Feature-oriented programming

Tero Hasu
tero@ii.uib.no

INF329 course

16 April 2012

contents

- focus on feature-oriented *programming*
  - many methods and tools
    - we pick one and focus mostly on it...
    - ...but do mention others
- also: domain implementation and software generation
- not covered: other aspects of *feature-oriented software development* (FOSD)
  - no FODA or such

primary source material

- Batory et al: Scaling Step-Wise Refinement (2004)
  - a popular citation for *feature-oriented programming*
- Batory: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (2004)
  - for a more concrete programming and tooling angle

Don Batory

- www.cs.utexas.edu/~dsb/ (homepage)
- www.cs.utexas.edu/users/schwartz/search.cgi (publications)
- research on *product-line architectures* and *automated software development*
  - entails: "model-driven engineering, feature-based software designs, extensible software, adaptive software, software architectures, object-oriented design patterns, extensible languages, domain modeling, and parameterized programming"

domain engineering

- for systematic code reuse
- create reusable assets for the application domain
  - possible approach: architect feature-oriented systems so that can “instantiate” products with the desired feature set

Why feature-oriented? ☞

- requirements tend to be given in terms of features
- the customer is unlikely to be interested in what DLLs (or other components) you're using to construct your software
- the customer (hopefully) knows their requirements, and can see how a certain feature set satisfies said requirements

different flavors of “shrink-wrapped” software products also possible

- “entry-level through deluxe”
  - Windows 8 Enterprise Edition
  - Windows 8 Enterprise Eval edition
  - Windows 8 Home Basic Edition
  - Windows 8 Home Premium edition
  - Windows 8 ARM edition
  - Windows 8 Professional edition
  - Windows 8 Professional Plus edition
  - Windows 8 Starter edition
  - Windows 8 Ultimate edition

Feature-Oriented Programming (FOP)

- term apparently coined by Christian Prehofer in 1997
- but "feature-orientation" has been around for a while
  - particularly in the context of *software product lines* (SPL)
- FOP is the *study* of feature modularity and programming models supporting it ☜
  - "Feature modularity goes far beyond conventional notions of code modularity."

separation of concerns ✶

- *separation of concerns* is one of the most important principles in software engineering
- means decomposing software into manageable pieces along a dimension in concern space
  - abstractions like features and classes are viewed as dimensions in concern space
- consists of identification, encapsulation, and integration
  - *identification* means a software is decomposed into entities that represent the abstraction,
  - *encapsulation* means some mechanism is provided so that these entities can be manipulated separately, and
  - *integration* means that some composition mechanism is provided that integrates said entities

compositional vs. annotative feature-oriented systems

- there are *compositional* and *annotative* approaches
  - "Compositional approaches for implementing features represent features as distinct modules, which are composed at compile time or deployment time or similar." ✷
  - "Annotative approaches implement features by identifying code belonging to a feature in the source and annotating it, so that variants may be created by including or removing annotated code from the source." ✷
- Sunkle et al believe that "by using a combination of compositional and annotative approaches, we can create a better representation of features" ✷

Programming with Feature Orientation (PFO)

- we don't consider the annotative approach as "true" FOP, as the code doesn't have feature-oriented structure
  - although: What about IDEs (like CIDE) that have first-class support for annotated features?
- annotative domain implementation can still be a part of a FOSD process
  - when so, we call it *programming with feature orientation* (PFO)

PFO with CPP conditionals

- `#if` probably the most common solution in the industry
  - even with commercial FOSD tooling such as pure::variants
- `#if` style annotations can be used without architecting or refactoring for modularity or explicit parameterizability
  - code may become messy as the number of features increases
  - feature implementations are spread around codebase, hard to see as a whole or reuse
    - but tools can help in viewing and analysis

other language support for PFO

- e.g. color annotations in CIDE
  - good for “featurizing” legacy codebases
- e.g. rbFeatures
  - Günther & Sunkle: rbFeatures: Feature-Oriented Programming with Ruby (2011)

How extensive should feature-oriented structuring support be?

- ideally across the code base (all languages)
  - general-purpose programming language code (both static and dynamic), resource files, makefiles
- ideally with statement and expression level feature-specificity (not just module, class, or function level)
  - problem: statements and expressions are (normally) not named

What language support does FOP require?

