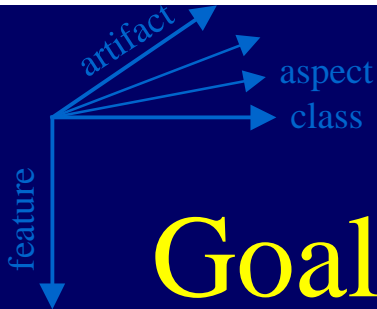


What is Multi-Dimensional Separation of Concerns?

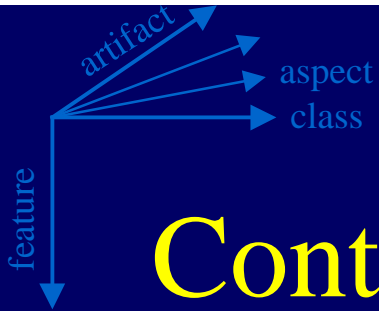
Harold Ossher and Peri Tarr
IBM T. J. Watson Research Center



Goals of Introduction

- Introduce concepts and issues
 - Of domain as a whole, not any specific approach
- Suggest terminology
 - Suitable across many approaches
 - Intended to absorb terms for various approaches
- Facilitate common understanding as basis for discussion

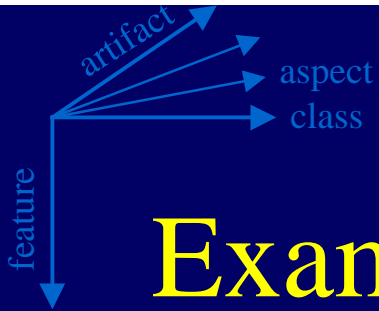




Contents

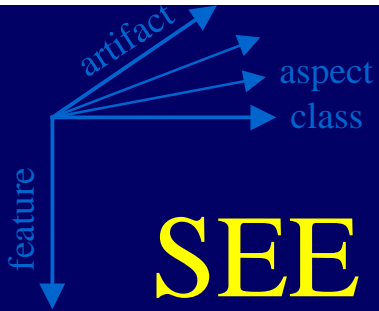
- **Motivating Example**
- Concepts and terminology
- Discussion
- Mechanisms and Tradeoffs



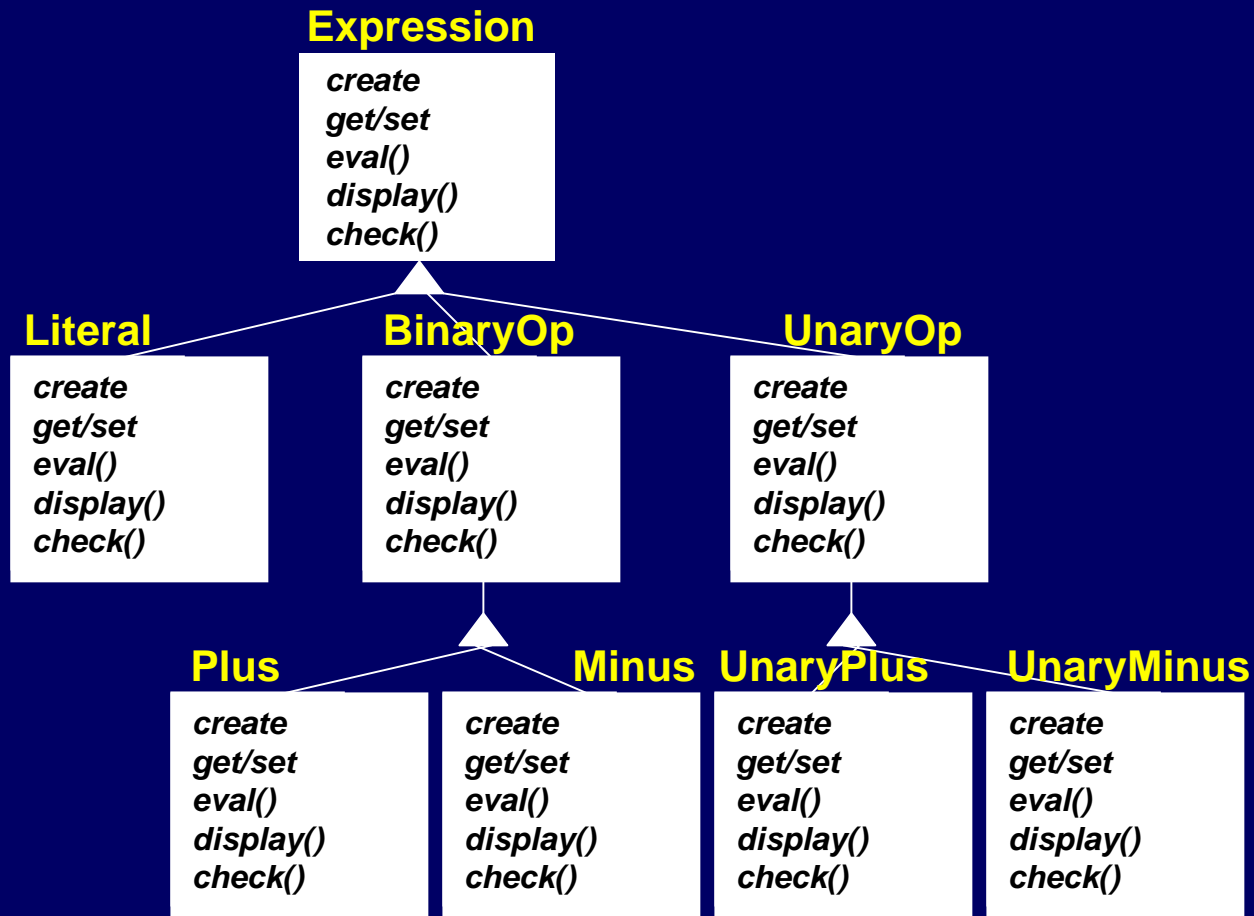


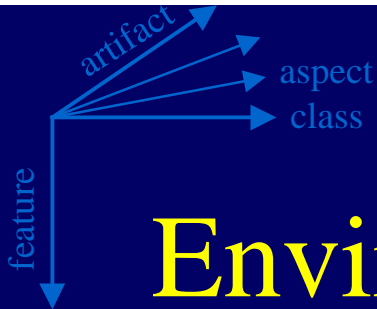
Example: Expression SEE

- Requirements
 - Shared, common representation of expressions
 - Tools operate on it
 - Initial toolset:
 - Evaluation, textual display, syntax checker
 - All tools optional
- Design: UML
- Code: Java



SEE (High-Level) Design

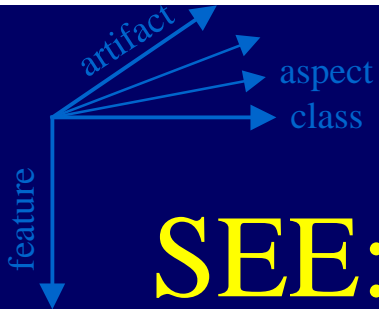




Environmental Concerns

- Artifact
 - Requirements, Design, Code, Test cases
- Feature
 - Kernel AST, Display, Evaluation, Check
- Class (or Object)
 - Expression, BinaryOp, UnaryOp, Literal
 - Plus, Minus, UnaryPlus, UnaryMinus
- Aspect, variant, unit of change, customization, ...





