17-708 SOFTWARE PRODUCT LINES: CONCEPTS AND IMPLEMENTATION

FEATURE INTERACTIONS

CHRISTIAN KAESTNER
CARNEGIE MELLON UNIVERSITY
INSTITUTE FOR SOFTWARE RESEARCH
READING
 ASSIGNMENT NOV 16

tbd
LEARNING GOALS

Understand the nature of feature interactions and the optional feature problem; disentangle the different meanings of the terms

Identify common sources of feature interactions

Use appropriate strategies to avoid, mitigate, or detect feature interactions

Select and apply implementation strategies for the optional feature problem
Today's weather: [:weather:]
Hands-free entry
Night lock
Electronic operation
Intruder defense
...

Example from P. Zav
Feature Interactions

Features designed in isolation (divide and conquer)

Interact in intended and unintended ways when composed

(Failure of compositionality due to hidden underlying domain)
The physical world has no compositionality."
M. Jackson, FI Dagstuhl 2014
SOURCES

Overlapping preconditions (nondeterminism)
Requirements inconsistency
Conflicting goals
Violations of assumptions
Resource contention

INTERACTION TYPES

Nondeterminism
Dependence
Override (same precondition)
Negative impact (same precondition)
Override (linked trigger events)
Negative impact (linked trigger events)
Order
Bypass
Infinite loop

Figure 2: Categorization of approaches to addressing feature interactions.
Handling Interactions

Composition mechanism with default resolution

(telecommunication, Android, home automation)
Handling Interactions

Isolation, noninterference  (to some degree)

(Android, Kernel drivers, browser plugins)
Handling Interactions

Detection in requirements or implementations

```java
public void timeShift() {
    if (Configuration.overload){
        if (areDoorsOpen() && weight > maximumWeight){
            if (Configuration.verbose) {
                System.out.println("Elevator blocked due to overloading (weight: "+ weight + ")");
            }
            return;
        } else {
            blocked = false;
        }
    }
    if (stopRequestedAtCurrentFloor()){
        doors = DoorState.open;
        // Iterate over a copy of the original list, avoids concurrent
        // modification exception
        for (Person p : persons){
            if (!p.isDestinationfloorsId() == currentFloorId){
                leaveElevator(p);
            }
        }
    }
    if (doors == DoorState.open) {
        doors = DoorState.close;
    }
    if (stopRequestedToDirection(currentHeading, true, true)){
        continueInDirection(currentHeading);
    } else if (stopRequestedToDirection(currentHeading.reverse(), true, true)){
        continueInDirection(currentHeading.reverse());
    } else if (idle){
        continueInDirection(currentHeading);
    }
}
```

```
#include <stdio.h>

#ifdef WORLD
    char * msg = "Hello_world\n";
#else
    char * msg = "Bye_bye!\n";
#endif

main() {
    printf(msg);
}
```
DETECTION

Formal methods

   Model checking

   Detecting overlapping preconditions

   ...

Requirements

   Identifying shared resources
- At the requirements level, a typical strategy is to systematically search for shared resources. Two features that share resources may potentially interact over this resource. For example, the features FireControl and FloodControl from Example 9.2 both affect the resource water supply. A typical strategy is to model all resources relevant for each feature and subsequently investigate manually all pairs of features that share a resource.

- The strategy applied to resources can be used also for events (and preconditions of operations). Two features that react to the same event (or that have overlapping preconditions) are potential candidates for feature interactions. For example, the features CallForwarding and CallWaiting from Example 9.1 both react to the same event (that is, an incoming call on a busy line). Again, modeling events allows us to manually investigate all pairs of features reacting to the same event.

- Inconsistent requirements and conflicting goals of features revealed during domain engineering can also be an indicator of potential interactions. For example, features Acceleration to increase the speed of a car and AdaptiveCruiseControl to automatically adjust the distance to other cars by decreasing speed have conflicting goals. Again, requirements and goals need to be made explicit, for example, by modeling them.

- Making assumptions (or invariants) of features explicit can help detecting when an assumption is violated by other features. For example, feature Index in Example 9.3 assumes that the data structure is immutable, an assumption violated by feature Write.
ONLINE TECHNIQUES

- *Feature Manager* based approaches. An entity, usually called the feature manager, is introduced into the network with the capability of observing and controlling the call processes. Hence, the control of the call is located with the feature manager. So far mainly centralised approaches featuring a single feature manager have been developed. However, distributed architectures for managers are also possible.

- *Negotiation* based approaches. Individual features have the capability of communicating their intentions to each other and negotiating an acceptable resolution. Most approaches advertise a direct communication where the call control resides with the features. However, if no resolution is possible the conflict can be forwarded to a third party to resolve.

OPTIONAL FEATURE PROBLEM

Code focused view – how to implement coordination code between two features

Applicable only once interactions identified and detected
Statistics (buffer hit ratio, table size and cardinality, …)

Transactions (locks, commit, rollback, …)

Transactions per second
PRODUCTS

DB with statistics and transactions

DB with statistics without transactions

DB with transactions, without statistics
UNDESIRED PRODUCTS

DB with transactions without statistics measuring transactions per second (larger and slower than necessary)

DB with statistics without transactions trying to measure transactions per second(?)
How to create product without transactions but with statistics

transactions/sec really a feature?

Where to implement transactions/sec
DOCUMENT DEPENDENCIES

Variability  Impl. Effort  Binary Size & Perf.  Code Quality

Impossible products
MOVING SOURCE CODE

Variability

Impl. Effort

Binary Size & Perf.

Code Quality

Products

Inefficient
CHANGE BEHAVIOR

Variability

Impl. Effort

Binary Size & Perf.

Code Quality

Products

Missing Behavior

Variability

✓

Impl. Effort

✓

Binary Size & Perf.

✓

Code Quality

✓
PREPROCESSOR

not modular

Variability

Impl. Effort

Binary Size & Perf.

Code Quality

Products

#ifdef TXN
lock();
#ifdef STAT
lockCount++;
#endif
#endif
GLUE CODE MODULES

- Variability
- Impl. Effort
- Binary Size & Perf.
- Code Quality

Products

extra module necessary

34
## OVERVIEW

<table>
<thead>
<tr>
<th>Lösung</th>
<th>Variability</th>
<th>Effort</th>
<th>Size &amp; Perf.</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple impl</td>
<td>✔️</td>
<td>✭</td>
<td>✔️</td>
<td>✭</td>
</tr>
<tr>
<td>Dependency</td>
<td>✭</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Move source code</td>
<td>✔️</td>
<td>✔️</td>
<td>✭</td>
<td>✔️</td>
</tr>
<tr>
<td>Change behavior</td>
<td>✭</td>
<td>✔️</td>
<td>✭</td>
<td>✔️</td>
</tr>
<tr>
<td>Preprocessor</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✭</td>
</tr>
<tr>
<td>Glue code modules</td>
<td>✔️</td>
<td>✭</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
EXPERIENCE BERKELEY DB

Dependencies?
- Would render important features de-facto mandatory

Change behavior?
- undesired

Glue code module?
- Extracted 76% of statistics code into 9 modules
- → possible but labor intensive

Preprocessor
- Faster, easier
- Scattered code
Scale
REMINDER: VARIABILITY MANAGEMENT

see mass customization in automotive

Identify relevant variability

Reduce unnecessary variation to avoid interactions and optional code problems