Fundamentals of Concern Manipulation

Talk Abstract

Harold Ossher
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10590, USA
+1-914-784-7975
ossher@us.ibm.com

ABSRACT

This talks describes a number of principles and key concepts underlying *concern manipulation*, the use of concerns to aid in a variety of software development tasks. Concern modeling and exploration, query and composition are considered. The principles and concepts guided work on the Concern Manipulation Environment (CME), which provides both prototype tools supporting aspect-oriented software development, and flexible components for use in building such tools.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features, D.2.3 [**Software Engineering**]: Coding Tools and Techniques, D2.13 [**Software Engineering**]: Reusable Software, D.2.2 [**Software Engineering**]: Design Tools and Techniques.

General Terms

Languages, Design.

Keywords

Aspect-oriented software development, separation of concerns, software queries, software decomposition and composition.

1. INTRODUCTION

As its name suggests, *concern manipulation* is about the use of concerns in any and all ways that are useful. This includes:

- Writing software that is modularized by concern.
- Identifying or mining concerns that were not modularized
- Modeling concerns and their relationships, and using these models to aid in development activities, such as assessing impact of change.
- Extracting concerns that are tangled with others.
- Composing concerns in flexible ways to yield full systems.

This talk is about general principles and key concepts of concern manipulation.

The principles were key considerations in the design and implementation of the Concern Manipulation Environment (CME) [5]. It provides both a set of prototype tools and a set of flexible components. The tools are for use during aspect-oriented development, and include a Concern Explorer for navigating and populating an underlying concern model [4], a query tool for searching for software elements using a variety of attributes and relationships [8], and a composition tool for composing concerns as guided by high-level, mostly simple specifications. The components are for tool builders to build upon and researchers to use for experimentation and prototyping. They include components for concern modeling [4], query [8], composition [7, 2] and related sub-activities. Extraction was planned but not implemented. The components are general and flexible, intended to be tailorable to a variety of AOSD approaches applied to a variety of different types of artifacts.

The CME is an open source project, though not currently under active development. It was developed as an Eclipse Technology Project, and is now available on SourceForge [1].

The rest of this abstract merely lists the principles and concepts covered, or alluded to, in the talk. In a few cases, it identifies architectural implications for tools aimed at supporting general concern manipulation. Further explanation and details, as well as discussion of and references to related work, are available in the referenced publications.

2. PRINCIPLES AND KEY CONCEPTS

This section begins with some general principles and concepts, and then discusses concerns, query and composition in separate subsections.

- The various concern-manipulation tools and components should provide a unified view and experience. This implies sharing of concepts wherever possible, such as regarding the body of software being worked on.
- The body of software being worked on is in a u*niverse* consisting of *container spaces* of *containers* made up of *elements.*
    - In the important special case of object-oriented software, the container spaces are *type spaces* (e.g., Java class paths), the con-

tainers are *types* (e.g., classes and interfaces), and the elements are *members* (methods and fields).

  - Containers that are referenced but are not to be manipulated themselves, such as Java library classes in most contexts, can be included in a special *library* container space, which is considered to be included in all container spaces.
  - All names used within a container space must be uniquely defined within that space (perhaps in the automatically-included library space). This is necessary for names to be properly understood and processed.

- The universe can, and usually does, involve software artifacts/elements of various kinds.
  - Architectural implication: Artifact-kind-specific code should be isolated, so that the bulk of the concern-manipulation support is generic.
- *Methoid*: a pattern identifying material inside element bodies, allowing the matching material to be treated as extractable methods for the purpose of identification, searching and composition. This allows support for code-level join points such as calls, throws and exception handler bodies.
- *Correspondence:* a tuple of *corresponding entities* (container spaces, containers, elements or methoids) that are to be composed with one another to form a composed entity. Correspondences identify join points in a symmetric way, and correspondence queries are the symmetric analogy of pointcuts.
- Each entity has *attributes,* which can be used in queries. When corresponding entities are composed, their attributes must be combined. Attributes include:
  - *Modifiers:* keyword attributes, e.g., "public."
  - *Classifiers:* modifiers that serve to classify their entities, e.g., "interface."

