feature-oriented software development

stakeholders’ mental model of system

feature-oriented software system
what is a feature?

“unit of functionality” [Hall, 2000]
- additive / incremental
- optional

“a modification of or an addition to a service, and does not stand on its own” [Bellcore1992]

“a tariffable unit” [Bellcore1992]

“a unit of variability”

“a requirement” [Zave2015]

“a characteristic / property / quality / aspect / profile / concern of the system”
feature-oriented software development

**feature**: a requirement

stakeholders’ mental model of system

feature-oriented software system

Cruise Control
Stability Control
Anti-Theft

Cruise Control
Stability Control
Anti-Theft
Cruise Control
feature interactions

feature interaction: a feature behaves differently in the presence of another feature than it behaves in isolation

› nondeterministic
› conflicting actions
› violate global correctness property
› emergent behaviours
hybrid brakes ⊕ anti-lock breaking

2010 Toyota Prius

hybrid brake system
  › (normal) hydraulic brake system
  › regenerative braking system
    – converts loss of vehicle momentum into electrical energy
    – stored in on-board batteries

anti-lock brake system (ABS)
  › maintains stability, steerability during panic braking

interaction
  › braking force after ABS actuation reduced
  › vehicle stopping distance is increased
  › 62 reported crashes, 12 injuries
cruise control \(\oplus\) traction control

**cruise control**
- vehicle set to maintain driver-specified speed

**traction control**
- brake fluid applied when wheels slip

**interaction**
- engine power is increased (to maintain speed)
- driver senses “sudden acceleration”
  - vehicle becomes difficult to control

**resolution**
- advise drivers not to use cruise control on slippery roads
not all interactions are bad!

intended interactions
› advanced cruise-control variants override basic cruise control
› prohibit navigation overrides navigation
› prohibit-navigation override overrides prohibit-navigation

unintended but harmless interactions
› call screening prevents activation of caller id

(planned) resolutions to conflicts
› brake override overrides (acceleration + braking)
all interactions require work

• verify *intended* interactions
• detect *unexpected* interactions
• analyze them for *undesired* interactions
• fix undesired interactions
  – faulty feature
  – disallow feature combination
  – resolve interaction using exceptions / new features
• test the fixes
feature modularity vs. interactions

FOSD emphasizes features
de-emphasizes interactions
but still must detect, analyze, fix, test interactions

this is exactly the chore that feature-orientation was meant to avoid!
what this talk is about

history of feature modularity

 › representing features as isolated modules
 › (some) information hiding

addressing feature interactions

 › scalable to large numbers of features
 › supports third-party development
approach #1:

feature module = co-location of feature’s code
interactions resolved using exceptions

\[ F_1 = f_1 + e_{f_2} + e_{f_3} + \ldots + e_{f_n} \]

\[ + e_{f_2f_3} + e_{f_3f_4} + \ldots \]

\[ + e_{f_2f_3\ldots f_n} \]
lots of features

e.g., telephony has 1000+ features per system

a system of feature-rich systems

› features from multiple providers
› multiple active versions of the same feature
lots of interactions

results of the second feature interaction contest
lots of types of interactions

**control-flow**
one feature affects the flow of control in another feature

**data-flow**
one feature affects (deletes, alters) a message destined for another feature

**data modification**
shared data read by one feature is modified by another feature

**data conflict**
two features modify the same data

**control conflicts**
two features issue conflicting actions

**assertion violation**
one feature violates another feature's assertions or invariants

**resource contention**
the supply of resources is inadequate, given the set of competing features
introduced in several phases
Bowen, SETSS’89

[req] understanding / specifying how features ought to interact

[req] the number of interactions (and resolutions) to consider grows exponentially with the number of features

[design] more interactions introduced during design due to sharing of resources, I/O devices, protocol signals, etc.

