the major and minor actions. This allowed us to determine and assess the relationships between themes without the added "noise" of the minor actions.

Were we to scale the requirements further, it would be necessary to apply other approaches, since it would not be feasible to fit an entire action view for a very large system onto a screen or a page. In this case, query functionality, provided by the Theme/Doc tool, is needed to form sub-views that could be examined separately from the entire action view.

The current approach has essentially provided two "zoom-levels" of action view; a developer can zoom-in to see all the actions, or can zoom-out and just see the major actions. It would likely be useful, in a larger system, to provide more degrees of zooming so at some level the entirety of the system could be seen in one view.

4. Discussion

In this section, we provide further discussion of issues we noted while evaluating the Theme approach.

4.1. Synonyms

Synonyms are handled through a synonym dictionary which, for the sake of the action view, automatically augments the requirements text so that the correct associations will be made. This is more complicated when two words are the same but have different meanings in terms of the system. For instance, the term *give* was used in the Crystal Game not only for giving crystals, but also for giving audio and visual signals to players. The action view helped identify instances where this occured, because the common action brought together other actions which, upon analysis, should not be linked. For instance, the common term *give* brought closer together *accumulate* and *prompt*. We could intuit from having read the requirements document that these two actions should be unrelated. When inspecting the relationships around the *give* action, it was clear that the term was being used in different senses. We then used the annotation feature of the tool to replace the audio sense with the term *give-audio*. These annotations are not shown in the theme view.

4.2. Scattered Requirements

When analyzing the requirements for the context aware game, we found that not only did behaviours crosscut other behaviours (such as *give* crossutting the *meet* action in the *enter-location* theme), but behaviours were also described in a way that crosscut the requirements. The *give* behaviour was described in three locations in the text: at lines 4, 18, and 69. This is because the original authors of the requirements did not see the *give* behaviour as a major enough behaviour to discuss separately, though they did organize the document in terms of the features as they saw them. This scattering was also true of the other crosscutting themes, with the exception of *room-mode* and *explore-mode* which were described in their own sections.

4.3. Ambiguities Found in Requirements

We found that the Theme approach helped us identify ambiguities in the original requirements specification. While refining

the 59 actions into the 16 themes we found that there were subtle ambiguities in the initial requirements document. For instance, we found that it was not explicitly mentioned how crystals were collected by a player, unless the crystal was actually given to them by another player or a game character. One requirement mentioned picking up crystals ("a player explores the world and picks up crystals"), and another mentioned the accumulation of crystals ("a player collects crystals by discovery in a location, or when a player or character gives one to them.") Though it was implied, there was no specific description of a player actually picking up a crystal when they discover it in a location. This subtle ambiguity was discovered when we saw that the *pick-up* action, and the *collect* action were not located close to one another in the view shown in Figure 10, though we knew intuitively that they should be related. This was highlighted because the *collect* action was closely placed to the *give* action.

4.4. Programming Aspects of Aspects

In our case study, we identified themes that crosscut other crosscutting themes, in other words, aspects of aspects. The *give* theme, for instance, crosscut *set-task*, which crosscut *enter-location*. Theme/UML models can be implemented in aspect-oriented programming languages that provide constructs for separating base code from aspect code and that provide weaving mechanisms and supporting constructs [5]. AspectJ [1], allows specification of join-points in aspect code as well as core code, allowing for the expression of aspects that weave into other aspects. This can be done by providing the aspect name and method in the pointcut statement of the aspect to be crosscut. To implement the first of the *give* bindings in AspectJ one would specify:

```
...
public aspect Give
pointcut uponMeeting():
  execution(void Character.meet());
...
```

5. Related Work

The intent of this work is to support analysis of requirements documents for identification of aspects, and traceability of those aspects to (and from) the design. For this reason, our related work primarily describes work on identification of aspects from requirements; we do not focus on standalone aspect-oriented design approaches. Previous publications on Theme/UML [4, 6, 5] describe other work on design.

5.1. Aspect-Oriented Requirements Engineering

There have been several efforts in capturing and relating aspect-oriented requirements [16, 18, 8, 13, 11, 10, 3] Here we consider the two which relate most closely to the Theme approach.

Rachid et al [13] provide the AORE (Aspect-Oriented Requirements Engineering) model and ARCaDe (Aspectual Requirements Composition and Decision support) approach and tool for describing components and requirements-level aspects. Examples of these aspects are compatibility, availability, or security. This work builds on the ViewPoints model [9], which is intended to support the integration of heterogeneous requirements specified from multiple perspectives. An early stage in the AORE model is the identification and specification of concerns. The approach to this differs from the Theme approach to concern identification in that it relies on the domain knowledge of the developer to identify possible non-functional requirements to be taken into account when implementing a particular requirement. Those concerns are not explicitly mentioned in the requirements specification; it is up to the developer to ascertain their relevance on their own. We see this as a complemetary approach to our own. Such domain knowledge will always play a large part in system design. The Theme/Doc approach aims to support the analysis of relationships between behaviours described in requirements specifications. It is possible that the Theme/Doc approach to aspect identification could be used during the concern identification phase of AORE, or could support AORE's extension to include functional as well as non-functional requirements.

