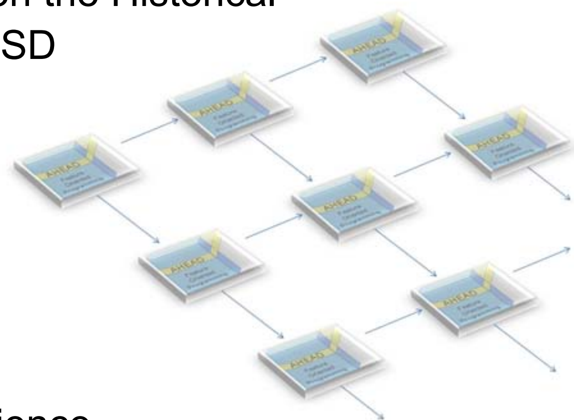
FOSD – A Science of Software Design

A Personal Perspective on the Historical Development of FOSD



Don Batory
Department of Computer Science
University of Texas at Austin
January 2011

Introduction

- Organizers asked me:

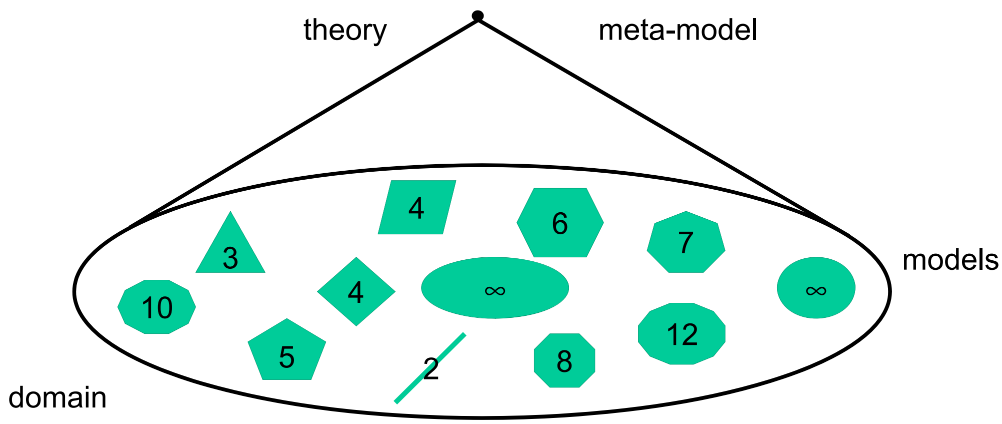
to give a tutorial that provides a broad perspective of FOSD including future, present, and past, especially for people from the outside.

- Invitation list suggested few "outsiders"

- Place FOSD in context within a broad, historical framework to explain its origin, goals, future expectations
  - if I discuss a topic, consider it well-studied
  - if I don't discuss a topic, likely an open research area

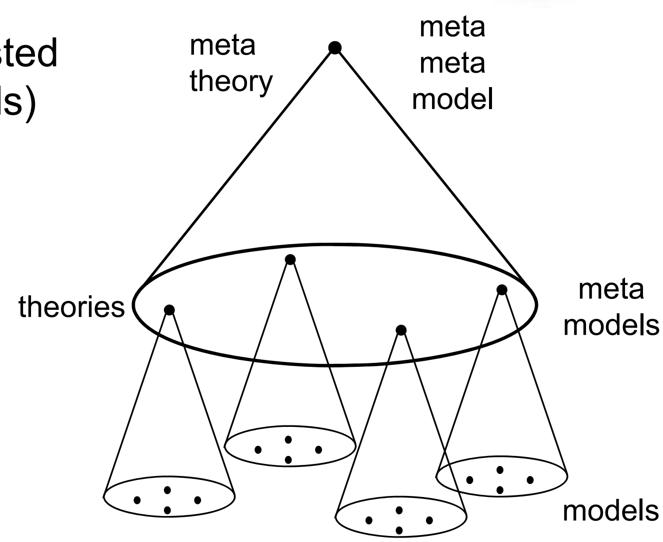
Origins of FOSD: Science & Physics

- Science is knowledge that has been reduced to a system. (Robert van de Geijn)
- Theoretical Physics: Find fundamental laws that explain families of known phenomena; the smaller the number of laws, the better.
  Laws predicts existence of other phenomena (ex: Higgs Boson)



FOSD Principles

- Although we are interested in theories (meta-models) of particular domains…
- FOSD is also about meta-meta model whose domain-independent principles are common across all FOSD "theories" or "meta-models"



Origins of FOSD: Automation

- Core demonstration of systemization of knowledge
  - not interested in “taxonomies”
- Future paradigms of software development will embrace:
  - **Generative Programming (GP)**
    - want software development to be automated
  - **Domain-Specific Languages (DSLs)**
    - not Java & C#, but high-level notations
  - **Automatic Programming (AP)**
    - declarative specs → efficient programs
- Need simultaneous advance in all three fronts to make a significant change

Not Wishful Thinking...

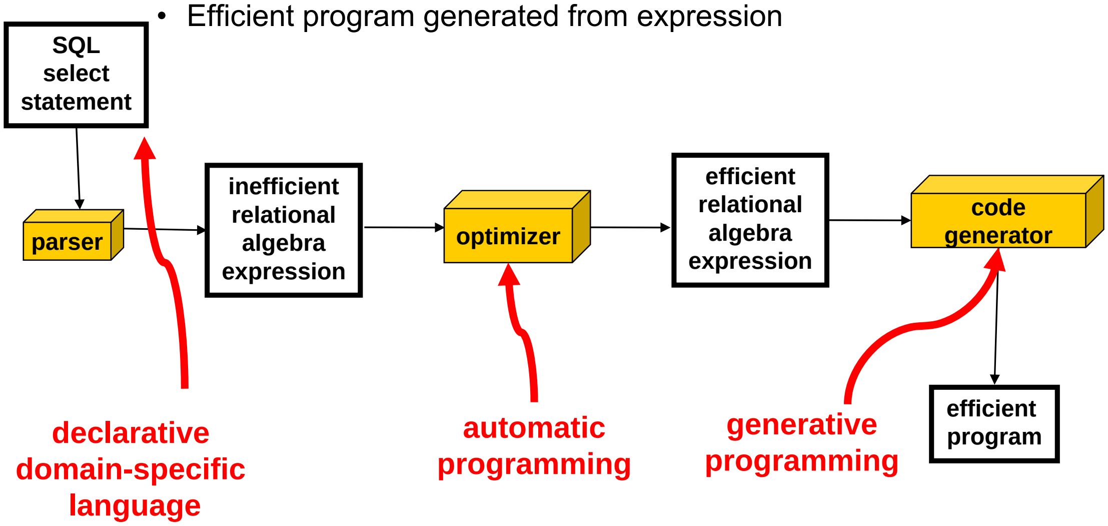
- Example of this futuristic paradigm realized 30+ years ago
  - when many AI researchers gave up on automatic programming

Relational Query Optimization

- The most significant result in automated program design and development, period
- Not mentioned in typical SE texts …

Relational Query Optimization (RQO)

- Declarative query is mapped to an relational algebra expression
- Each expression represents a unique program
- Expression is optimized using algebraic identities
- Efficient program generated from expression



