Supporting Separation of Concerns Throughout the Lifecycle with Theme/UML

Siobhán Clarke

Trinity College
Dublin
Republic of Ireland
Agenda

- The Aspect-Oriented Software Development (AOSD) field
- The problem addressed with Theme/UML
- Decomposing/composing object-oriented designs
- Composition patterns
Other approaches to “improving” OO include domain-specific languages, generative programming, generic programming, constraint languages, reflection and metaprogramming, feature-oriented development, views/viewpoints, …

(See [www.aosd.net](http://www.aosd.net) for a more complete set of references to the field)
Requirements Specification Paradigm

- **Service O from Perspective P**
- **Function X for Role Y**
- **Feature C from View A**
- **Feature A from View B**
- **Feature A from View A**
- Description in natural language
Object-Oriented Specification Paradigm

Most basic units of decomposition: *object*, and *class*

Object encapsulates additional units

- Structural: attributes and relationships
- Behavioural: operations, interfaces, methods
Structural Mismatch
Requirements - OO Designs: Scattering

Add Resource
Remove Resource
Order Resource
Search Resource
Borrow Book
Return Book
Pay Fine
Synchronize

ResourceManager

Borrower
Copy
Fine
Book
Periodical
Location
Course
CopyNoGenerator
Structural Mismatch
Requirements - OO Designs : Tangling

Add Resource
Order Resource
Remove Resource
Search Resource
Borrow Book
Return Book
Pay Fine
Synchronize

ResourceManager

+ addBook()
+ addPeriodical()
+ removeResource()
- deleteResource()
+ orderBook()
+ orderPeriodical()
+ search()
# waitWriterReaders() {concurrent}
# waitWriter() {concurrent}
# wait() {concurrent}
- incrementReaders()
- decrementReaders()
- incrementWriters()
- decrementWriters()

Resource
- title : String
- publisher : String
- date: Date

Resource
+ setLocation(Location)
+ setCourse(Course)
+ setOrder(Order)
+ addCopy(Copy)
+ deleteCopies()

Book

Periodical

CopyNoGenerator
What about crosscutting concerns?

Crosscutting concerns are not well encapsulated by OO languages

- The object-oriented paradigm modularises based on class/object, interfaces and methods
- Where behaviour impacts multiple different classes and methods, (and therefore is *crosscutting*) it is not possible to encapsulate that behaviour using standard OO languages

Simple tracing example: the entry to and exit from each operation called is traced:

Structurally, design elements handling tracing can be separated...

... and structurally, classes requiring trace behaviour need simply add a relationship

But, what about the interactive behaviour?
What about crosscutting concerns?

Crosscutting behaviour must be specified wherever required.

Of course, we could say:

“This isn’t code, so a designer would never bother repeating all these interactions”

But other approaches are ad-hoc and undependable.
What about crosscutting concerns?

Change to crosscutting behaviour has significant impact across design

- If trace behaviour changes, all interactions have to change.
- If we want to remove trace behaviour from any particular operations, we have to change their interactions.
What about crosscutting concerns?

Reuse of crosscutting behaviour is not straightforward

- From a structural perspective, reuse may be reasonably straightforward – in this case, the classes and methods relating to trace have been separated.
- However, reuse of the behavioural specification is less easy. We have to examine the interactions, and extract the relevant pieces.
Theme/UML: Decomposition based on Requirements Specification

Add Resource
- Remove Resource
  - AddResource
  - OrderResource
  - RemoveResource
- Order Resource
  - OrderResource
- Borrow Book
  - BorrowBook
  - SearchResource
- Pay Fine
  - PayFine
- Return Book
  - ReturnBook
- Synchronize
  - Synchronize
Implications of Decomposition based on Requirements

- Support for *overlapping* specifications

  Same core concepts appear in different subjects with possibly different specifications

- “Crosscutting” specifications supported

  Behaviour affecting multiple classes may be separated using composition patterns
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:

✓ the elements that correspond (implicit, explicit)
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:
  - the elements that correspond (implicit, explicit)
  - how corresponding elements are to be integrated – that is, understood as an integrated whole….

MERGE Integration: All design elements relevant for composed design
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:
  - the elements that correspond (implicit, explicit)
  - how corresponding elements are to be integrated – that is, understood as an integrated whole….

**OVERRIDE Integration:** Design elements replace previous specifications
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:
  - the elements that correspond (implicit, explicit)
  - how corresponding elements are to be integrated
  - how to reconcile conflicts

Defined as a leaf!
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:
  - the elements that correspond (implicit, explicit)
  - how corresponding elements are to be integrated
  - how to reconcile conflicts

- potential for conflict with any UML construct’s properties
- composition model supports reconciliation attachments to CR:
  - precedence
  - default values
  - explicit values
  - transform functions
Composing Design Models: Composition Relationship

- A new kind of design relationship
- Defined between design elements indicating:
  - the elements that correspond (implicit, explicit)
  - how corresponding elements are to be integrated
  - how to reconcile conflicts
  - the “real” elements to replace placeholders in patterns

  - Particularly useful for composing “cross-cutting” behaviour patterns
  - Combination of composition semantics and UML templates
Merging behaviours from different models

- This is specified with a composition relationship which details the elements to be merged.