Katera and Katz [11] propose architectural views of aspects as a means for reasoning about the relationships among aspects in a system. They describe aspects as crosscutting augmentations to an existing design. In particular, they allow for specification of the overlap between aspects through the concept of a *sub-aspect* that provides the overlapping functionality, and they make relationships between aspects explicit. A UML approach is given to support these views which differs from the Theme/UML approach: it provides additional architectural support for aspect modelling to that provided by Theme/UML, and it uses aspect mappings rather than multi-dimensional composition style semantics. Theme/Doc could be integrated into this approach since the relationships exposed between behaviours in a set of requirements could be used to establish the behaviours between aspects and sub-aspects in this approach, as well as support the identification of functionality shared between components.

5.2. Concept Graphing

The graphical approach employed by Theme/Doc is similar in spirit to conceptual graphs (CGs), as introduced by Sowa [14]. These are visual systems of logic that are readable by humans. CGs provide a formal graphical representation of concepts and the relationships between concepts. CGs have been used for automating consistency checking in multiple-perspective software specifications [15]. This work differs from the approach used in Theme/Doc in that it is concerned with the integration of heterogeneous requirements, whereas we are focused on providing views which expose relationships between functions and entities described in requirements. More investigation would be needed to see whether the formalisms employed by these techniques could be integrated into the Theme/Doc approach as a way to blend it with aspect-oriented requirements engineering approaches.

The Theme/Doc approach draws from experience developing the Design Pattern Rational Graph [2] approach and tool, which provides developers with a way to link high-level concepts described in a design pattern with their implementation in source. A DPRG consists of three levels, the pattern level, the code level, and the link level which relates the other two. The pattern level is a graphical representation of the sentences in the design pattern

text. The format for the graph of a sentence is based on the separation of nouns and verbs in the sentence, and the identification of keywords. The resultant graph provides chains of sentences that contained high-level concepts linked down to sentences that provided concrete implementation details. This representation differs from the views offered in Theme/Doc. The pattern level graph is generated using natural language parsing, whereas the Theme/Doc views are produced by lexical analysis alone. Pattern level graphs represent each word in a sentence as a node; Theme/Doc views isolate key actions and entities from their source sentences. These approaches also differ in intent: the pattern level graph of a DPRG is intended to assist a developer in understanding an existing design by decomposing its textual description and relating it to design and implementation. Theme/Doc provides visual cues for requirements analysis, and aspect identification, composition, and design. As such, the views presented in this paper are designed to map directly to Theme/UML, and to expose relationships in requirements text to facilitate identification of aspects.

6. Conclusions

In order to identify aspects in a set of requirements and map them to design, we need to see how behaviours described in the requirements relate to one another. In this paper we have presented the Theme approach, which provides a model and tool support for identification of aspects in requirements specifications, design level modelling of aspects and their composition, and checking design decisions in the context of the requirements. The Theme approach is based on Theme/UML which is augmented by the Theme/Doc process presented here. Theme/Doc provides four views of a requirements specification: the action view; the clipped action view, which clusters requirements with particular behaviour, and shows which actions were chosen to crosscut others; the theme view, which depicts the entities and actions of a cluster from the clipped view; and the augmented theme view, which places design decisions into the context of the requirements. Our case study showed that this approach is effective in helping to identify aspects in requirements specifications. In addition, these views provide traceability links from the requirements to the design as modeled in Theme/UML. We also found that the approach could be used to check coverage of requirements in design, and we identified functionality that would enhance the scalability of the approach.

7. Acknowledgements

We would like to thank Conor Ryan, Alan Gray, David McKitterick, Karl Quinn and Tonya McMorrow from the original Crystal Game development team, on which our case study was based.

References

[1] Aspectj home page, Xerox PARC, USA, http://aspectj.org/.

[2] E. Baniassad, G. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. In *Proceedings of the International Conference on Software Engineering*, pages 352–362, 2003.

[3] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The tropos project, 2002.

[4] S. Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, July 2002.

[5] S. Clarke and R. Walker. Towards a standard design language for AOSD. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 113–119. ACM Press, 2002.

[6] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, pages 5–14, 2001.

[7] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 50–59, 2003.

[8] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190, 1996.

[9] A. Finkelstein. The viewpoints faq. *BCS/IEE Software Engineering Journal*, 11(1), 1996.

[10] J. Grundy. Aspect-oriented requirements engineering for component based software systems. In *4th IEEE International Symposium on Requirements Engineering*, pages 84–91.

[11] M. Katera and S. Katz. Architectural views of aspects. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 1–10, 2003.

[12] E. Koutsofios and S. North. *Drawing graphs with dot*. Murray Hill, NJ.

[13] A. Rashid, A. Moreira, and J. Araujo. Modularisation and composition of aspectual requirements. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 11–20, 2003.

[14] F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1998.

[15] T. T. Sunetnanta and A. Finkelstein. Automated consistency checking for multiperspective software applications. In *Proceedings of the International Conference on Software Engineering Workshop on Advanced Separation of Concerns*, 2001.

[16] S. Sutton and I. Rouvellou. Modeling of software concerns in cosmos. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 127–133, 2002.

[17] P. Tarr, H. Ossher, W. H. Harrison, and S. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[18] X. Wang and Y. Lesperance. Agent-oriented requirements engineering using congolog and i*. In *Submitted to AOIS-2001, Bi-Conference Workshop at Agents 2001 and CAiSE'01.*, 2001.