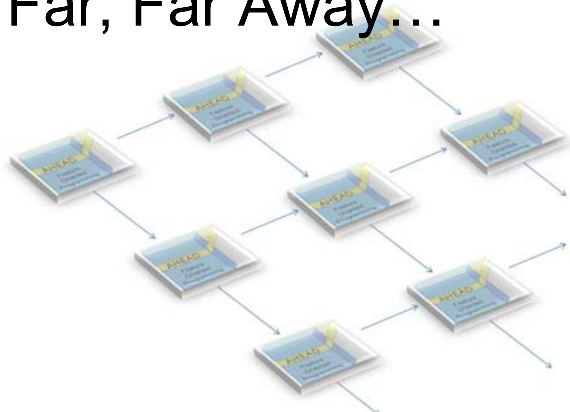
Feature Oriented Software Development (FOSD)

- Is experimentally-driven approach aimed at reproducing an RQO-like results in other domains
- Domains of artifacts (software) can be understood and constructed algebraically
  - systematizing knowledge of a domain
  - to automate construction, improve quality, reduce maintenance costs
  - to teach engineers, undergraduates, graduates about a science of design

Review basic concepts

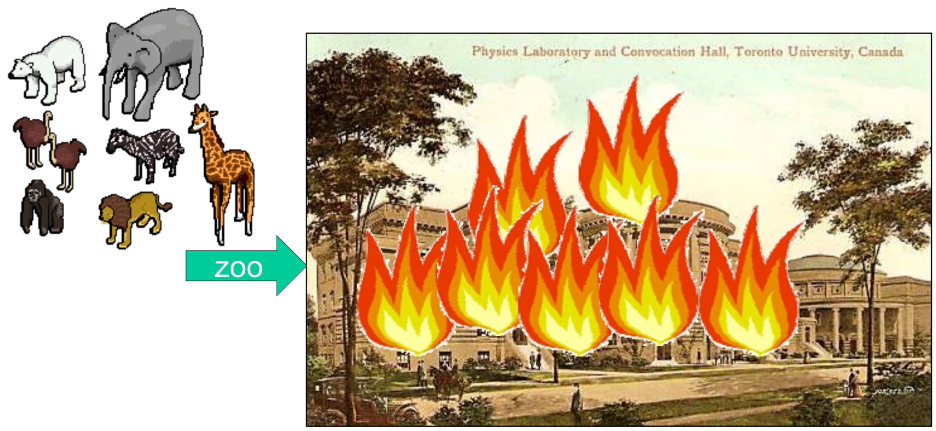
FOSD is a Technically Rich Area

A Long Time Ago,
in a University Far, Far Away…



Toronto, January 1977

- Largest snowfall in decades
- Given a desk in Sanford Fleming (known for proposing worldwide standard time zones) building, constructed in 1907
- Miracle of February 1977



Fire of February 1977

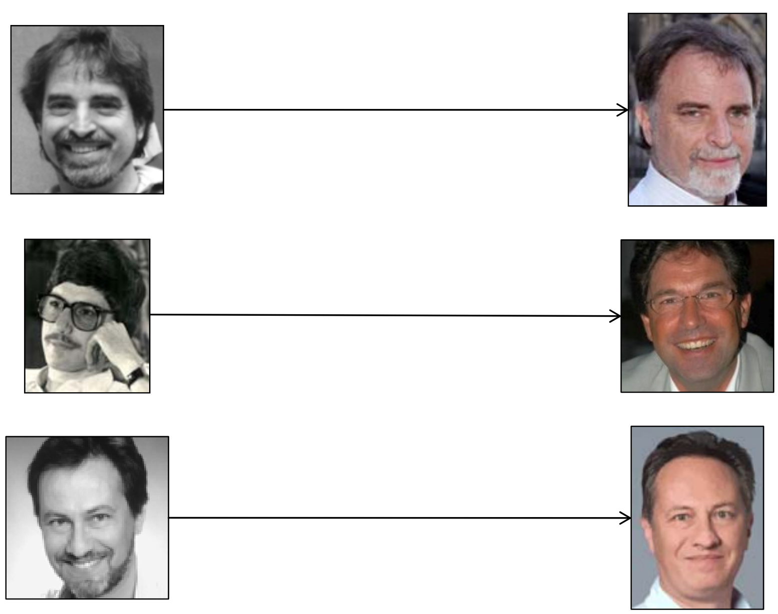
- Moved to 121 St. Joseph, St. Michaels College (Theology)

Fire of February 1977

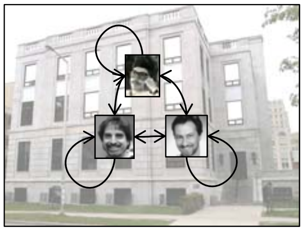
- Over time, shared an office with others

30+ Years Later…

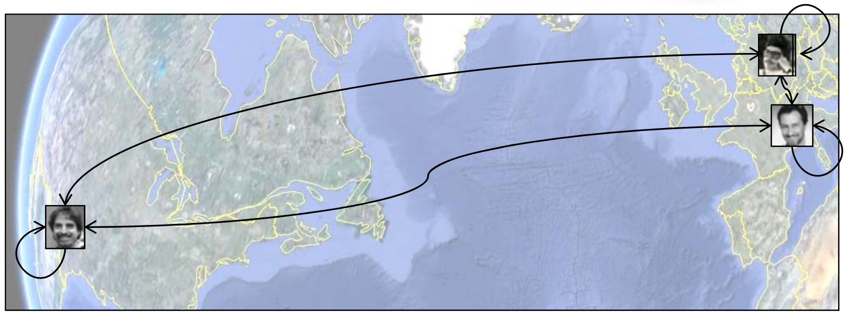
- Here we are again



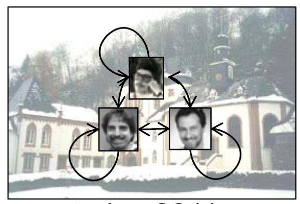
People And Relationships Evolve



1980

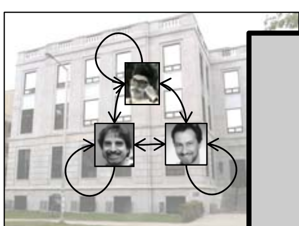


circa 1981-2010



today 2011

People And Relationships Evolve

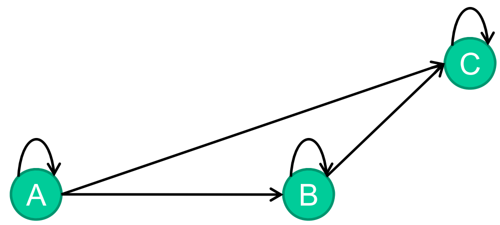


1980

True for all objects and relationships, including those in software

Notation

- “Objects” are entities that we want to relate

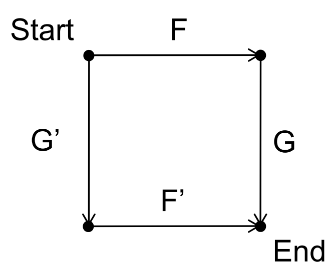


Category
(external diagram)

- “Arrows” are relationships
  - arrow is a map (A→B) says A maps to B
- Rules are simple:
  - arrows compose
  - composition is associative
  - always identity arrows (often not drawn)

Commuting Diagrams

- Are categories where all paths from one object to another yield the same result

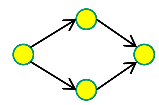


F • G = G’ • F’

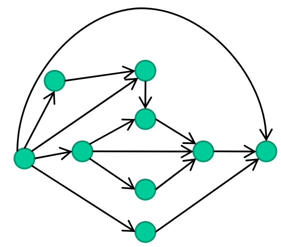
- Are theorems of category theory

What to Do With Categories?