- at least *some* support for modularity required
  - parameterizable modules or classes, classes with inheritance, mixins, concepts, ...
- AOP style code insertion may be useful to adapt existing "base code" for a feature
- if necessary, first-class feature module support can be added e.g. through source-to-source translation
  - Batory talks of *precompilation* or *preprocessing*
- even highly dynamic and reflective languages may not be easy to adopt for FOP
  - see Günther & Sunkle: Enabling Feature-Oriented Programming in Ruby (2009)

the library scaling problem

- just e.g. having "traditional" parameterizable (non-feature) modules may not be that practical
  - i.e. when instantiating a module specify concrete implementations of all types and functions that have variability
- if your components are large they're probably too specific; if they are small and highly parameterizable people must write a lot of code to instantiate and compose them
  - Biggerstaff: The Library Scaling Problem and the Limits of Concrete Component Reuse (1994)

Solutions for the library scaling problem?

- perhaps: features should allow for adding new components and for cross-cutting refinement of (multiple) existing components
  - no advance parameterization: applied "externally"
  - inheritance allows for *before*, *after*, and *overriding* "advice" of methods ☛
  - but may have to identify *join points* for some artifact types
    - e.g. XML documents: How to refine elements? Which ones? How to identify them?—Anfurrutia et al: On Refining XML Artifacts (2007)
- Batory: we need a combination of building blocks and generative techniques

RQO: a “spectacular example” of futuristic software engineering ☛

- *relational query optimization* (RQO)
- SQL is a *domain-specific language* (DSL)
  - a declarative language for retrieving data from tables
  - an SQL compiler translates an SQL statement into a relational algebra expression
- a query optimizer realizes *automatic programming* (AP) by applying equational rewrite rules
- the back end does *generative programming* (GP) by translating the optimized expression into an efficient program

AHEAD (Algebraic Hierarchical Equations for Application Design) ☛

- a *theory* of feature-oriented programming
- aims to generalize the success of RQO to other domains
- direct successor and generalization of GenVoca

AHEAD Tool Suite (ATS)

- http://www.cs.utexas.edu/users/schwartz/ATS.html
- a suite of tools that *implement* the AHEAD theory

step-wise refinement (SWR) ☛

- a methodology that can serve as a basis for a powerful form of FOP
- a simple and ancient idea: construct complex programs from simple ones by incrementally adding details
- if the increments are features, the SWR becomes FOP
  - This is the starting point of AHEAD.

feature refinement

- a *feature refinement* adds a feature to a program
  - a module that encapsulates an individual feature
  - may e.g. encapsulate *fragments* of multiple classes
    - then the refinement *cross-cuts* those classes
- feature refinements are composed using *generators* to synthesize code for a full program
- one feature refinement might consist e.g. of a set (or sequence) of class refinements

implementing class refinement

- refinements must be realized in code somehow
- e.g., a *class refinement* refines a class by e.g. introducing new methods and extending or overriding existing ones
- How to represent a class refinement? Want it as a separate, modular fragment.
  - can implement e.g. based on so-called *mixin inheritance*
    - i.e. the superclass of a class is parameterized
    - one problem: a mixin doesn't assume the name of its superclass, so cannot add to a class (cf. *open classes*)—can be addressed via generative techniques
  - some languages (e.g. Ruby) support "mixing in" mixins without inheritance

a mixin in C++

```
template <class Graph>
class Counting : public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : nodes_visited(0), edges_visited(0), Graph() { }
  node succ_node(node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }
  ...
};
```

Smaragdakis & Batory: Mixin-Based Programming in C++ (2000)

synthesizing classes

- one we have feature specific class fragments represented as mixins or whatever, can have tooling synthesize a concrete class that has the desired name and mixes in all the fragments required for the desired feature set
- in the mixin inheritance case, only *terminal classes* of the *refinement chains* are instantiated

Should features be *first class* rather than *design patterns*?