SEE: Concern Details

Requirements

- Central representation of expressions
- Display
- Evaluation
- Checking

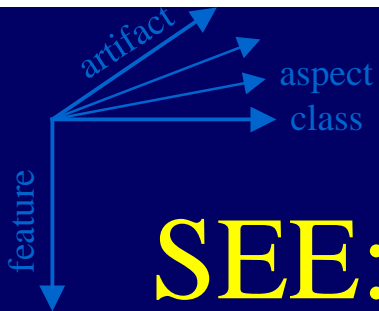
Design

- Expression
 - get/set
 - display()
 - eval()
 - check()
- BinaryOperator
 - get/set
 - display()
 - eval()
 - check()
- UnaryOperator
 - ...
- Plus, Minus, ...

Code

- ExpressionImpl
 - get/set
 - display()
 - eval()
 - check()
 - _display
 - _cachedResult
- BinaryOperatorImpl
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _leftOpnd
 - _rightOpnd
- UnaryOperatorImpl
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _opnd
- Plus, Minus, UnaryPlus, UnaryMinus
 - get/set
 - display()
 - eval()
 - check()





SEE: Concern Details

Requirements

- **Central representation of expressions**
- Display
- Evaluation
- Checking

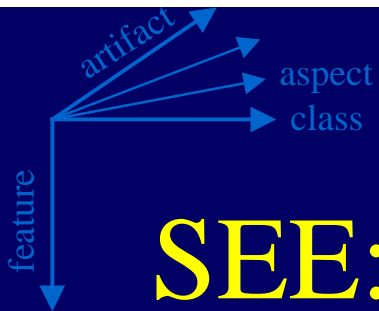
Design

- **Expression**
 - **get/set**
 - display()
 - eval()
 - check()
- **BinaryOperator**
 - **get/set**
 - display()
 - eval()
 - check()
- **UnaryOperator**
 - ...
- **Plus, Minus, ...**

Code

- **ExpressionImpl**
 - **get/set**
 - display()
 - eval()
 - check()
 - **_display**
 - **_cachedResult**
- **UnaryOperatorImpl**
 - **get/set**
 - display()
 - eval()
 - check()
 - **_opName**
 - **_opnd**
- **BinaryOperatorImpl**
 - **get/set**
 - display()
 - eval()
 - check()
 - **_opName**
 - **_leftOpnd**
 - **_rightOpnd**
- **Plus, Minus, UnaryPlus, UnaryMinus**
 - **get/set**
 - display()
 - eval()
 - check()





SEE: Concern Details

Requirements

- Central representation of expressions
- **Display**
- Evaluation
- Checking

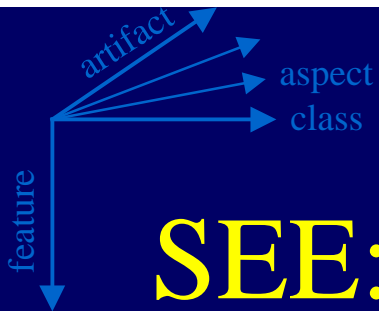
Design

- Expression
 - get/set
 - **display()**
 - eval()
 - check()
- BinaryOperator
 - get/set
 - **display()**
 - eval()
 - check()
- UnaryOperator
 - ...
- Plus, Minus, ...

Code

- ExpressionImpl
 - **get/set**
 - **display()**
 - eval()
 - check()
 - `_display`
 - `_cachedResult`
- BinaryOperatorImpl
 - **get/set**
 - **display()**
 - eval()
 - check()
 - `_opName`
 - `_leftOpnd`
 - `_rightOpnd`
- UnaryOperatorImpl
 - **get/set**
 - **display()**
 - eval()
 - check()
 - `_opName`
 - `_opnd`
- Plus, Minus, UnaryPlus, UnaryMinus
 - **get/set**
 - **display()**
 - eval()
 - check()





SEE: Concern Details

Requirements

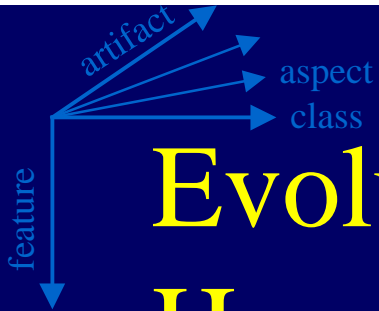
- **Central representation of expressions**
- **Display**
- **Evaluation**
- **Checking**

Design

- **Expression**
 - get/set
 - display()
 - eval()
 - check()
- **BinaryOperator**
 - get/set
 - display()
 - eval()
 - check()
- **UnaryOperator**
 - ...
- **Plus, Minus, ...**

Code

- **ExpressionImpl**
 - get/set
 - display()
 - eval()
 - check()
 - _display
 - _cachedResult
- **UnaryOperatorImpl**
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _opnd
- **BinaryOperatorImpl**
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _leftOpnd
 - _rightOpnd
- **Plus, Minus, UnaryPlus, UnaryMinus**
 - get/set
 - display()
 - eval()
 - check()

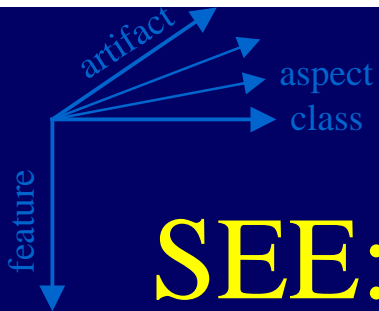


Evolution: An Environmental Hazard

- **New requirements:**
 - **Optional persistence of expressions**
 - **Add optional style checkers (IBM, SEI, ...)**
 - *Permit any combination of syntax and/or style checking*

➔ **(Note: Bang! You're hosed.)**





SEE: Concern Details

Requirements

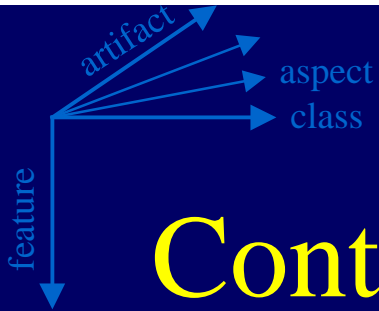
- Central representation of expressions
- Display
- Evaluation
- Checking
- Persistence
- Style checking

Design

- Expression
 - get/set
 - display()
 - eval()
 - check()
 - save()
- BinaryOperator
 - get/set
 - display()
 - eval()
 - check()
 - save()
- UnaryOperator
 - ...
- Plus, Minus, ...