- Map them: A is mapped onto/into B
  - **embedding of A into B**
  - each object (arrow) in A maps to an object (arrow) in B
  - such that all inferred arrows in A can be inferred in B



category A

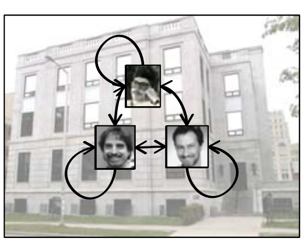


category B

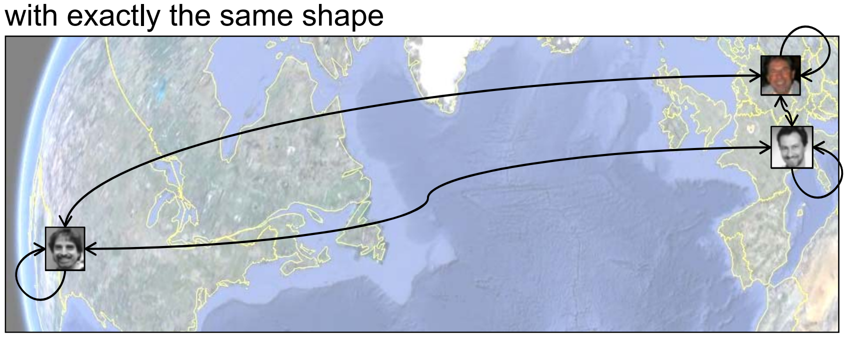
Functor

Functors

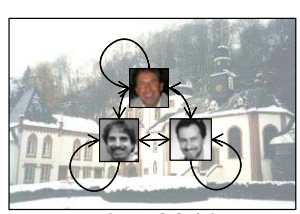
- Isomorphic categories are very common
  - categories with exactly the same shape



1980



circa 1981-2010



today 2011

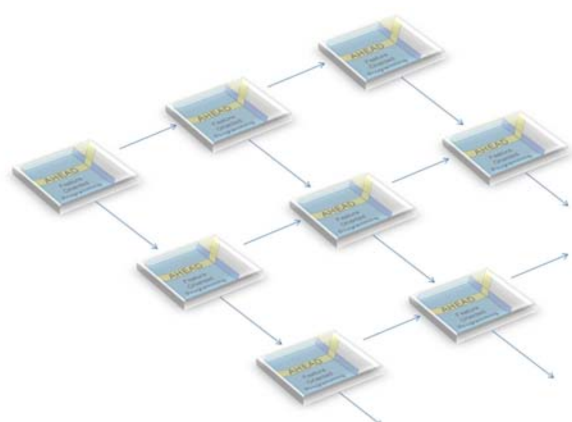
Recap

- Category (at least what we need)
  - is a directed graph whose edges can be inferred
  - commuting diagrams define equivalent paths
  - a functor embeds one category into another

- With this as a backdrop,
  review progression of generations of FOSD technology

- Can't cover everything – my apologies to those who I don't acknowledge

1^{st} Generation FOSD

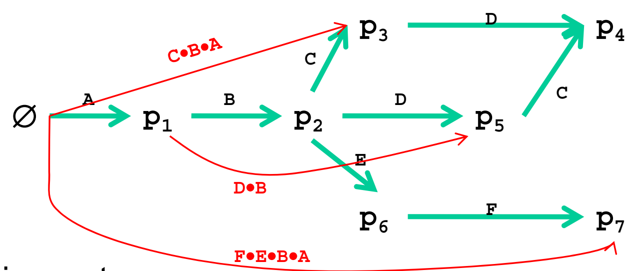


GenVoca

1992 – 1997

Product Line

- Family of related programs and the lone base (or empty) program ∅
- **Features** are "increments in functionality", drawn as arrows
- Programs related by features or compositions of features



- Product line is a category
    - says nothing about how arrows (features) are implemented; only represents feature-based relationships among programs

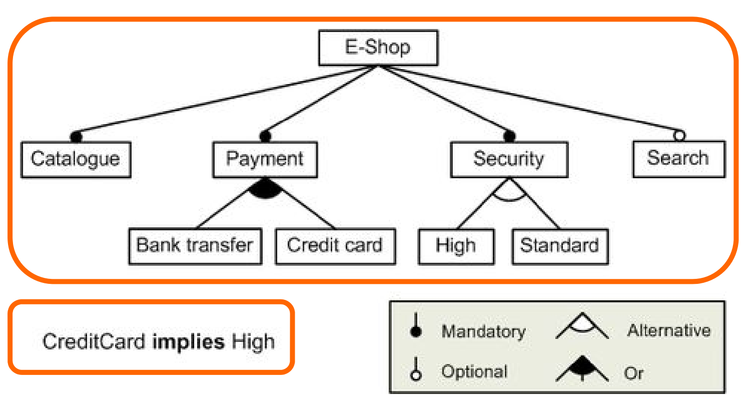
- Questions: how are PL categories encoded? & how are arrows implemented?

(Unordered) Feature Models

- Kyo Kang 1990
    - **tree** that encodes containment, exclusion, and aggregation relationships among features
    - **cross-tree constraints** express additional restraints
    - now a standard modeling concept

tree

cross-tree constraints



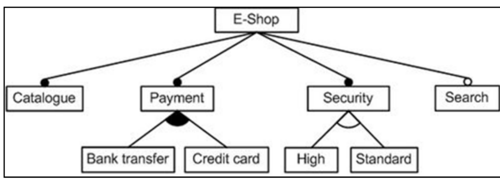
Ordered Feature Models (OFMs) 1992

- Viewing features as "arrows" is natural for **incremental development** (Wirth & Dijkstra), a fundamental way to control complexity

- FMs are inherently unordered
  - program specification is a set of features
  - to relate to categories, a specification must be a sequence of features

- Ordering information added by encoding FM as a grammar
  - encode a feature tree as a context-free grammar where each token is a feature
  - cross-tree constraints are context-sensitive
  - sentences of grammar is a language

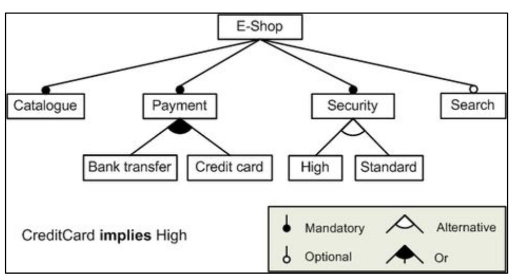
language = product line
sentence = product



