Granularity in Software Product Lines

Christian Kastner, Sven Apel and Martin Kuhlemann

Kattiana Fernandes Constantino

Reutilização de Software
DCC / ICEx / UFMG

Summary

- Introduction
- Approaches to SPL Implementation
- Granularity
- Case Studies
- CIDE
- Conclusion

Introduction

- A software product line (SPL) aims at generating tailored programs from a set of features.
- Using SPLs it is possible to create program families of related programs for a domain.
- They explored effects of granularity of different approaches on SPL Development.
- Presenting an tool called Colored IDE (CIDE).
- Two case studies: ***Berkeley DB*** (legacy software) and ***The Graph Product Line*** (SPL).

Approaches to SPL Implementation

- The Compositional Approach

```
class Stack {
  boolean push(Object o) {
    elementData[size++] = o;
  }
}

refines class Stack {
  boolean push(Object o) {
    lock l=lock(o);
    super.push(o);
    l.unlock();
  }
  lock lock(Object o) { /*...*/ }
}

refines class Stack {
  boolean push(Object o) {
    super.push(o);
    log("added " + o);
  }
  void log(String msg) { /*...*/ }
}
```

Figure 1: A basic stack and two features implemented in Jak.

Approaches to SPL Implementation

- The Annotative Approach

```
static int __rep_queue_filedone(dbenv, rep, rfp)
  DB_ENV *dbenv;
  REP *rep;
  __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
  COMPQUIET(rep, NULL);
  COMPQUIET(rfp, NULL);
  return (__db_no_queue_am(dbenv));
#else
  db_pgno_t first, last;
  u_int32_t flags;
  int empty, ret, t_ret;
#ifdef DIAGNOSTIC
  DB_MSGBUF mb;
#endif
  // over 100 lines of additional code
#endif
}
```

Figure 2: Code excerpt of Berkeley DB.

Granularity

- Compositional Approaches
  - Existing compositional techniques usually allow coarse-grained extensions only.
  - They noticed several limitations when implementing fine-grained extensions compositionally.

Granularity

- Annotative Approaches
  - Conceptually, annotations can mark code fragments at arbitrary levels of granularity. They simply introduce markers at the exact positions that should be extended.
  - Annotative approaches support fine-grained extensions better than compositional approaches.

Case Studies

- Berkeley DB
  - 84.000 LOC
  - It was not designed as an SPL with features in mind.
  - It was implemented as a single application with a clean object-oriented design modularized in classes and packages.
  - Some larger features like Synchronization and Transaction affected up to 30 (of 300) classes in Berkeley DB in over 150 places.

Case Studies

- Berkeley DB
  - Decomposition with Aspect J
    - They were confronted with fine-grained extensions in almost every feature.
    - Of 1,144 extensions used to implement the 38 features, 640 extensions (56%) introduced new classes, methods, or fields.
    - 214 other extensions (19%) were simple method extensions that added wrappers to existing methods. They were well supported by AspectJ.
    - However, 261 extensions (23%) were required at statement level and 24 (2%) at expression level, which posed major problems.
    - They experimented with different possibilities to implement the ***Transaction feature***.

Case Studies

- Berkeley DB
  - Decomposition with CIDE
    - The decomposition of Berkeley DB was more convenient and much faster (3 days instead of 1 month).
    - By using CIDE, understanding the SPL became simpler, because neither do additional statements obfuscate the source code, nor is it necessary to understand complex workarounds.

Case Studies

- The Graph Product Line
  - 2.000 LOC
  - It was designed from scratch as an SPL and suggested as a benchmark for SPL technologies.
  - The original version of GPL was implemented with mixin layers, but other implementations e.g. in AspectJ and Hyper/J are available.

Case Studies

- The Graph Product Line
  - Decomposition with Mixin Layers
    - The granularity required to implement GPL is coarse.
    - The most features introduce only new code fragments (25 methods and 21 fields into existing classes and 18 new classes). Methods were rarely extended (by means of method overriding). Of 84 methods in GPL only 4 were ever extended.
    - They found some code replication that could have been avoided if fine-grained extensions were available.

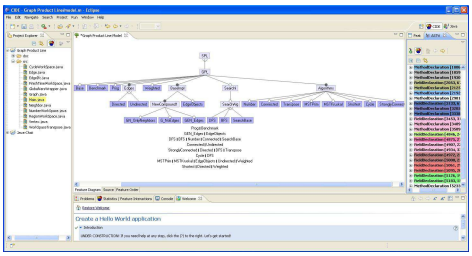
Case Studies

- The Graph Product Line
  - Decomposition with CIDE
    - They were surprised that despite GPL's apparent coarse granularity, we could still identify several situations, in which we benefit from CIDE's fine granularity and could reduce code replication.
    - Because of reduced code replication and because we do not need implementation overhead of mixin layers, the code size of GPL in CIDE is 36% smaller than in the original implementation (1,222 LOC instead 1,920 LOC).

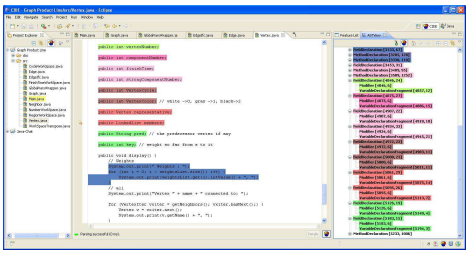
CIDE

- An Eclipse-based prototype tool for decomposing legacy applications into features that may have a fine granularity.
- Because of its colorful appearance, this tool is called ***C**olored **I**ntegrated **D**evelopment **E**nvironment* (**CIDE**).

CIDE



CIDE



Conclusion

- Compositional approaches do not support fine-grained extensions, so that workarounds are required which raise the implementation's complexity.
- In contrast, annotative approaches can implement fine-grained extensions but introduce readability problems by obfuscating the source code.
- CIDE is a tool developed in order to simplify SPL development.

Conclusion

- CIDE is based on preprocessor semantics but uses background colors instead of source code statements and by providing the possibility to hide features it avoids obfuscating the code.
- CIDE supports developers in understanding features with navigation and projection facilities and the possibility to export the SPL into distinct feature modules.

Conclusion

- In two case studies they showed the advantage of CIDE over existing compositional approaches.
- It was possible to implement fine-grained features including statement and expression extensions and even signature changes without resorting to workarounds.
- With CIDE they could implement additional features that were not reasonably possible with compositional approaches and reduce code replication of earlier implementations.

Questions