Code

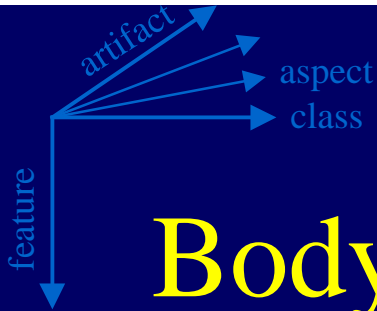
- ExpressionImpl
 - get/set
 - display()
 - eval()
 - check()
 - _display
 - _cache
 - save()
 - _database
 - _checkScore
- BinaryOpImpl
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _leftOpnd
 - _rightOpnd
 - save()
- UnaryOpImpl
 - get/set
 - display()
 - eval()
 - check()
 - _opName
 - _opnd
 - save()
- Plus, Minus, UnaryPlus, UnaryMinus
 - get/set
 - display()
 - eval()
 - check()
 - save()



Contents

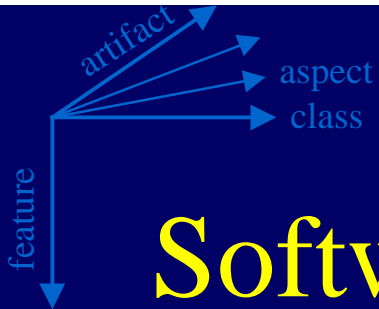
- Motivating Example
- Concepts and terminology
- Discussion
- Mechanisms and Tradeoffs





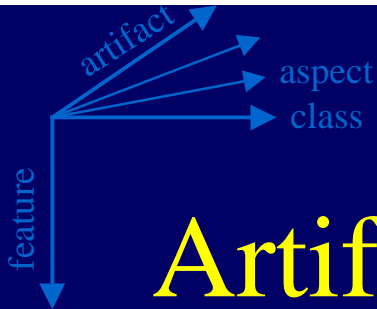
Body of Software

- All software that forms part of a software development effort
 - Material from all lifecycle phases
 - Might be for multiple related software products
 - Might include multiple variants
- The product of the software development process



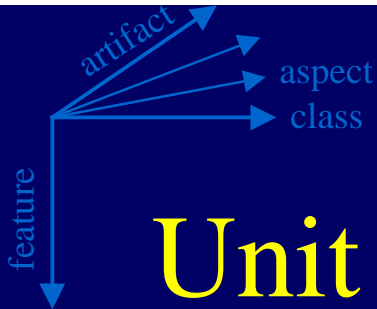
Software Artifact

- Major portion of a body software
 - Requirements document
 - Design document
 - Body of code
 - Test plan
 - Etc.

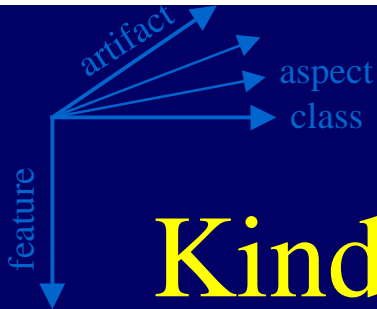


Artifact Language

- Language in which an artifact is written
 - Specification formalism (e.g., Z)
 - Design notation, perhaps graphical (e.g., UML)
 - Programming language (e.g., Java)
- Also called *notation* or *formalism*
- Different artifacts are often written in different languages

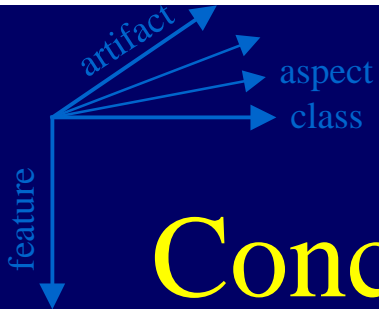


- Identifiable piece of software within an artifact
- Nature depends on:
 - artifact language
 - granularity
- Examples:
 - Requirement, box representing class, method
 - Logical formula, statement, expression



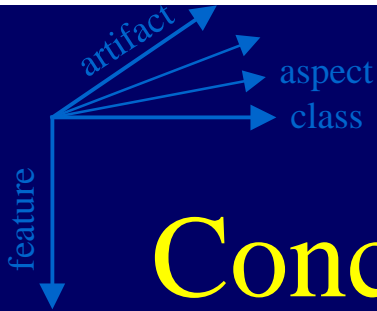
Kinds of Units

- Primitive unit
 - Considered atomic
- Compound unit
 - Contains primitive or subsidiary compound units
- Contextual:
 - Method can be a primitive unit,
 - or a compound unit containing statements



Concern

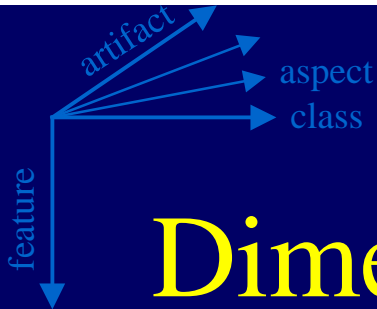
- Area of interest in a body of software
 - Artifact
 - Function or feature
 - Data type (class, object)
 - Aspect
 - Non-functional
 - Distribution, concurrency, error-handling...
 - Variant, configuration, version, use case, ...
- **Many different kinds**



Concerns Map to Units

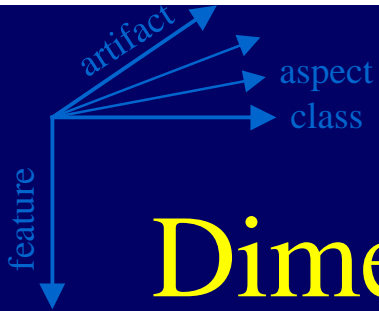
- For any concern, there is a set of units that “pertain to” or “affect” that concern
 - Their job is to contribute to the concern’s realization (e.g., display() methods pertain to Display feature)
- Possible definition:
 - A concern is a predicate over units
 - Want to express intention, not merely set of units (e.g., all methods named “display,” not just current set)





Dimensions of Concern

- Multiple kinds of concerns
 - Function, feature, class, aspect, variant, ...
- Each unit may pertain to multiple concerns
 - Method pertains to aspect and class(es)
 - `Plus.display()` pertains to class `Plus` and feature `Display`
- Kinds of concerns are *orthogonal*:
 - Method being in aspect `A` does not imply what class
 - Method being in feature `Display` does not imply what class
- Call them *dimensions*



Dimension Example

Feature

Display

Eval

Expression.display()

PlusImpl.display()

Expression.eval()