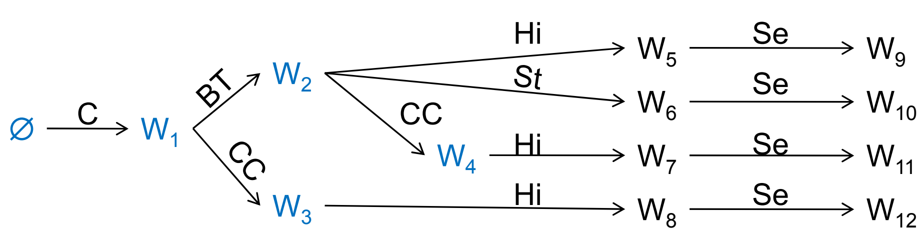

```
E-Shop : Catalogue Payment+ Security [Search];

Payment : BankTransfer | CreditCard ;

Security : High | Standard ;
```


OFMs and Categories

- Given an OFM (context-sensitive grammar) of:

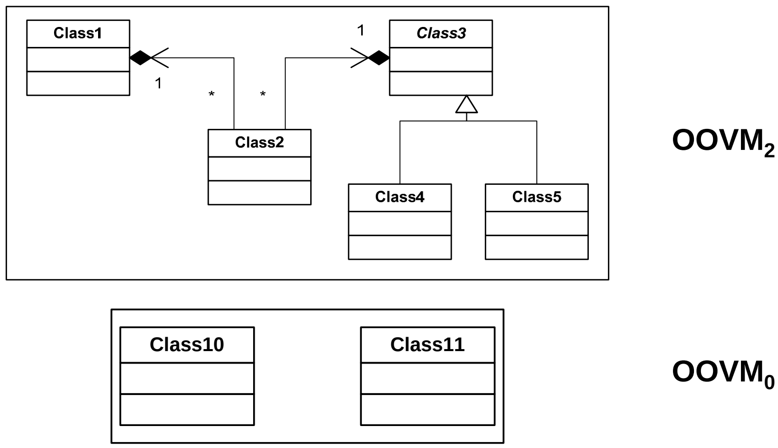


- Produces:



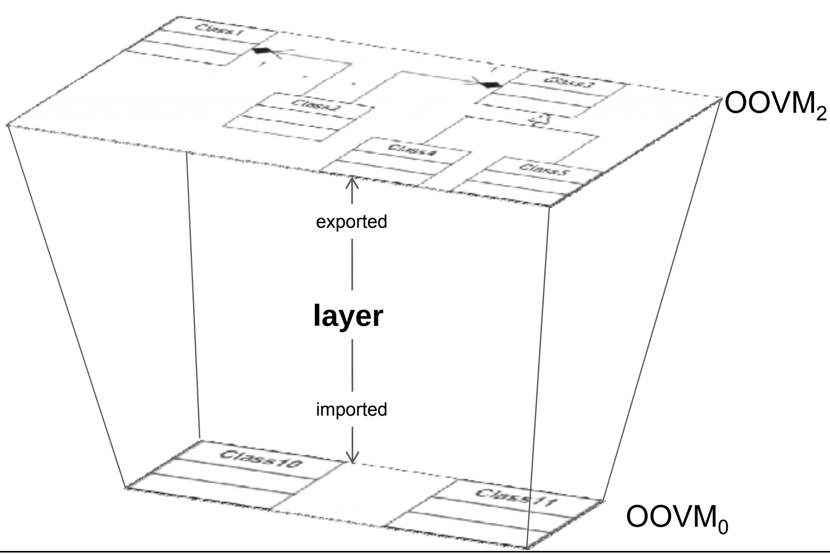
How to Implement Arrows (Features)?

- Feature = layer of software = “lego”
- Features exported and imported **Object-Oriented Virtual Machines**



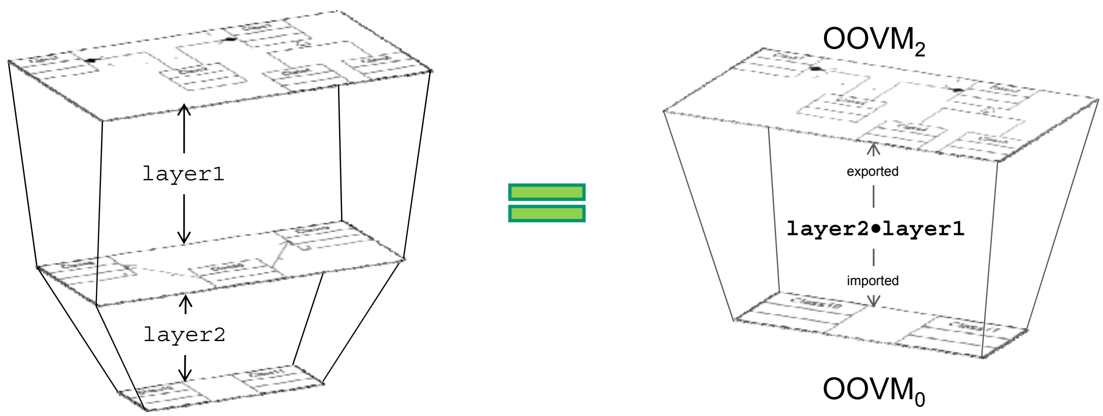
Feature or Layer

- A **layer** maps between an exported OOVM and an imported OOVM



Layer Composition

- A composition of 2+ layers = another (composite) layer
- Closure, arrow composition:
    - “lego composition”, “layered abstractions”,



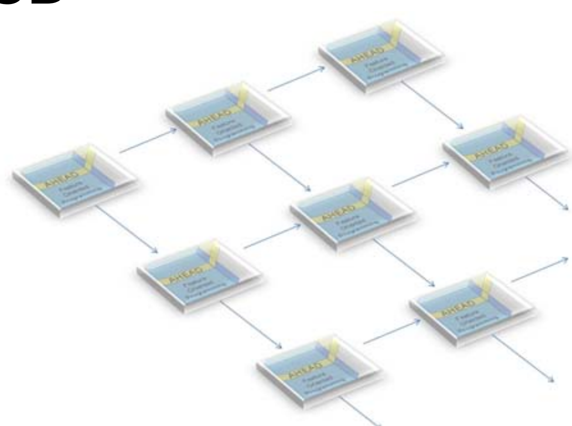
Other Foundational Contributions

- **Principles of Parameterized Programming**
  J.A. Goguen, IEEE TSE 1984
- **On the Design and Development of Program Families**
  D.L. Parnas, IEEE TSE 1976
- **Program Development using Stepwise Refinement**
  N. Wirth, CACM 1971
- **Structured Programming**
  E.W. Dijkstra, Software Engineering Techniques, 1970
- **Mass Produced Software Components**
  D. McIlroy, ICSE 1968

Key Limitations

1. Layers with fixed interfaces were too rigid
2. How to generalize beyond source code?
3. Are OFMs essential? Could multiple orderings exist? Which is best?
4. If there are features, where are feature interactions?