- many techniques are used to implement features ✷
  - the main kind of concern supported by them is one of functions, classes, aspects, hyperslices, mixins, and frames, etc.
    - features, which are themselves a kind of concern, are essentially implemented in terms of entities that basically represent some other kind of concern
- this abstraction and representation mismatch causes problems such as hierarchical misalignments, limitations in feature composition and order, and inexpressive program deltas ✷

FOP languages

- Jak
- FeatureC++
  - Apel et al: FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++ (2005)
  - Apel et al: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming (2005)
- XAK
  - Anfurrutia et al: On Refining XML Artifacts (2007)

FOP language implementation support

- FeatureHouse
  - provides an easy-to-use plug-in mechanism for new languages, based on attribute grammars ✻
    - for preparing languages for implementing and composing features
  - Java, C#, C, Haskell, etc. have been plugged in
  - Apel et al: Language-Independent and Automated Software Composition: The FeatureHouse Experience (2012)

Jak

- short for Jakarta
- an extended and extensible Java
  - extended with first-class feature support
    - `feature`, `refines`, `Super`
  - extended with meta programming support
  - extended with language for state machines
- bootstrapped: implemented based on a toolkit implemented in Jak
- Jak-to-Java compiler included in ATS

constructing and composing code fragments

```
import jak2java.*;

class ex1 { // example from ATS documentation
  public static void main( String args[] ) {
    AST_Modifiers m = mod{ public final }mod;
    AST_Exp e = exp{ i+1 }exp;
    AST_FieldDecl f = mth{ int i;
      int inc( int i ) { return $exp(e); } }mth;
    AST_TypeNameList t = tlst{ empty }tlst;
    AST_QualifiedName q = id{ foo }id;
    AST_Class c = cls{ interface empty{};
      $mod(m) class $name(q)
      implements $tlst(t) { $mth(f) } }cls;
    System.out.print(c);
  }
}
```

class refinement in AHEAD and Jak: a class ☞

```
feature Base;

class B {
  int x;
}
```

- a base artifact (here: a class) is a *constant* in the AHEAD algebra

a class refinement ☜

```
feature Customization;

refines class B {
  int y;
  void z() {...}
}
```

- a refinement (here: a class refinement) is a function mapping artifacts
  - a single-argument function (no multiple inheritance)

composition in a flattened form ☜

```
class B {
   int x;
   int y;
   void z() {...}
}
```

composition by jampack ☜

```
feature Program;

class B {
  int x;
  int y;
  void z() {...}
}
```

equivalent *refinement chain*

class B_P {
  int x;
}

class B_R **extends** B_P {
  int y;
  void z() {...}
}

class B **extends** B_R {}

- recall INF220

composition by `mixin`

```
feature Program;

SoUrCe Base "Base/B.jak";
abstract class B001 {
  int x;
}

SoUrCe Customization "Customization/B.jak";
public class B extends B001 {
  int y;
  void z() {...}
}
```

a “program” may be something more than a set of classes ☛

- a full system (of multiple programs and libraries) with associated knowledge representations (e.g., process models, UML models, makefiles, design documents, etc.)
- AHEAD specifies an algebraic model of application synthesis that treats *all* representations in a uniform way: code and noncode, individual programs, and multiple programs

a containment hierarchy of artifacts

- classes in a package
- packages in JAR files
- JAR files in a program
- programs in an application suite

Principle of Uniformity

- Impose an object-based structure on artifacts of a given type, taking advantage of any natural indexing scheme that may already exist, and define refinement to follow the notions of mixin inheritance (or more specifically, class refinement).
  - many artifact types have an object-based structure, although few support inheritance
    - a refinement operation realizing mixin inheritance must be implemented for AHEAD support

makefile example

- **see** makefile refinement and composition (Figure 5)
- impose an object-based structure on makefiles ☛
- natural indexing scheme: targets are uniquely named ☛
- cf. targets with actions vs. methods with statements
- `super` references expanded by textual substitution
    - Is this strictly necessary?

the algebra of AHEAD ☛

- AHEAD talks of *units*: either *constants*, *functions*, or *collectives*
- *units* may be grouped into *collectives*
  - a single feature may consist of multiple constants (base artifacts) or functions (refinements)
- composition of units is defined by the laws of *inheritance*
- composition is recursive (as is the definition of units), pairwise according to the names of units
- the composition operator • is polymorphic on the artifact type

recursive composition

$$\begin{aligned} h \bullet f &= \{a_h, b_h, c_h\} \bullet \{a_f, b_f, d_f\} \\ &= \{a_h \bullet a_f, b_h \bullet b_f, c_h, d_f\} \end{aligned}$$

equation file

- a synthesization specification
- `Program.equation`:

  Base
  Customization

- `composer --target=Program Base Customization`
- $\mathit{Program} = \mathit{Customization} \bullet \mathit{Base}$
- Principle of Uniformity here, too. An equation file may be a refinement, and may use the `super` keyword to refer to the refined equation.