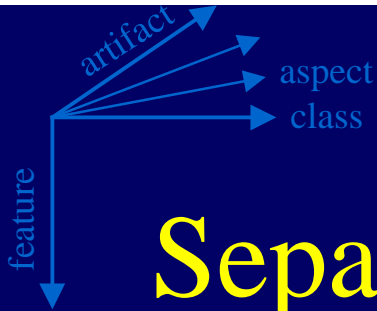
PlusImpl.eval()

Expression

PlusImpl

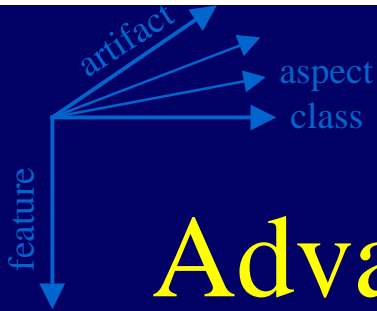
Class





Separation of Concerns

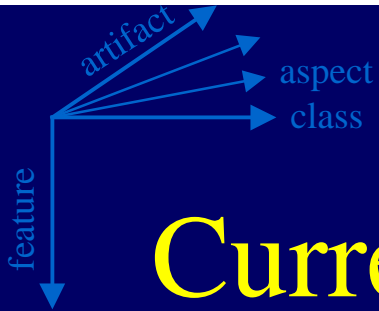
- Keep the units pertaining to different concerns separate
- *Module*
 - Collection of units pertaining to a concern
 - Module *encapsulates* that concern
 - Concern-based decomposition
- Compound units are modules
 - Provided by artifact languages



Advantages of SOC

- Traceability
 - Can find all units that pertain to separated concern
- Improved comprehensibility, reusability
 - Focus on one concern at a time
 - Concerns are useful units of reuse
- Evolvability, reduced impact of change
 - Module shields outsiders from changes to inner details
- Plug-and-play, mix-and-match
- ...

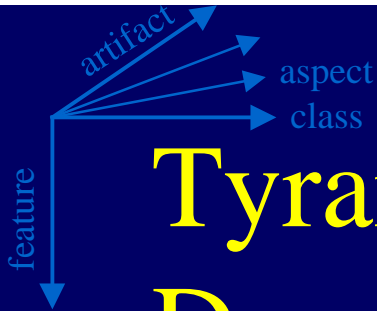




Current SOC Falls Short

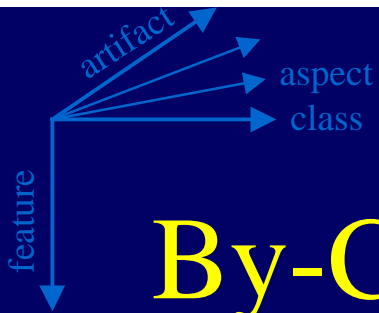
- Touted advantages are realized to only a very limited extent in practice
 - Despite much effort and ingenuity

- Why?



Tyranny of the Dominant Decomposition

- Artifact can typically be decomposed in just one, *dominant* way
 - Often determined by the language
 - OO permits just object (data) decomposition
 - Functional languages just functional decomposition
- So, separation can be achieved according to only one kind of concern (perhaps a few)
- Other concerns are not separated



By-Class Decomposition

Feature

Display

Eval

Expression.display()

PlusImpl.display()

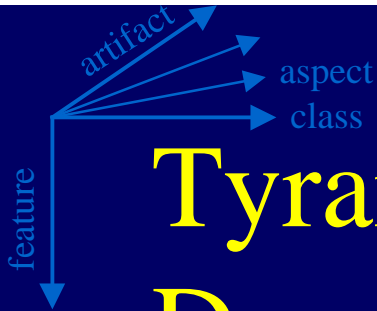
Expression.eval()

PlusImpl.eval()

Expression

PlusImpl

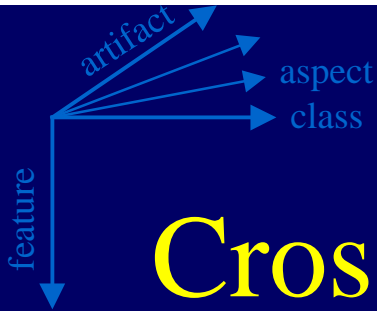
Class



Tyranny of the Dominant Decomposition

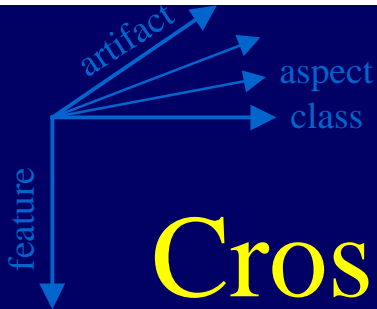
- Concern structure is multi-dimensional
- Module structure is 1-dimensional
- One dimension is dominant
 - Its concerns are encapsulated
- Others are secondary
 - Their concerns are not encapsulated





Cross-Cutting Concerns

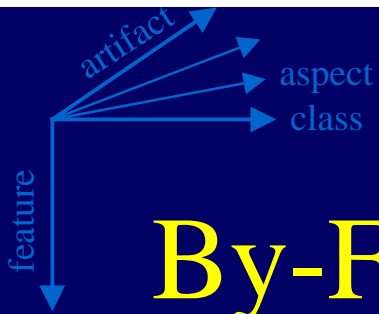
- Non-dominant concerns typically *cut across* modules encapsulating dominant concerns
 - *Scattering*: units affecting a single concern are scattered across many modules
 - *Tangling*: a single module contains units affecting many concerns in the same dimension
- No wonder the advantages of SOC are not achieved!