2^{nd} Generation FOSD



Mixin Layers
Smaragdakis
1998-2001

Subjectivity 1992

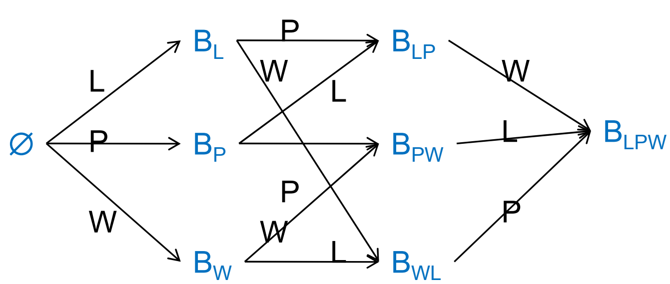
- Ossher and Harrison observed that there is no such thing as a standardized interface
  - in effect, an interface presents our current understanding of an abstraction
  - in time, our understanding evolves
  - MS COM – query interface

- Thought experiment: what are attributes of a book?
  - author, title, publisher
  - printer: how much paper, ink?
  - warehouse: how much volume?

- In short, attributes of an object are subjective w.r.t. application it is being used

But Wait!

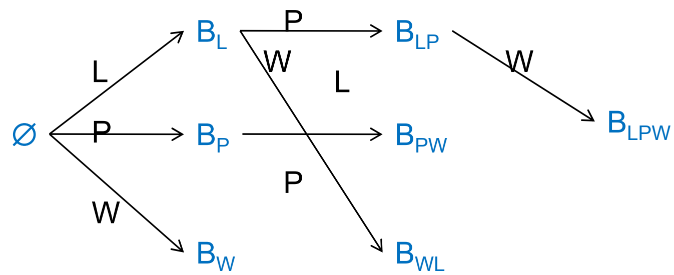
- This is precisely the reason for features and product lines
  - product line of object (class) definitions



- Or an OFM representation (projection) of it
  - only one way to derive a particular artifact

But Wait!

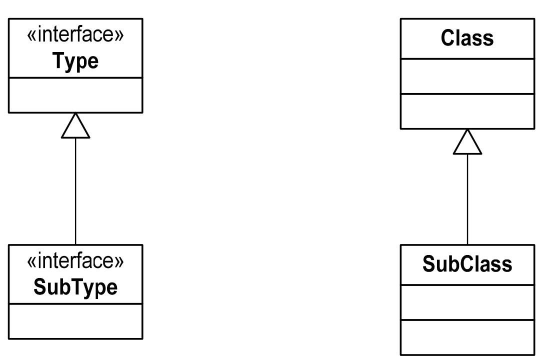
- This is precisely the reason for features and product lines
  - product line of object (class) definitions



- Or an OFM representation (projection) of it
  - only one way to derive a particular artifact

The Real Challenge Is

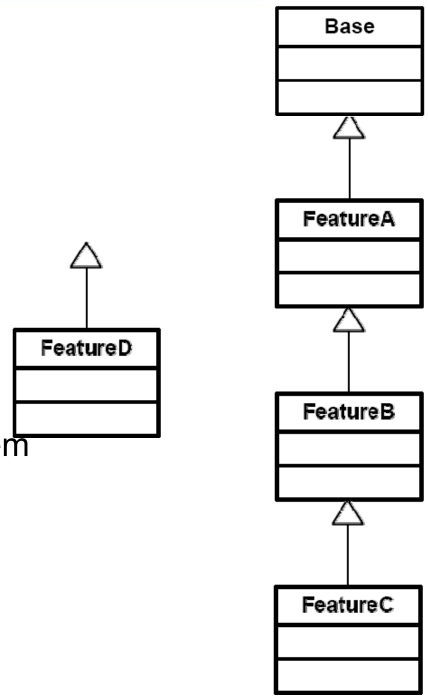
- How to implement features so that (Java) interfaces and classes could evolve incrementally and in a modular way?
  - incrementally add new classes, interfaces, methods, members, extend existing methods?
- Object-Oriented languages provide an obvious answer:



Inheritance

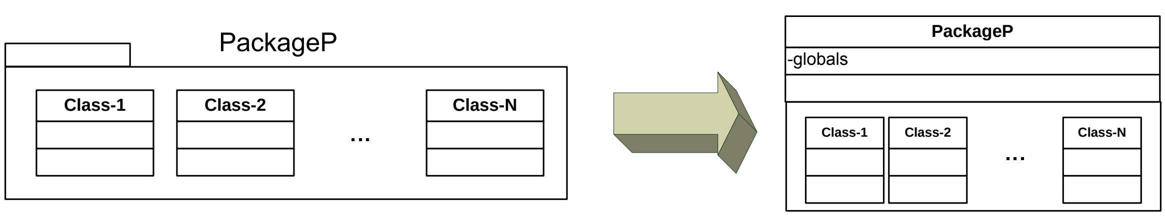
A Twists

- Need something more flexible than subclassing
- Mixins – classes whose superclasses are specified by parameter
  (Bracha & Cook 1990)
- But not quite – mixins are microscopic; need to scale them



Another Twist

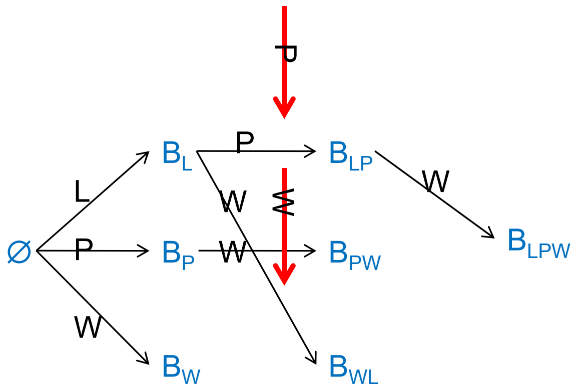
- Scale to mixins to encode entire packages
- Smaradakis solution: nested class refactoring

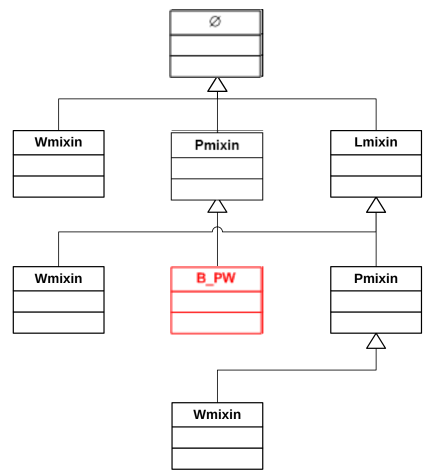


- Allows us to create inheritance hierarchies of packages
- Nested classes + mixins = **Mixin-Layers**
- Produce huge programs by composing small number of huge features

Connection to Categories is Simple

- Rotate category by 90°
- Arrows (classes) are mixin-layers
- Objects (programs) are computed





Other Foundational Contributions

- **Classes and Mixins**
  M. Flatt, S. Krishnamurthi, M. Felleisen. POPL 1998.

- **Using Role Components to Implement Collaboration-Based Designs**
  M. Van Hilst and D. Notkin. OOPSLA 1996.

