

# Lecture 8

- Style rules: Class graph minimization
- Evolution of object behavior
- Current issues in Software Engineering:  
The year 2000 crisis

# Algorithmical/theoretical nuggets

- Software development is a hard problem.
- Pick out well-defined subproblems which can be solved algorithmically or which can at least be understood better. E.g.
  - class dictionary minimization/transformation
  - programming traversals
  - class graph learning
  - strategy minimization

# Class graph minimization

- Definition: size of class graph =  
number of construction edges +  
 $1/4$  \* number of alternation edges
- Why  $1/4$ ?
  - encourage use of inheritance, factoring out commonality is good!
  - simplifies algorithm, any constant  $< 1/2$  also works.

# Class graph minimization

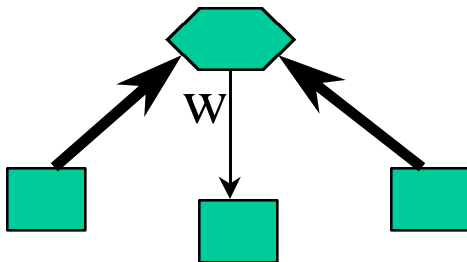
- Fruit : Apple | Orange

\*common\* Weight.

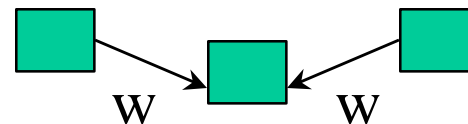
Apple = .      Orange = .       $1 + 0.5 = 1.5$  ( $1+2=3$ )

- Apple = Weight.

Orange = Weight. <sub>2</sub>



11/19/98



AOO/Demeter

# Class graph minimization

- Problem:
  - input: class graph  $G$
  - output: class graph  $H$  of minimum size and object-equivalent to  $G$
- An NP-hard minimization problem: at least as hard as any problem in NP.

# Complexity theory excursion

- Problem kinds:
  - decision problems: Is  $x$  in  $X$ ?
  - optimization problem (considered here): For  $x$  in  $X$  find smallest (largest) element  $y$  in  $X$  with property  $p(x, y)$ .
- Decision problems:
  - Is Boolean formula always true?
  - Is class dictionary ambiguous?
  - Are two class graphs object-equivalent?

# Levels of algorithmic difficulty

- Unsolvable: no algorithm exists
  - Is class dictionary ambiguous?
  - Define 2 class dictionaries the same language?
- Only slowly solvable (no polynomial-time algorithm exists or is currently known)
  - Is Boolean formula always true? (co-NP hard)
- Efficiently solvable (polynomial-time)
  - Are two class graphs object-equivalent?

# Word of caution

- Complexity theory is an asymptotic theory.
- All algorithmic problems of finite size can be solved by a computer (Turing machine).
- But all practical algorithmic problems are of finite size.
- Complexity theory still practically very useful: It guides your search for algorithms.

# Other NP-hard problems

- Has a Boolean formula a satisfying truth assignment? (in NP, hence NP-complete)
- Exists there a class graph  $x$  which is object-equivalent to  $y$  but of smaller size? (in NP)
- Can we color the nodes of a class graph with three colors so that no two adjacent nodes have the same color. (in NP)
- Strategy minimization (class graph known).

# Why are they all NP-hard?

- They can be reduced to one another by polynomial transformations similar to the transformations:
  - A class graph which is not flat can be transformed into an object-equivalent one which is flat and by at most squaring the size.
  - Law of Demeter transformation: a program which violates LoD can be transformed to satisfy LoD with small increase in size.

# Class graph minimization

- Focus on class graphs allowed by single inheritance languages, like Java and Smalltalk.
- Definition: A class graph is single-inheritance if each class has at most one incoming subclass edge.
- Problem: Is there an object-equivalent class graph  $G'$  for  $G$  which is single inheritance?

# Class graph minimization

```
ChessPiece : Queen | King | Rook |  
           Bishop | Knight | Pawn.  
Officer   : Queen | King | Rook.
```