Cross-Cutting Modules*

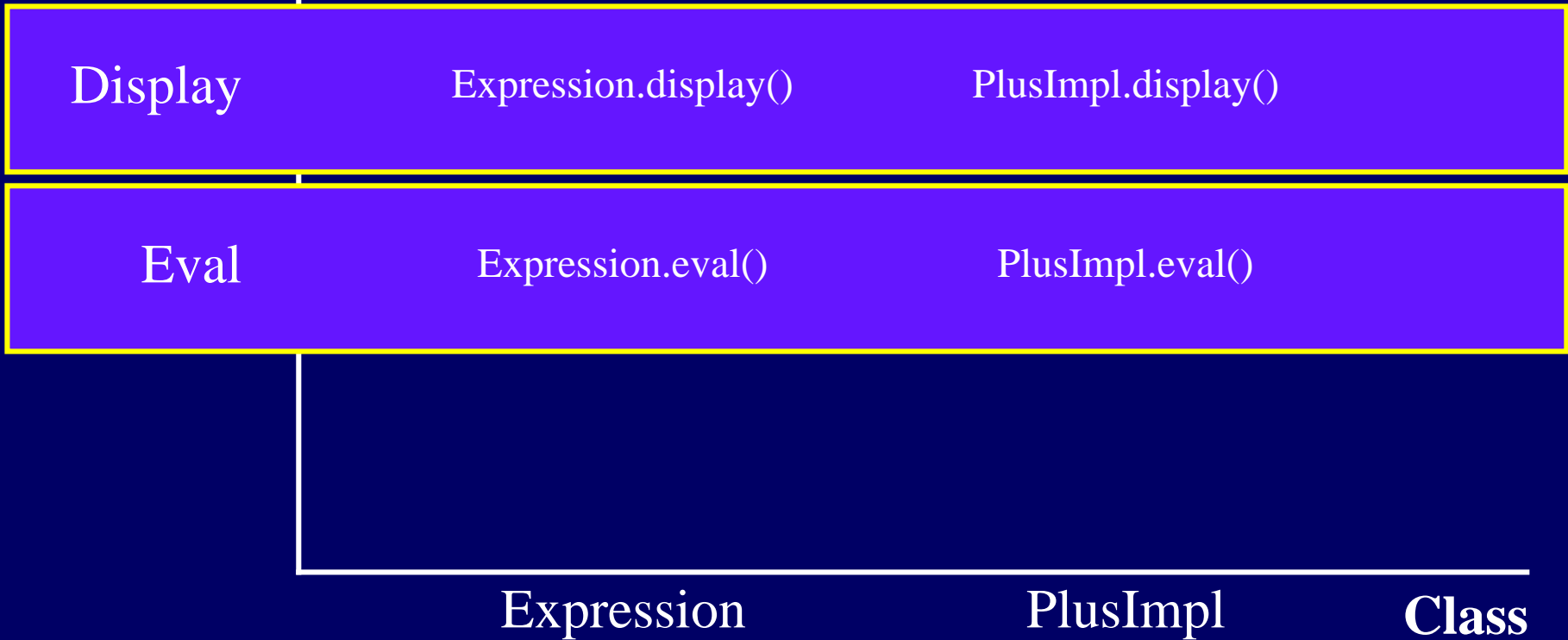
- Modules that can encapsulate cross-cutting concerns
- Usually not provided by artifact languages
 - Need new mechanisms
 - This community has produced several
 - Groundswell of interest and experimental use

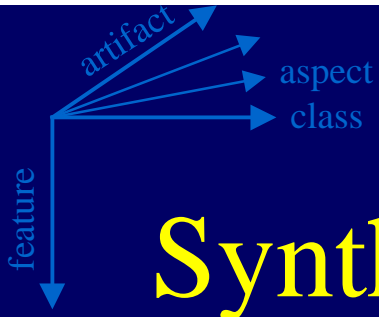
* Term due to Aspect-Oriented Programming group



By-Feature Decomposition

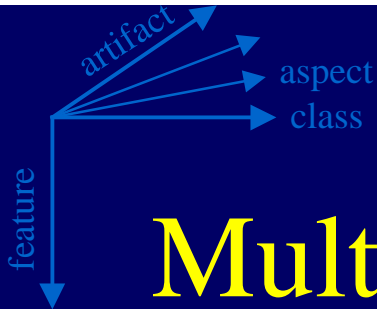
Feature





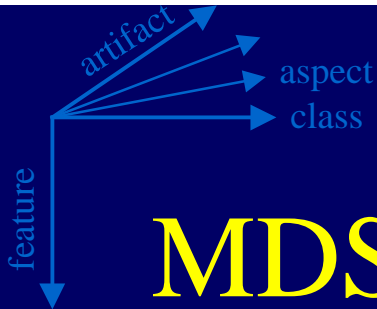
Synthesis

- For modules not in artifact languages
 - Extend the languages
 - Synthesize the modules to produce artifact modules
 - Manual or automatic
- Synthesis has generally been preferred
- Also called: *composition, weaving*



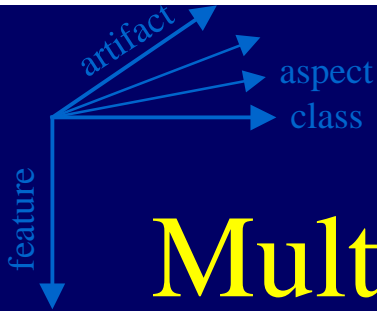
Multi-Dimensional SOC

- Cross-cutting modules
- Encapsulate multiple concerns
- Separation according to these *simultaneously*
 - No dominance
- Interactions between modules
 - Not independent



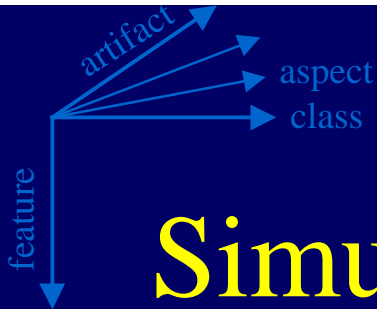
MDSOC Vision: Ultimate Goal

- Cross-cutting modules to the limit!
 - They also cut across one another, not just artifact modules
- Encapsulate multiple, *arbitrary* concerns
 - Addition/removal of new kinds/concerns at will
- Separation according to these *simultaneously*
 - No dominance
- Interactions between modules (not independent)
- Cross-lifecycle
 - Many concerns span artifacts



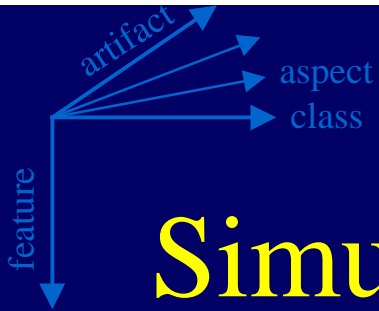
Multiple, Arbitrary Dimensions

- Multiple kinds of concerns
 - No restrictions
- Cut across one another
 - E.g., features and persistence in SEE example
 - Presence of persistence can affect each feature
- New dimension or just new concern?
 - Concerns in the same dimension don't cut across one another