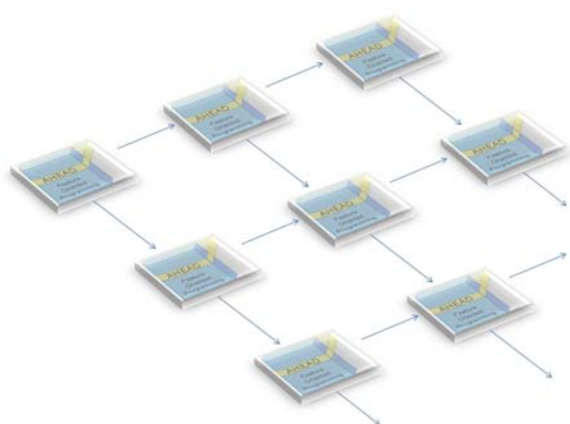
- **Mixin-Based Inheritance**
  G. Bracha and W. Cook. OOPSLA 1990.

- **Virtual classes: A powerful mechanism in Object Oriented Programming**
  O. L. Madsen and B. Møller-Pedersen, OOPSLA 1989.

Key Limitations

1. Layers with fixed interfaces were too rigid
2. How to generalize beyond source code?
3. Are OFMs essential? Could multiple orderings exist? Which is best?
4. If there are features, where are feature interactions?

3^{rd} Generation FOSD



AHEAD
2001-2006

Multiple Representations

- All programs have multiple representations
  - source code
  - documentation
  - makefiles
  - formal models
- What happens when you add a new feature to a program?
- Ans: update all representations simultaneously for consistency
- Significance: generalize modularity to encompass (virtually) arbitrary representations in programs and features
- Ideas of feature-based synthesis apply uniformly

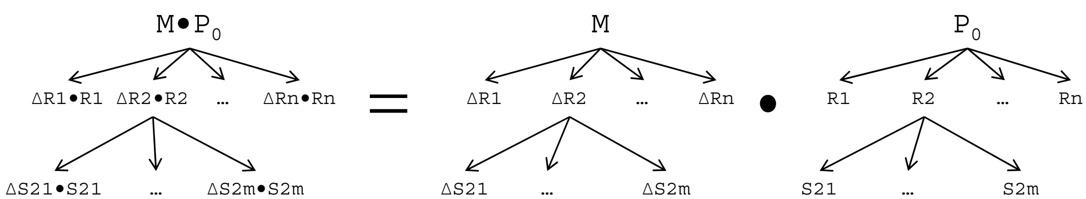
Typical Example

- A parser has multiple representations
  - grammar, source code, documentation
  - represented as a tuple of artifacts
  - example parser $P_0 = [\, g_0, s_0, d_0 \,]$
- Every feature modifies each representation lock-step
- Suppose feature M adds state machine to P's language
  - makes changes Δg_m to grammar
  - makes changes Δs_m to source
  - makes changes Δd_m to documentation
  - feature is a tuple of changes: $M = [\, \Delta g_m, \Delta s_m, \Delta d_m \,]$

Element-Wise Composition

$$
\begin{aligned}
P &= M \bullet P_0 && \text{-- feature expression}\\
&= [\Delta g_m,\ \Delta s_m,\ \Delta d_m] \bullet [g_0,\ s_0,\ d_0] && \text{-- substitution}\\
&= [\Delta g_m \bullet g_0,\ \Delta s_m \bullet s_0,\ \Delta d_m \bullet d_0] && \text{-- compose element-wise}
\end{aligned}
$$

- Recursive – each representation has its own (sub-) representations, recursively



Principles

- **Principle of Uniformity** – treat all representations similarly
  - <u>worst thing you could do is to have different compositional models for different representations</u>
  - key to representation scalability
- **Principle of Abstraction** – treat all levels of abstraction similarly
  - <u>worst thing you can do you is to have multiple compositional models at the same or different levels of abstraction</u>
  - key to abstraction scalability
- Simplicity without sacrificing power
- Standard engineering practice

Eating Your Own Dog Food

- Took an astonishingly long time to realize this, but you can build a single generic tool that defines the grammar of artifact types and how tree nodes compose
    - as opposed to building a specialized composition tool for each language

- See Apel's Feature Structure Trees in FeatureHouse

Key Limitations

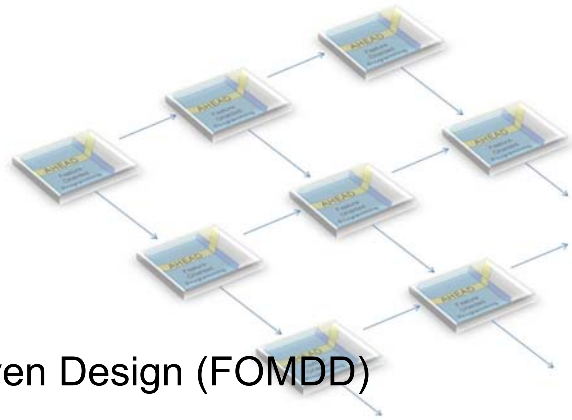
1. Layers with fixed interfaces were too rigid

2. How to generalize beyond source code?

3. Are OFMs essential? Could multiple orderings exist?   Which is best?

4. If there are features, where are feature interactions?

4rd Generation



Feature-Oriented Model Driven Design (FOMDD)
Trujillo, Diaz
2006-2008

Other Functional Relationships

- Different representations can be related
  - parser's grammar **g** related to its source **s** by **javacc**
  - source **s** related to bytecode **b** by **javac**
- A commuting diagram expresses these relationships

$$\begin{pmatrix} g_1 \\ s_1 \\ b_1 \end{pmatrix} \xrightarrow{j} \begin{pmatrix} g_2 \\ s_2 \\ b_2 \end{pmatrix} \xrightarrow{k} \begin{pmatrix} g_3 \\ s_3 \\ b_3 \end{pmatrix}$$

generation #1
generation #2
generation #3

Other Functional Relationships

- Different representations can be related
  - parser's grammar `g` related to its source `s` by `javacc`
  - source `s` related to bytecode `b` by `javac`

- A commuting diagram expresses these relationships

$$
\begin{tikzcd}
g_1 \arrow[r, "\Delta g_j"] \arrow[d, "javacc"] & g_2 \arrow[r, "\Delta g_k"] \arrow[d, "javacc"] & g_3 \arrow[d, "javacc"] \\
s_1 \arrow[r, "\Delta s_j"] \arrow[d, "javac"] & s_2 \arrow[r, "\Delta s_k"] \arrow[d, "javac"] & s_3 \arrow[d, "javac"] \\
b_1 \arrow[r, "\Delta b_j"] & b_2 \arrow[r, "\Delta b_k"] & b_3
\end{tikzcd}
$$

generation #3
generation #4

- Utility: **Verification**. If there are multiple ways to produce an artifact, yielding different results, then your implementation is wrong

Significance of Ordering

- And if there are multiple orders in which features are composed, how do you choose the "best"?