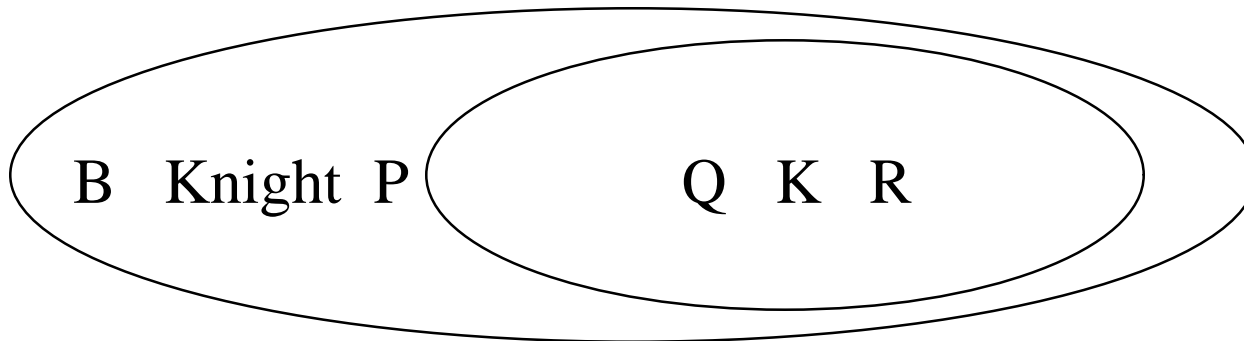
```
ChessPiece : Officer |  
           Bishop | Knight | Pawn.
```

Class graph becomes single inheritance,  
Object-equivalence preserved.

# Class graph minimization

```
ChessPiece : Queen | King | Rook |  
            Bishop | Knight | Pawn.  
Officer   : Queen | King | Rook.
```

```
ChessPiece : Officer |  
            Bishop | Knight | Pawn.
```

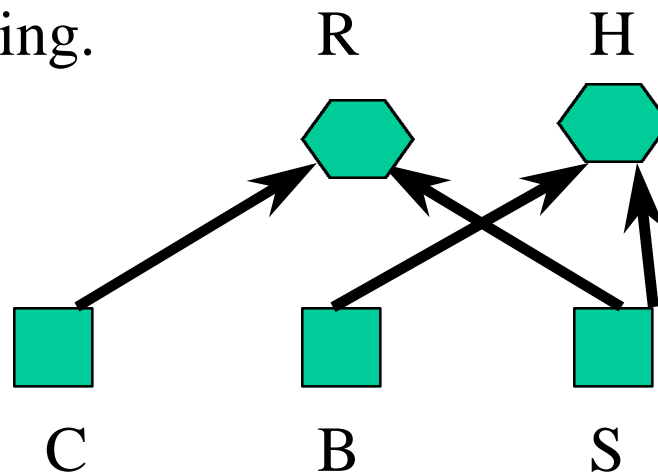
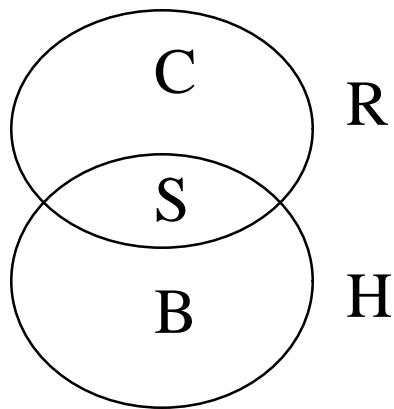


# Requires multiple inheritance?

RadiusRelated : Coin | Sphere \*common\* Radius.

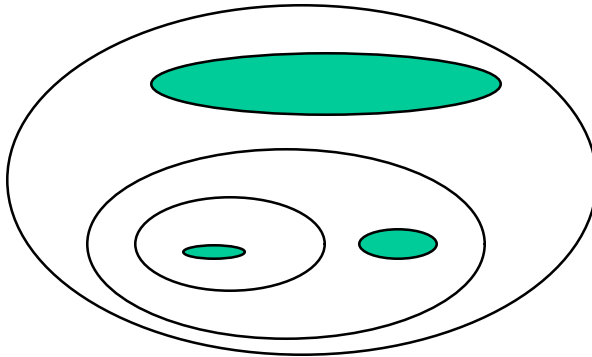
HeightRelated : Brick | Sphere \*common\* Height.

Why? not all or nothing.