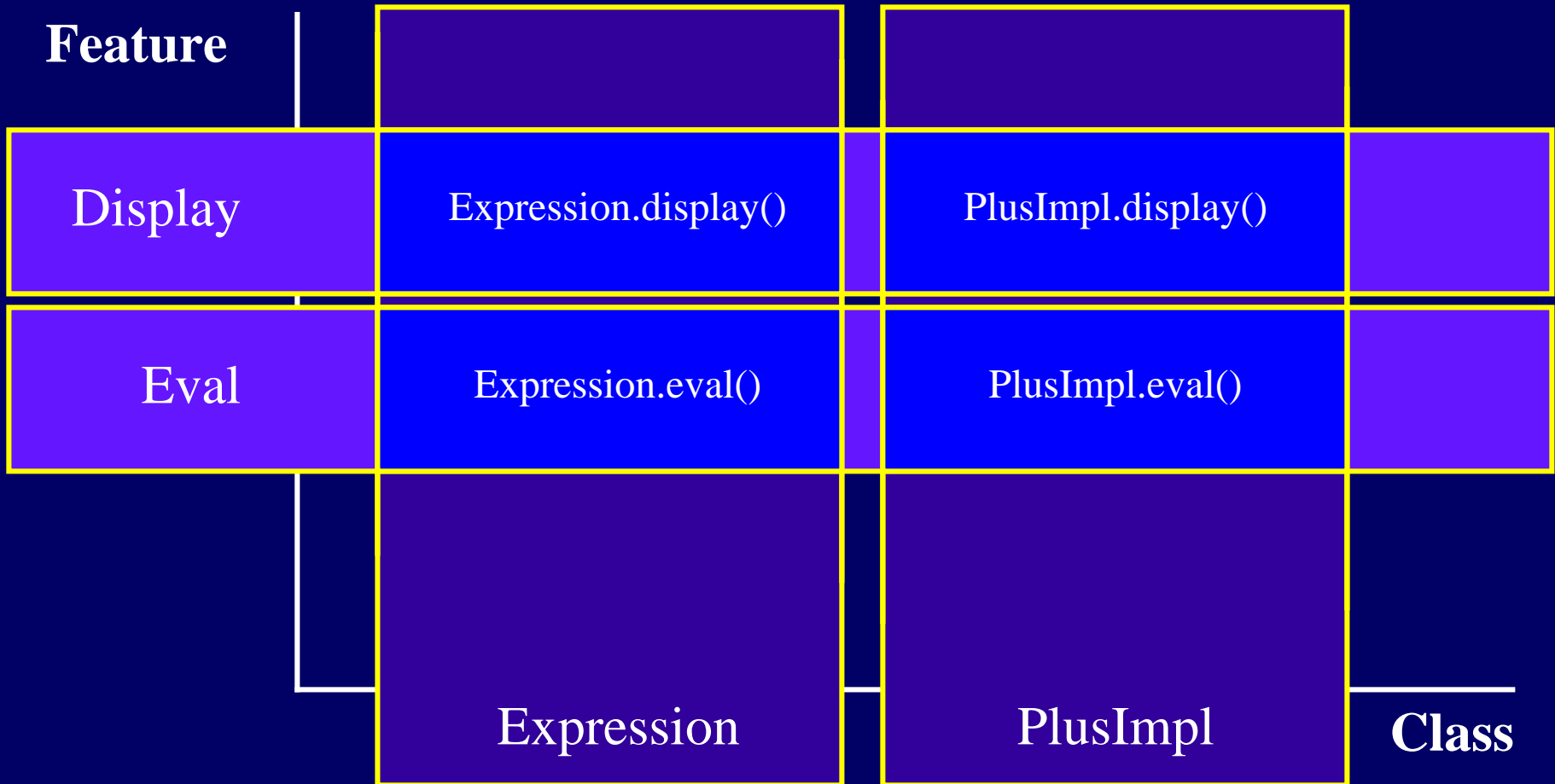


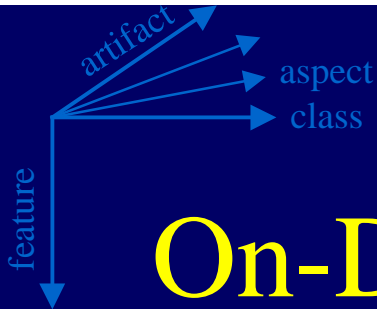
Simultaneous Separation

- Standard software is still linear
 - Each unit is in one place (class, filter, propagation pat., ...)
- *Mandated* decomposition
 - Language dictates decomposition (E.g., Java)
- *Choice* of decomposition: better
 - Mechanism offers choice, pick one and live with it (SOP)
- *Simultaneous* decomposition: the goal
 - Decompositions coexist, use as appropriate



Simultaneous Separation

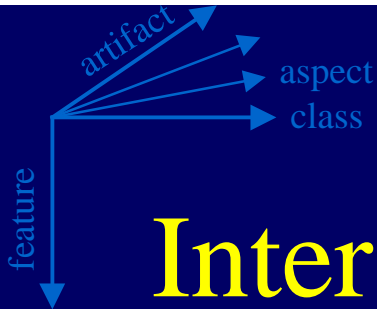




On-Demand Remodularization

- Development activity might uncover need for new decomposition
 - Work with classes for a while
 - Recognize features/aspects/variants tangled within classes
 - Want to tease them apart “post hoc”
- New decomposition without invasive change
 - Typically highly invasive, massive change (refactoring)
- Some major challenges

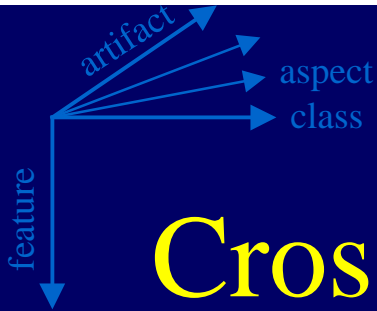




Inter-Module Interactions

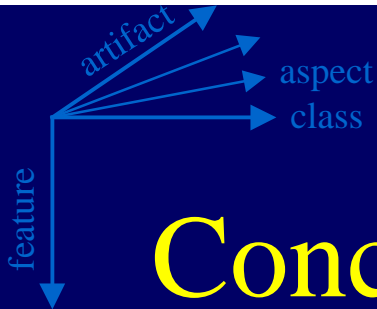
- Some concerns often considered independent
 - Concurrency, transactions, logging, ...
- But they seldom are
 - Presence/choice of one affects behavior of others
 - “Feature interaction”
- MDSOC mechanism must cope with interactions
 - What kinds of interactions are appropriate within a dimension? Across dimensions?





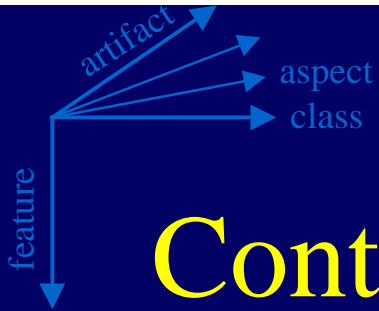
Cross Lifecycle

- Community focus has been on code (& design)
- Many concerns originate earlier or later
 - Requirements, test plans, units of change, variants, ...
- SOC is also important during other phases
- Many concerns span lifecycle phases
 - **Traceability**
- MDSOC can affect the software process
 - Wide-open area with great potential



Conclusion

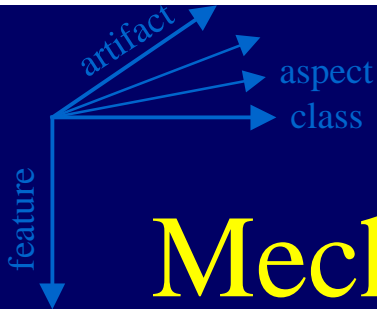
- Much recent good work on many sub-areas
 - Helped, opened field, raised awareness and interest
- MDSOC is an attempt to abstract and evolve
- MDSOC Vision is an attempt to extrapolate:
 - Ambitious set of goals
 - No mechanism does it all, yet
 - Shared understanding, agreement, disagreement
 - Research agenda