- An answer exposes fundamental assumptions or properties about features and their implementations

- Features F and G **commute** if the following diagram commutes

$$
\begin{tikzcd}
\cdot \arrow[r, "F"] \arrow[d, "G"'] & \cdot \arrow[d, "G"] \\
\cdot \arrow[r, "F"] & \cdot
\end{tikzcd}
$$

$$F \bullet G = G \bullet F$$

Interesting Observation

- Apel and Kästner noted the following:

$$\begin{tikzcd} \cdot \arrow[r, "F"] \arrow[d, "G'"'] & \cdot \arrow[d, "G"] \\ \cdot \arrow[r, "F'"] & \cdot \end{tikzcd}$$

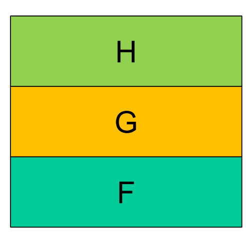
$F \bullet G = G' \bullet F'$

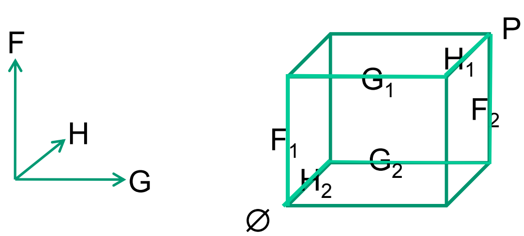
"pseudo-commutativity"

- What they discovered was the key to changing the composition order of features
  - order need not be fixed
  - order may change the contents of a feature (module)

Said Something Important

- Any path from ∅ to P is equivalent
  - if you assign "weights" to each arrow, you want the shortest path **geodesic**
- Typically geodesic reflects bottom-up construction





- Goguen observed order doesn't matter
- Choose order that you want
- Automatic translation?

Key Limitations

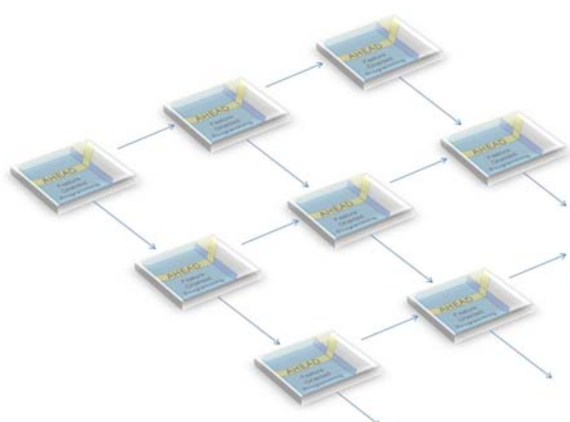
1. Layers with fixed interfaces were too rigid

2. How to generalize beyond source code?

3. Are OFMs essential? Could multiple orderings exist? Which is best?

4. If there are features, where are feature interactions?

5^{rd} Generation

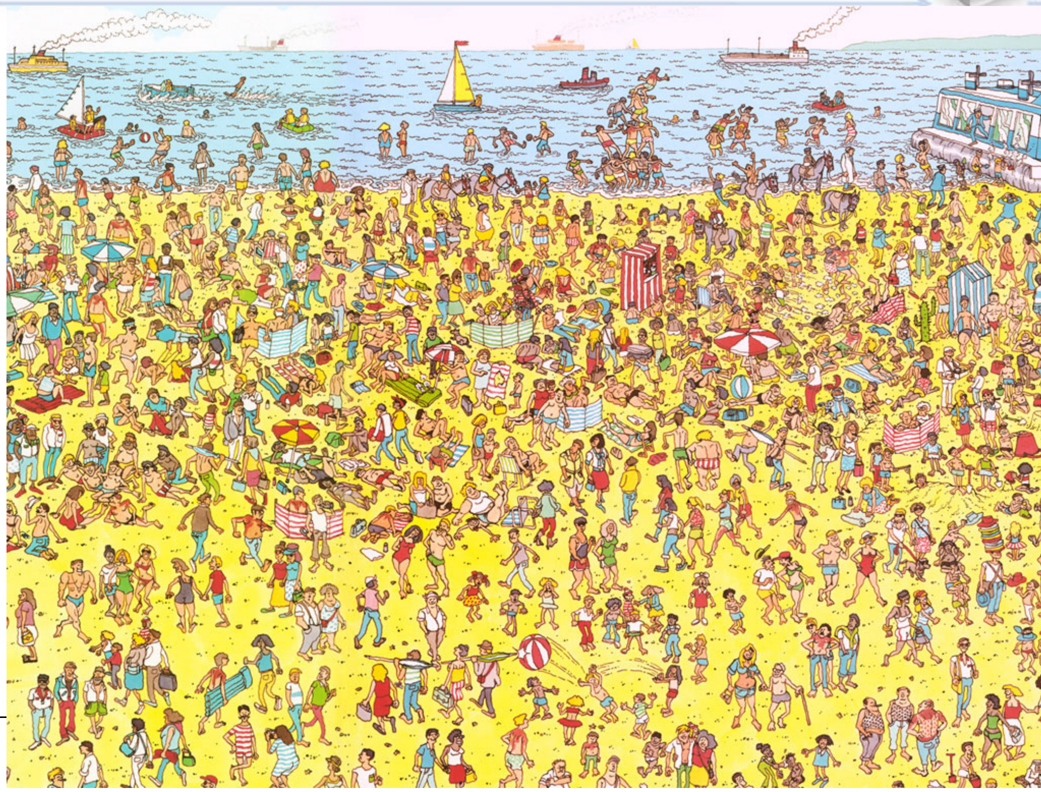


Feature Interactions

and Virtual Modularity

(today – but started long ago)

Where's Waldo?



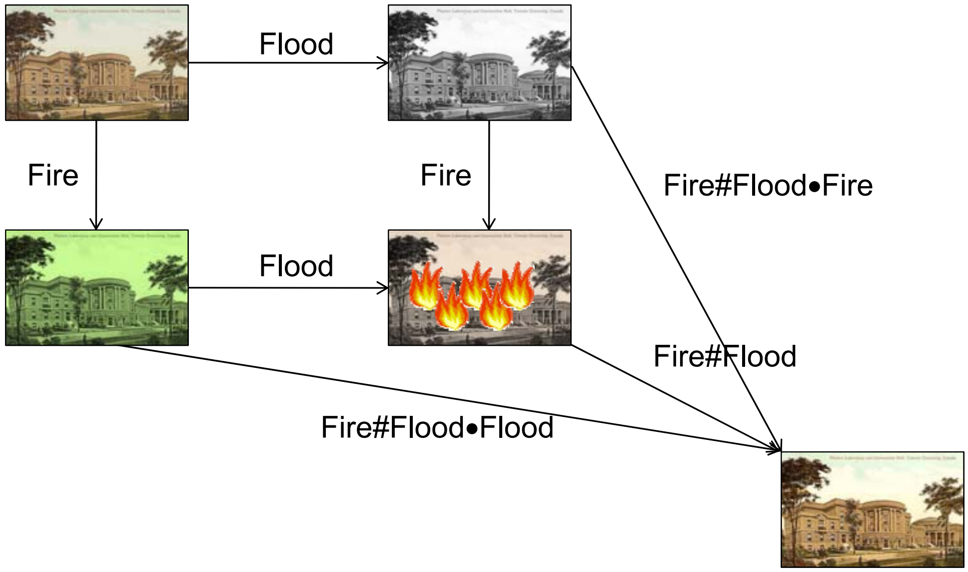
Commuting Diagrams

- Keep the following image in your mind

$$
\begin{tikzcd}
\bullet \arrow[r, "F"] \arrow[d, "G'"'] & \bullet \arrow[d, "G"] \arrow[ddr] & \\
\bullet \arrow[r, "F'"'] \arrow[drr] & \bullet & \\
 & & \bullet
\end{tikzcd}
$$