2.1 Concerns

- Concerns should be first-class entities, explicitly represented (modeled) and manipulable by users and tools.
- An underlying *symmetric* model should be used, with a convenient *asymmetric* façade available. Both symmetric and asymmetric scenarios [6] are important: some concerns are naturally peers, possibly freestanding, whereas others are naturally extensions or specializations of *base* concerns. This approach provides convenient, unified support for both. It is possible because asymmetric models are restrictions of symmetric models.
- An individual concern can be heterogeneous, involving artifacts/elements of multiple kinds.
- A concern has an *intension*, indicating the meaning of the concern, and an *extension*, the set of software elements that currently pertain to it. The intension might be expressed by a query. In the degenerate case, the intension can be merely a comment and the extension specified explicitly.
- Software can be written explicitly encapsulated in concerns, such as in modules or packages that represent concerns. Concerns can also be obtained by identifying or mining elements scattered across other concerns.
- A concern, unlike a container space, may contain names that resolve to definitions not included in the concern. In general, obtaining container spaces from concerns requires *extraction*, which must deal with such names (perhaps by including definitions, or *requires* declarations, within the space).

2.2 Query

- Queries are needed in many contexts, such as for exploration, definition of concern intensions, and correspondence identification for composition.
- Uniform query support should be available in all contexts, and the same query language(s) be usable throughout.
- Different query languages and underlying engines are appropriate for different AOSD approaches and experiments.
  - Architectural implication: Query languages and engines should be extensible and pluggable.
- Despite this variation, to provide the uniform support desired, a query language must provide at least the following capabilities:
  - Selection of elements based on names (including parameter signatures for methods), modifiers, classifiers, attributes and containment.
  - Selection of methoids, based on their patterns.
  - Selection of relationships, based on their names and characteristics of their end points.
  - Selection of correspondences: tuples of corresponding elements related as desired (e.g., having the same unqualified names in different scopes).
  - Navigation via relationships, including transitive closure.
  - Predicates and set operations.
  - Variables and unification. This is absolutely required for correspondence queries used for composition, and us useful in other contexts also.

2.3 Composition

- *Static composition* is sufficient to support dynamic join points and pointcuts. *Dynamic residue,* where the paradigm requires runtime tests (or other activities) to be performed at join points during execution, is handled by generating code to perform the appropriate tests and composing it statically at the right locations. This is, in fact, what aspect compilers typically do.

- Three composition levels are important, with different needs and tradeoffs:
  - *Concern assembly* level: the lowest level, at which the key issue is the nitty-gritty details of composing specific artifacts, such as Java class files.
  - *Reusable component* level: the middle level, at which the key issue is providing tool builders with flexible alternatives, allowing them to realize different composition paradigms.
  - *Tool* level: the highest level, at which the key issue is providing AOSD developers with convenient language constructs that support a particular paradigm.

2.3.1 Concern Assembly
Concern assembly involves some concepts specific to the low-level details of synthesizing composed artifacts from source artifacts:

- *Mapping* and *translation*, enabling a formal element, such as a method body, to be copied correctly from its source context to the composed context with proper name resolution.
- *Relationships* among elements, such as subtyping.
- *Method combination graphs,* specifying the details of how multiple, corresponding methods should be combined, including such issues as sequencing, exception handling and parameter mapping.
- Primitives for:
  - Container and element creation.
  - Mapping and relationship specification.
  - Copying and translating formal elements.
  - Generating code based on method combination graphs.

2.3.2 Reusable Composition Component
The CME composition component provides great flexibility by allowing composition to be specified in terms of the following concepts:

- *Weaving directive*s specify composition details.
- *What* elements are to be joined: correspondences.
- *How* elements are to be joined:
  - *Selection*, indicating which are to be included.
  - *Ordering,* specified by *combination graphs*.
  - *Structure,* specifying how the component elements are to be related in the composed result (e.g., facets of the same object, separate objects, separate object and aspect, etc.) [3].
- Making assumptions explicit:
  - *Encapsulation* indicates at what level name-matching is to be applied, if at all.
  - *Opacity* indicates whether class hierarchy structure is to be taken into consideration during composition, or if all classes are to be "flattened" before composition by having their inherited members explicitly included.
- Resolving multiple weaving directives that apply to the same element:
  - *Exclusivity* indicates whether multiple directives can cooperate to produce a single composed result, or whether just one must be selected.
  - *Precedence* determines the order of selection.