Contents

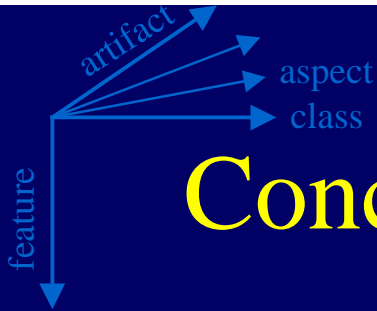
- Motivating Example
- Concepts and terminology
- Discussion
- **Mechanisms and Tradeoffs**





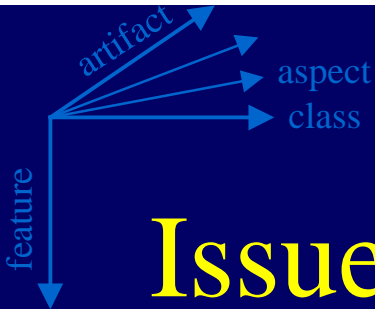
Mechanisms and Tradeoffs

- MDSOC describes a set of *goals* and *requirements*
 - A large space of possible approaches and mechanisms can be used to achieve MDSOC
 - Each point in the space has different properties
 - The properties may render them more or less suitable for use in a particular context
 - Satisfying requirements entails
 - Choosing some set of concerns to address
 - Making a number of design and implementation decisions and tradeoffs
 - Goals:
 - Identify key issues for MDSOC approaches
 - Identify key design and implementation decisions, tradeoffs
 - Begin to determine circumstances to which different mechanisms and tradeoff are better (or worse) suited
- ➔ **NOTE:** *We hold this truth to be self-evident: that no single mechanism can achieve all goals*



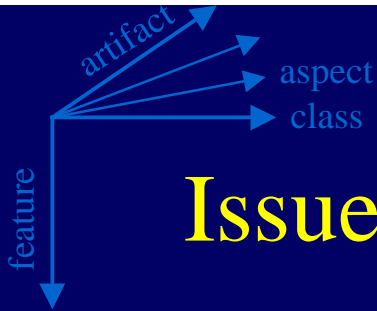
Concern Spaces: A Space of Choices

- Aid to analysis and comparison of MDSOC approaches
- Components of a concern space model:
 - Units
 - Dimensions and concerns
 - Modules
 - Relationships
 - Synthesis (composition/weaving) approach
- “Cross-cutting” issues
 - Mechanism approach
 - Linguistic, extra-linguistic
 - Design patterns
 - Reflection
 - ...
 - Selection time and strength
 - Fixed, fluid
 - Dynamic, static
 - Tight, loose
 - ...



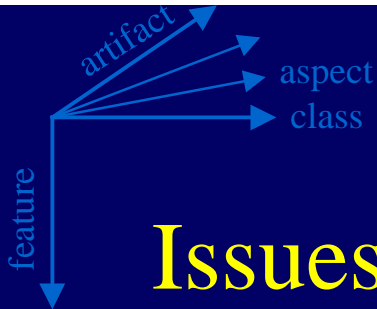
Issues: Units

- Units: The building blocks of software
 - Primitive and compound
 - Granularity of primitive units
 - member, statement, expression, token, role, pattern, section, sentence, oracle ...
 - ★ Method vs. sub-method
- “Cross-cutting”:
 - How are units described?
 - Underlying language(s) for describing units...
 - E.g., “Source,” “binary,” (formal) specification, structured text, ...
 - ...Or external mapping from constructs to units
 - E.g., “the sentences/tokens on lines 32-39 map to a unit called *U*”
 - When are units selected?
- Example tradeoffs and issues:
 - Using sub-method granularity \Rightarrow more flexible unit selection, but difficult to retain unit mappings during evolution, and difficult synthesis
 - Using standard languages for describing units \Rightarrow no learning curve, but may only be able to deal with a restricted set of unit types



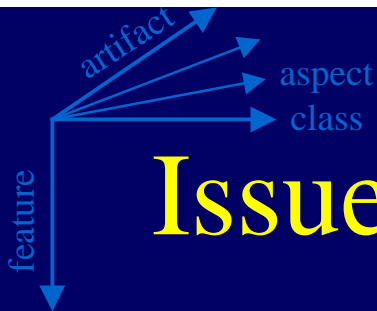
Issues: Dimensions, Concerns, Modules

- Dimensions: The decomposition approach
 - How many dimensions? Which ones? When are they applicable/used?
 - How are dimensions organized? (Hierarchical? Flat? Other?)
- Concerns: Sets of units of interest for some purpose
 - What kinds of concerns are supported?
 - Are any required or distinguished? E.g., distinguished base vs. symmetry?
 - When decided (A priori? A posteriori?)
 - What cross-cutting is supported?
 - Across artifacts? Within artifacts? Functional/non-functional?
 - How are concerns identified? When are they identified?
 - Extensionally, intentionally, by delta, by extraction, ...
 - Requirements, design, analysis, architecture, implementation, testing, evolution, ...
- Modules: The means of encapsulating concerns
 - How are concerns encapsulated?
 - Are concerns (reusable) first-class entities?



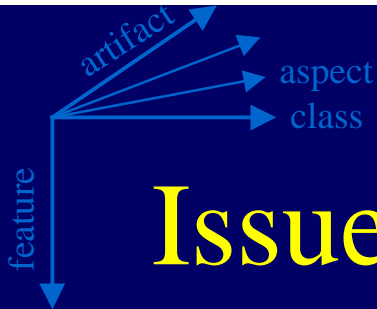
Issues: Dimensions, Concerns, Modules

- “Cross-cutting”:
 - How are the concerns and modules represented?
 - Using underlying language constructs? Provided by SOC mechanism?
 - When are dimensions, concerns, modules selected?
 - What do dimensions, concerns, modules know about each other?
- Example tradeoffs and issues:
 - One (few) dimension(s) \Rightarrow conceptually simple but expressively limited (tangling, scattering)
 - Orthogonality of concerns/dimensions



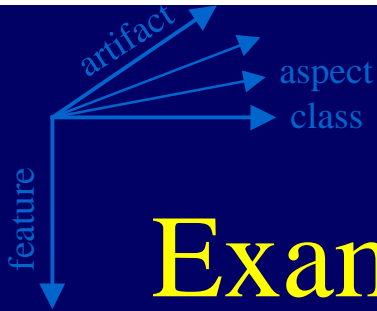
Issues: Relationships

- Relationships: (Semantic) connections and interactions among concern space entities
- What kinds of relationships can be specified?
 - Concern-to-unit
 - Concern-to-concern
 - Module-to-module
 - Concern-to-module
 - Concern-to-dimension
 - Etc.
- What concern interactions are supported?
 - What can concerns know about one another? When can they know it?
- “Cross-cutting”:
 - How and when are relationships described?
 - Determined by artifact or separately specified?
 - When is the relationship specified/determined? Can it be changed?
 - Changeable non-invasively? Which modules encapsulate which concerns?
- Example tradeoffs and issues:
 - Concerns (etc.) know about each other \Rightarrow encapsulate dependencies with concerns, but limit or eliminate reusability in other contexts (coupling)
 - Dynamic relationship specification \Rightarrow more flexibility but may require on-demand modularization and/or software restructuring