ATS functionality ☛

- collective implemented as a file system directory
  - feature composition is directory composition
- `composer` takes an equation, and creates a composite feature directory (named after the equation), invoking artifact-specific composition tools
- `jampack` and `mixin` are alternative composition tools for Jak files
- `unmixin` back-propagates updates made to `mixin` generated files
- `jak2java` translates Jak into Java

AHEAD compared to RQO ☛

- programs with the desired feature set are specified as expressions in a *domain-specific language* (DSL) of sorts
  - in `.equation` files
- *automatic programming* (AP) is realizable in theory as can pick from multiple implementations of a feature, and AHEAD expressions can be optimized
  - Batory et al: Design Wizards and Visual Programming Environments for GenVoca Generators (2000)
- ATS includes tools for *generative programming* (GP): modules implementing features can be composed by synthesizing code required for a complete program

FeatureIDE

- FeatureIDE Eclipse plugin for FOSD
- "supports all phases" of FOSD
- includes:
  - Feature Model Editor (graphical and text based)
  - Constraint Editor (constraint checking, content assist, etc.)
  - Configuration Editor (for creating and editing configurations, with support for deriving valid ones)
- supports AHEAD (in addition to FeatureC++, FeatureHouse, etc.)
  - Jak language aware editing with refactorings, etc.

FeatureIDE with AHEAD scenario

- **a possible project organization for a pure Java project**
- define your feature model in a `.m` file
  - can edit dependencies and constraints graphically
- have an `.equation` file for each product configuration
  - editor support for ordering / optional auto-ordering of refinement chains
- implement classes as `.jak` files
  - one file per feature involving said class
  - different directory for each feature used to store files
  - usual assisted editing (as for Java in Eclipse)

GUI calculator example

- **see** GUI calculator (Figure 20)
- addition and subtraction features of a graphical calculator

AHEAD in use

- AHEAD is being used to build next-generation distributed fire support simulators (FSATS) for the US Army Simulation, Training, and Instrumentation Command (STRICOM).
- Bootstrapping AHEAD itself. As mentioned earlier, AHEAD tools were initially built using JTS. To bootstrap AHEAD, JTS source was converted into AHEAD features.

feature interaction *

A *feature interaction* is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation.

feature Base;
class List {}
class Node {}

feature interaction *

```
feature Single;
refines class List {
   Node first;
   void prepend(Node n) {
      n.next = first; first = n;
   }
}
refines class Node { Node next; }
```

feature interaction *

```
feature Reverse;
refines class List {
   Node last;
   void append(Node n) {
      n.prev = last; last = n;
   }
}
refines class Node { Node prev; }
```

feature interaction *

```
class List {
  Node first;
  void prepend(Node n) {
    n.next = first; first = n;
  }
  Node last;
  void append(Node n) {
    n.prev = last; last = n;
  }
}
class Node {
  Node next; Node prev;
}
```

references

- Apel and Kästner: An Overview of Feature-Oriented Software Development (2009) ✻
  - an FOSD survey
- Sunkle et al: Features as First-class Entities – Toward a Better Representation of Features (2008) ✷

further reading

- Batory et al: JTS: Tools for Implementing Domain-Specific Languages (1998)
  - info on Jak and the associated Jakarta Tool Suite (JTS)
    - JTS is a *domain implementation* for producing extended industrial PLs and component-based generators
- Batory et al: The Objects and Arrows of Computational Design (2008)
  - about AHEAD etc., for the categorically inclined
- Prehofer: Feature-Oriented Programming: A Fresh Look At Objects (1997)
  - highly sited for FOP (coined the term?)

further reading and listening

- Thüm et al: Applying Design by Contract to Feature-Oriented Programming (2012)
  - presented at FASE 2012
  - correction: discusses integrating *design by contract* with FOP
    - if it's not hard enough with just OO
- www.fosd.de
  - for links to lots of FOSD tools and material
- Feature-Oriented Software Development with Sven Apel (Software Engineering Radio episodes 172 & 173)
  - easy listening