Matching criteria on the composition relationship defines the operations to be merged.

- Composition semantics uses delegation to merge corresponding operations from different models.
Capturing crosscutting behaviour in a composition pattern

• A “Composition Pattern” (CP) is a package that contains the design models required to specify crosscutting behaviour.

• A CP may be composed with other design models, merging those design models with the crosscutting behaviour.

• A CP does not contain a reference to any particular design element its aspect may crosscut.

• These properties of CPs present two important requirements for a design language:

  1. Merge semantics for crosscutting behaviours
  2. The ability to reason about elements it may crosscut without explicit reference
Reasoning about elements to be crosscut

- The UML has a notion of *templates* which are described as “parameterised model elements”, with formal parameters which may be “bound” by actual model elements.

- CPs extend the UML notion of templates, allowing a package to have multiple *pattern classes* with *template parameters* within those pattern classes.

```plaintext
A

\[ T, \text{ Integer} \]

\[ \text{AClass} \]

\[ \text{BClass} \]

\[ \text{Trace} \]

\[ \text{TracedClass} \]

\[ \text{File} \]

\[ \text{traceEntry(String)}\]

\[ \text{traceExit(String)}\]

\[ \text{traceEntry(tracedOp.name)}\]

\[ \text{traceExit(tracedOp.name)}\]

\[ _\text{tracedOp}()\]

\[ \text{AClass} \{ \text{BClass, 24} \} \]

\[ \text{AClass} \{ \text{BClass, 24} \} \]
```
Binding composition pattern to a base design

- CPs also build on the UML notion of a template binding relationship by extending composition relationships to include the ability to *bind* multiple real elements to the parameters.

- Any class from the base design that replaces a pattern class has the specification of the pattern class merged with it.

- Crosscutting behaviours are merged as defined by the interactions – one is generated for each of the operations that replace the template.
Observer aspect example (1 of 2)

- The observer pattern defines interactions between multiple pattern classes - *subjects* whose changes in state are of interest to *observers*.

Subject

- `+ aStateChange()`
- `- _aStateChange()`
- `- notify()`
- `+ addObserver(Observer)`
- `+ removeObserver(Observer)`

Observer

- `+ update()`
- `+ start()`
- `- _start()`
- `+ stop()`
- `- _stop()`

Vector

```
< Subject, _aStateChange(..) >
< Observer, update(..), _start(.., Subject,..), _stop(..,Subject,..) >
```

**Diagram:**

- The `Subject` class has methods `aStateChange()`, `notify()`, and methods to add and remove observers.
- The `Observer` class has methods `update()`, `start()`, and `stop()`.
- The `update()` method in the `Observer` class triggers the `notify()` method in the `Subject`, which sends an `update()` event to all observers in the `Subject`'s observers list.
Observer aspect example (2 of 2)

< Subject, _aStateChange(..) >
< Observer, update(..), _start(.., Subject,..), _stop(..,Subject,..) >
CPs as reusable aspect designs

- CPs do not explicitly refer to any actual design elements they work with. As such, they may be composed with any design model.

```
binder < BookCopy, {meta:isQuery=false} >
< BookManager, updateStatus(), addView(), removeView() >
```

```
binder < Room, { book(), cancel() } >
< Reservations, update(), add(Room), remove(Room) >
```

```
binder < Subject, _aStateChange(..) >
< Observer, update(..), _start(.., Subject,..), _stop(..,Subject,..) >
```

```
binder < BookCopy, {meta:isQuery=false} >
< BookManager, updateStatus() addView(), removeView() >
```
Theme/UML within the software lifecycle

Compose the designs vs. map to an implementation language

- Specification of composition at design stage
- Two options for composition:
  - Implement individual designs – then compose
    - ✓ Traceability, with extensibility benefits
    - ✓ Reusable implementations
    - ✗ Need similar compositional language (e.g. AspectJ, Hyper/J)
  - Compose designs, and then implement
    - ✗ Lose traceability with code – inherent extensibility difficulties
    - ✗ Code is not reusable
    - ✓ Can write code using favourite OO language
Summary

- Current design approaches lead to design models that are difficult to extend, change and reuse.
- Theme/UML supports independent, reusable designs, including designs of crosscutting behaviour with composition patterns.
- CPs may be composed with non-pattern designs, and multiple designs may be composed together.
- Mapping Theme/UML decomposed models to compositional implementation languages supports:
  - traceability of the designs to code
  - ease of initial implementation and evolution of crosscutting and base code.