2.3.3 Tool-level composition
The concepts at the tool level are dependent on the paradigms (aspect languages or approaches) being implemented: the whole intent is that each tool be able to provide its own model and concepts. There is thus great variation at this level, but the following general concepts apply:

- Ideally, a composition tool should provide composition capabilities that are convenient and easy to understand. It need not necessarily provide the full flexibility of the lower levels, which are intended to be able to support multiple paradigms.
- Concerns should be first-class elements in composition specifications.
  - In general, obtaining container spaces needed for the lower levels of composition from concerns requires extraction, as noted earlier.
- For full integration with concern modeling, the composition specifications should be expressed as *composition relationships* between elements of the concern model.
- The composition specifications supported by the tool should be compiled down to the directives offered by the reusable composition component.
- *Dynamic residues* are handled at the tool level, since their details are paradigm-specific. The tool should generate methods that perform the desired runtime tests or other activities, together with directives causing the composition component to include them where appropriate.
- An *attribute rewriting system*, capable of transforming attributes of high-level composition specifications to those of mid-level weaving directives can provide some generic support for implementing diverse composition paradigms. The transformation is based on rules that (partially) define the paradigm.

3. CONCLUSION
This abstract described a number of principles and key concepts of concern manipulation. They were used in the design and implementation of the CME, but validation is limited due to the

limited number of tools built on the CME and limited experience obtained with them.

Follow-on research is an open area, including: validation and improvement of these concepts, exploration of alternatives and of design and implementation details, implementation of varied AOSD paradigms in terms of them, and exploration of new issues, such as handling of concerns containing artifacts that are versioned in an SCM system.

4. ACKNOWLEDGMENTS

The principles and concepts listed here were the product of joint work with Bill Harrison and Peri Tarr. Many of them derive from earlier work, by us or others. The CME was designed and implemented by a joint team from the IBM T.J. Watson Research Center and IBM Hursley Park. People involved at various stages of the project were: Matthew Chapman, Bill Chung, Andrew Clement, Adrian Colyer, Bill Harrison, Helen Hawkins, Sian January, Vincent Kruskal, Harold Ossher, Tova Roth, Stanley Sutton, Peri Tarr, and Frank Tip,

5. REFERENCES

This abstract refers only to our own detailed publications about the CME and the underlying concepts. Discussion of and references to related work can be found in each of them.

[1] CME web site: http://sourceforge.net/projects/cme/.

[2] William Harrison, Vincent Kruskal, Harold Ossher, Peri Tarr and Frank Tip, "Common Low-Level Support for Composition and Weaving." OOPSLA '02 Workshop on Tools for Aspect-Oriented Software Development.

[3] William Harrison and Harold Ossher, "Member-Group Relationships Among Objects." AOSD '02 Workshop on Foundations Of Aspect-Oriented Languages (FOAL).

[4] William Harrison, Harold Ossher, Stanley M. Sutton Jr., and Peri Tarr, "Concern Modeling in the Concern Manipulation Environment," ICSE '05 workshop on Modeling and Analysis of Concerns in Software (MACS '05).

[5] W. Harrison, H. Ossher, S. Sutton, P. Tarr, "The Concern Manipulation Environment – Supporting Aspect-Oriented Software Development." IBM Systems Journal 44(2): 309--318, 2005, special issue on Open Source Software.

[6] William Harrison, Harold Ossher and Peri Tarr, "Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition." Research Report RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December, 2002.

[7] William Harrison, Harold Ossher and Peri Tarr, "General Composition of Software Artifacts." In Proceedings of the 5th International Symposium on Software Composition (SC '06), March 2006, Springer, LNCS 4089.

[8] Peri Tarr, William Harrison, and Harold Ossher, "Pervasive Query Support in the Concern Manipulation Environment." IBM Research Report RC23343, 2005.