[imp] near-commonalities among features leads to questions about how to effectively reuse software components

[test] the sheer number of interactions and resolutions to be tested lengthens the testing phase
feature interaction problem

death by exceptions [Zave]

\[ F_1 = f_1 + e_{f_2} + e_{f_3} + e_{f_4} + e_{f_5} + e_{f_6} + e_{f_7} \]

\[ + e_{f_8} + e_{f_9} + e_{f_{10}} + e_{f_{11}} + e_{f_{12}} \]

\[ + e_{f_{13}} + e_{f_{14}} + e_{f_{15}} + e_{f_{16}} + \ldots + e_{f_n} \]

interactions

• dominate feature development
• complicate third-party feature development
approach #2:

feature module = black box
feature modules

features are independent entities that are oblivious to the presence of other features

- each module’s interface is simply its inputs and outputs to a shared environment

• enables third-party feature development
• simplifies detection of interactions
detecting feature interactions

\[
\begin{align*}
F_1 & \models \Phi_1 \\
F_2 & \models \Phi_2 \\
\vdots \\
F_n & \models \Phi_n \\
\end{align*}
\]

\[
F_1 \oplus F_2 \oplus \cdots \oplus F_n \not\models \Phi_1 \land \Phi_2 \land \cdots \land \Phi_n
\]
call forward vs. voice mail

CFNA ⊨ call is forwarded to new directory number
VM ⊨ message is from the caller is recorded

? CFNA ⊕ VM ⊨ forward call ∧ record message
nonmonotonic resolutions
(Veldhuijsen’95)

adding a new feature can change the requirements of existing features:

• nonmonotonic extensions
  – e.g., Freephone changes billed party

• changes to definitions of terms
  – e.g., refinement of the notion of being busy
  – e.g., evolution of a call
  – e.g., evolution of directory numbers; of private numbers

• violation of invariants / assumptions
  – “I have not been able to think of a single interesting assertion that would be true of a system incorporating all [features of the public switched telephone network].” [Zave’01]
feature interaction problem

dead by “interaction features”

$$F_1 + F_2 + F_3 + ... + F_n$$

$$+ F_1#F_2 + F_1#F_3 + ... F_{n-1}#F_n$$

$$+ F_1#F_2#F_3 + F_1#F_2#F_4 +$$

$$+ F_1#F_2#F_3#...#F_n$$

aiming for perfect resolution of interactions doesn’t scale
X calls Y, which forwards the call to Z, and the call attempt fails.

whose VM should react?

- what if Y is a sales group and Z is a sales representative?
- what if Y is on a long leave of absence?
approach #3:
feature modules with interfaces
interfaces and information hiding

**interface** advertises what services a module provides to the rest of the system, and how they can be accessed.

**information hiding** encapsulates a design decision inside a module, whose interface reveals only externally visible properties [Parnas’72]
**interfaces**

**feature interface** defines what services a feature provides to the rest of the system and how other features can access those services.

- **public interface**
  - inputs / outputs
  - less expressive
  - accessors
  - mutators

- **family interface**
  - extension points
  - more expressive
feature families

CruiseControl
SpeedLimitCC + CC#SLCC
CurveCC + CC#CCC + SLCC#CCC + CC#SLCC#CCC
HeadwayControl + CC#HC + SLCC#HC + CCC#HC…

Anti-theft
Climate Control
Lighting
Power Windows
minimal public interface

most inter-feature references are to high-level common modes of operation

examples

FeatureX_Fail flag is set to true when FeatureY is in fail state

FeatureQ is enabled only if FeatureP is enabled
minimal public interface

most inter-feature references are to high-level common modes of operation
generic public interface

modes of operation can serve as criterion for structuring feature modules
generic public interface (2)
2 strategies for resolving interactions

**feature families**
resolve interactions within a feature family *perfectly* (e.g., new “interaction” features)

**unrelated features**
handle unexpected interactions between unrelated features using default resolution strategies
resolving interactions between unrelated features
feature coordination

given a collection of feature families, each with a minimal public interface...

... want to coordinate the features’ executions such that entire classes of interactions are resolved by default
serializing features

Distributed Feature Composition [Jackson, Zave, TSE’98]

pipeline architecture

+ features make no assumptions about other features
+ avoids simultaneous reactions to the same event
+ conflicts are resolved through serialization
+ feature ordering realizes a priority scheme
- resolution is implicit
parallel execution (resolution modules)

+ features make no assumptions about other features
+ conflicting actions are resolved by resolution modules
+ all actions are considered in resolution
+ resolution strategies can be variable-specific
feature coordination

- fixed set of features
  - pre-determined selection of features
  - static integration
  - perfect coordination possible

- fixed set of features
  - semi-configurable selection of features
  - set of static integrations
  - perfect coordination possible, but impractical

- unlimited features
  - user-defined selection of features
  - dynamic integration
  - loose coordination
modular features + tight integration of feature families + (looser) coordination of unrelated features

- no interfaces
- perfect resolution
- feature interfaces
- default resolution
- relax "correctness"