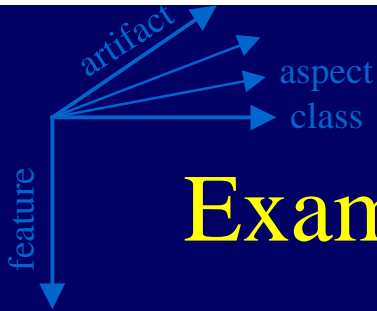
Issues: Synthesis

- What kinds of units, concerns, modules (other?) can be integrated (join points)? How can they be integrated?
 - Merge/union
 - Name/structural equivalence
- How is synthesis specified? E.g.,
 - Within modules/concerns/units? Separately?
- When is synthesis performed?
 - “Source,” load time, run time, ...
- Special properties, limitations
- Example tradeoffs and issues:
 - Specifying synthesis within concerns, etc., encapsulates integration issues, but reduces reuse and increases coupling
 - “Source-level” integration may be easier to analyze, predict, but may not be powerful to capture important semantics (e.g., role acquisition) and doesn’t work if “source” isn’t available (e.g., COTS software)



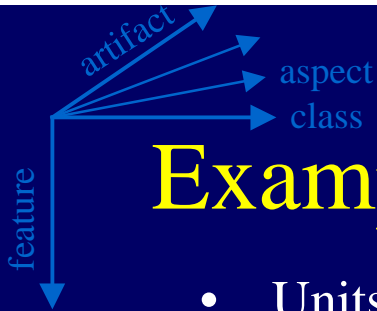
Example Concern Space: Java™

- Units: packages, classes, interfaces, members
- Dimensions and concerns: Data (one dimension)
- Modules: packages, classes, interfaces
- Some Relationships
 - Modules encapsulate data concerns, “imports”
 - No cross-cutting (NB: design patterns provide limited help)
 - A member is in exactly one class or interface
- Synthesis: compilation and class loading



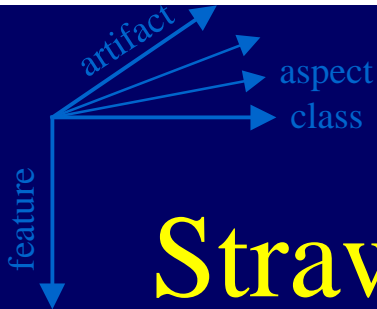
Example: Subject-Oriented Programming

- Units: as for OO
- Kinds of concerns supported: data, views (any)
- Modules: as for OO, and subjects
- Some Relationships
 - Subjects cut across classes, and other subjects
 - Inter-subject interactions specified by external composition rules
 - Subjects do not know about each other
 - A member is in one class in one or more subjects
- Synthesis:
 - Compile time or load time
 - Controlled by composition rules
- “Cross-Cutting”
 - Subjects are coded explicitly; can’t be changed without recoding



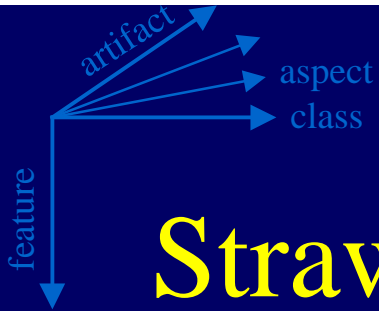
Example: Hyper/J™ (Hyperspaces for Java)

- Units: as for Java
- Kinds of concerns supported: data, any others
- Modules: as for Java, and *hyperslices*
- Some Relationships
 - Hyperslices cut across classes, and other subjects
 - Inter-hyperslice interactions specified by external composition relationships
 - Hyperslices do not know about each other
 - A member is in one class in one or more hyperslices
- Synthesis:
 - Load time (class files)
 - Hyperslices composed into *hypermodules*
 - Controlled by composition relationships
- “Cross-Cutting”
 - Concerns specified by flexible *concern mapping*
 - Hyperslices can be extracted from standard Java code, based on concerns
 - New decompositions without recoding



Strawman: AspectJTM

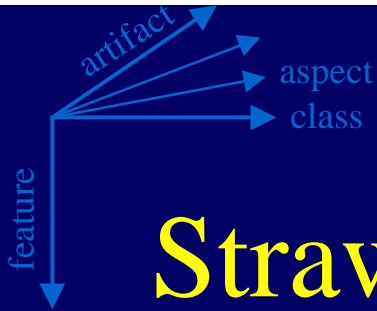
- Units: as for Java
- Kinds of concerns supported: data, aspects (any?)
- Modules: as for Java, and aspects
- Some Relationships
 - Aspects cut across classes, but not across one another
 - Aspects are independent (no aspect-aspect relationships)
 - A member is in one class/interface, or in an aspect from which it is woven into classes/interfaces
 - Aspects know about the “base”; relationships specified internally
- Synthesis:
 - Source code is woven based on weave specifications in aspects



Strawman: Composition Filters

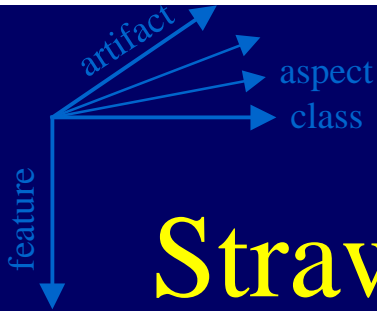
- Units: as for OO, filters
- Kinds of concerns supported: data, aspects (any?)
- Modules: as for OO. Filters are individual objects
- Some Relationships
 - Filters are attached to objects
 - Attached filters are ordered and cascaded
 - A member is in one class, or in a filter attached to (multiple) objects of (multiple) classes
- Synthesis
 - Dynamic configuration of filter objects





Strawman: Propagation Patterns

- Units: as for OO, propagation patterns
- Kinds of concerns supported: data, traversals (any?)
- Modules: as for OO, and collections of patterns
- Some Relationships
 - Patterns specify traversals of object networks, and functions to execute at visited objects
 - A function is in one class, or in a propagation pattern
- Synthesis
 - Standard OO program is generated from class graph and patterns



Strawman: Catalysis

- Units: as for OO, design included
- Dimensions and concerns: data, roles
- Modules: OO, and role models, packages and frameworks
- Some Relationships
 - Role models and classes cut across one another
- Synthesis:
 - Synthesis of classes from role models
 - Framework composition