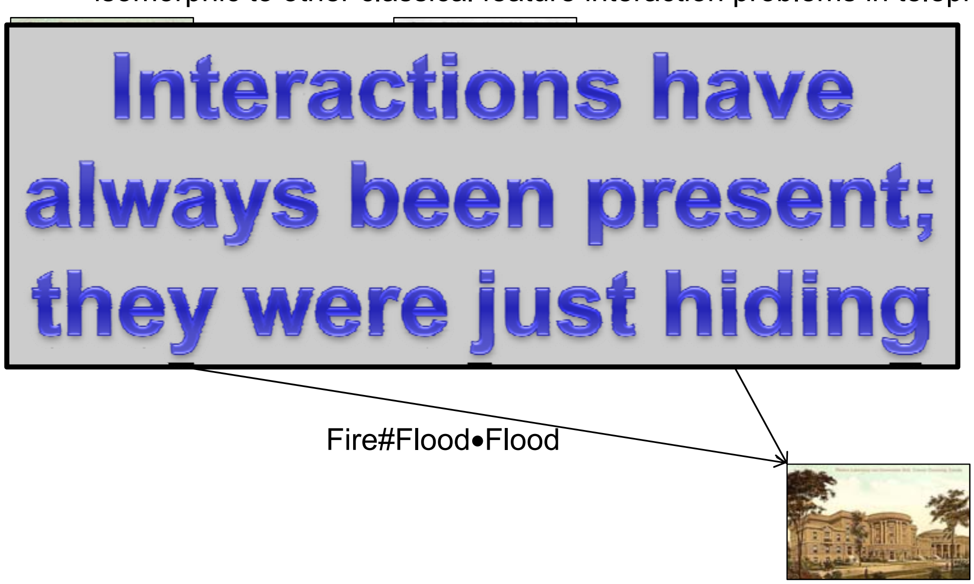
Feature Interactions

- Flood control – Fire control problem (Kang 2003)
  - isomorphic to other classical feature interaction problems in telephony



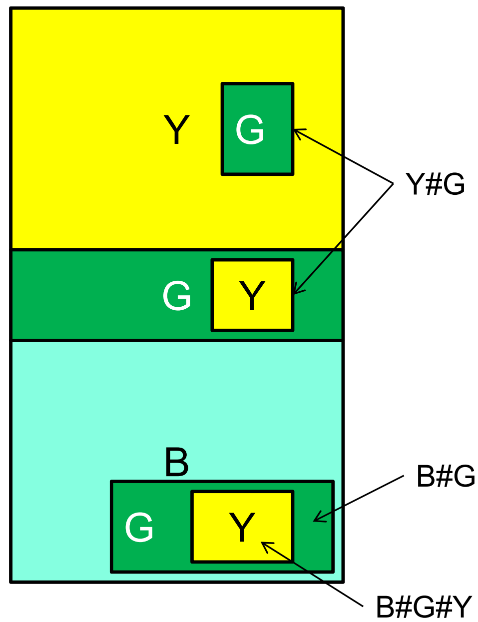
Feature Interactions

- Flood control – Fire control problem (Kang 2003)
  - isomorphic to other classical feature interaction problems in telephony



Kästner's CIDE

- Ressurrects use of preprocessors
  - standard technique for building product lines
  - #ifdef <code> #endif
- Color structures according to the feature that implements or introduces that structure
- Idea applies to directories of files (artifact hierarchies) as well
- Nesting of colors indicates feature interactions



Significance: Projectional FOSD

- Projectional approach to product-lines
  - build artifacts with everything inside them
  - project (remove) parts that you don't need
  - **virtual modularization**
- Addresses a basic limitation in mixin-layer like approaches
  - mixin-layers work well with large-scale, medium-scale changes, but not with small-scale (code fragment) changes
- Alternate way to implement features (arrows)
  - isomorphic to compositional approaches
  - can't yet rule out compositional approaches
- Still want to think in terms of arrows to relate to

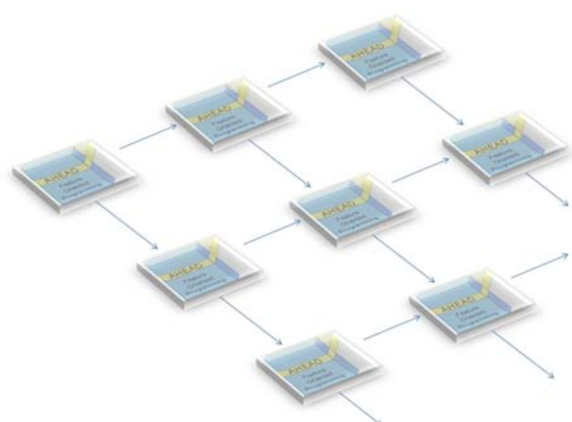
Model Driven Engineering and Refactoring

Other Foundational Contributions

- **Feature Oriented Programming: A Fresh Look at Objects**
  C. Prehofer, ECOOP 1997

- **Mapping Features to Models: A Template Approach Based on Superimposed Variants**
  K. Czarnecki and M. Antkiewicz, GPCE 2005

Oops…



I'm out of time and there's lots more…

Closing Thoughts

- Software design is in the dark ages
  - practiced as an art, not as a science
  - ruled by fads, personalities, dogma; rather than technical prowess
  - celebrates complexity and eschews simplicity
  - reassures our greatest weaknesses

"We are geniuses at making the simplest things look complicated"

"We are governed by the inability to abstract,
to distinguish essential from artificial complexity
and thus we pay the consequences"

- The hard part is revealing the simplicity behind what we do

It takes effort, time.
The reward is the future.

Closing Thoughts

- Features are a fundamental form of modularity
  - see them everywhere, from automotives, PC, and software
  - integrates ideas from the entire history of software design elegantly
  - program design and synthesis has a simple algebraic underpinning

    design is all about structure definition and manipulation

    which is what mathematics is about

- Features will be an integral part of a future of software design and a Science of Design

Closing Thoughts

- FOSD is a generalization of the Relational Model
- Relational Model was based on set theory
  - this was the key to understanding a modern view of databases
  - set theory used was shallow
  - fortunate for programmers and database users
  - set select, union, join, intersect
  - disappointment for mathematicians

- Categories lie at the heart of FOSD
  - as a language to express our results
  - places research results in context
  - new insight in design, verification, optimization

Thank You

More Foundational Work

- Feature Models
  - Czarnecki & Wasowski (SPLC 2007)
  - Perry (ICSE 1989)

- Features, Verification, and State Machines
  - Classen & Heymans & Schobbens &; Legay & Raskin (ICSE 2009)
  - Li, Krishnamurthi, & Fisler (FSE 2002, ASE 2002)
  - Stärk & Schmid & Börger (Jbook 2001)

- Dynamic compositions of features
  - Zave (IEEE TSE 1998)

- + Others…