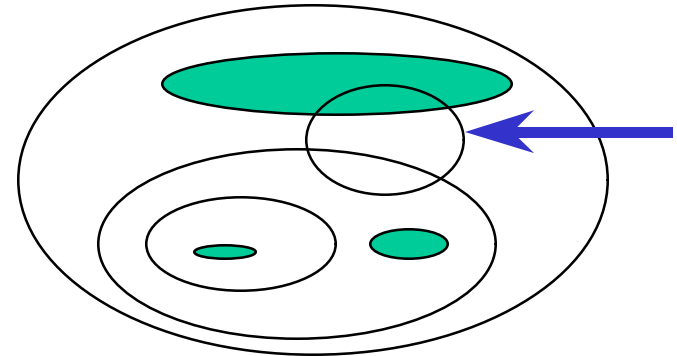


# Tree property - all or nothing

yes



no



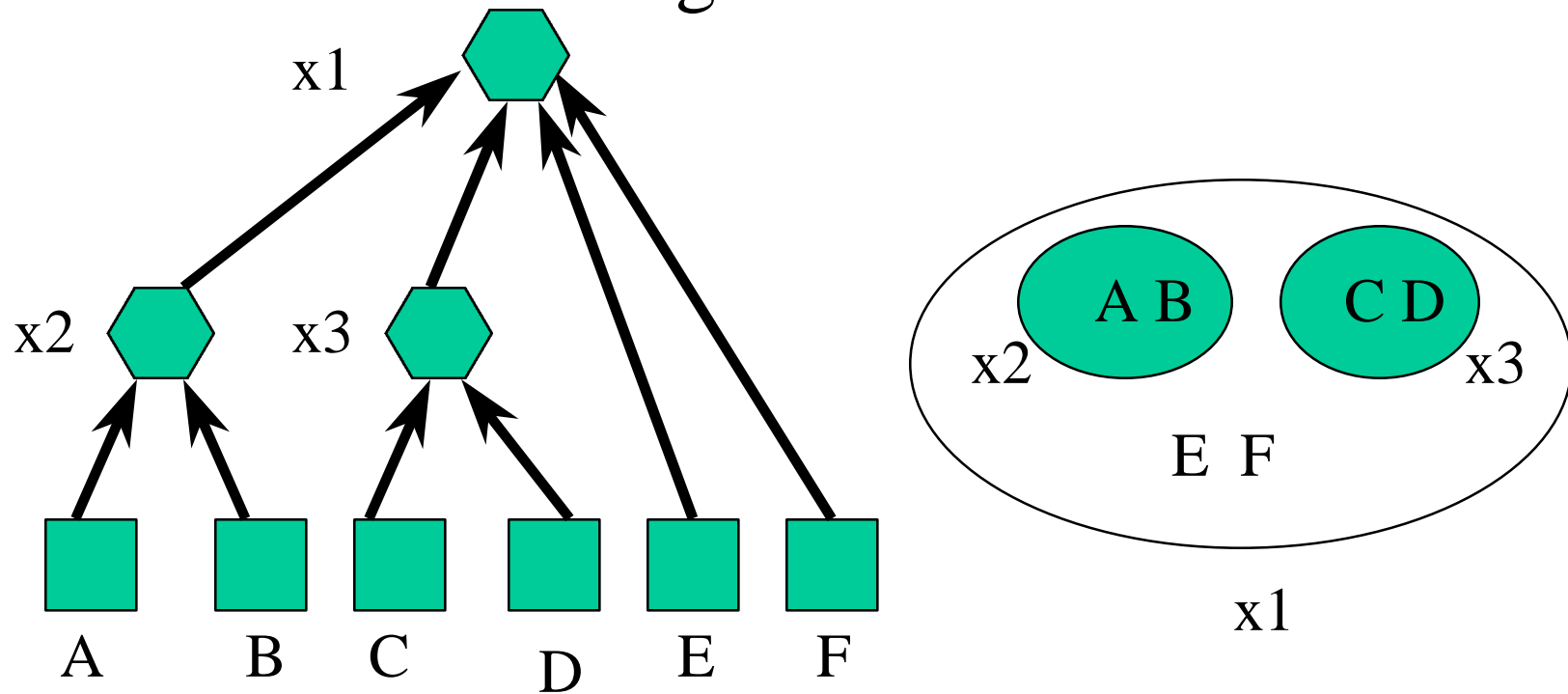
A collection of subsets of a set satisfies the tree property, if for any two subsets either one contains the other or the two are disjoint.

# From multiple to single inheritance

- G is object-equivalent to a single-inheritance class graph if and only if the collection of concrete subclass sets of G satisfies the tree property.
- The collection of concrete subclass sets of G is the collection of subsets consisting of all concrete subclasses of classes in G.

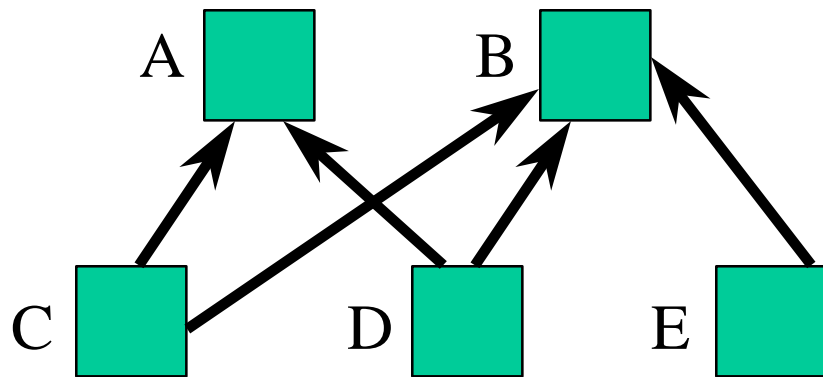
# Algorithm

- Containment relationships between subclass sets determine single inheritance structure.

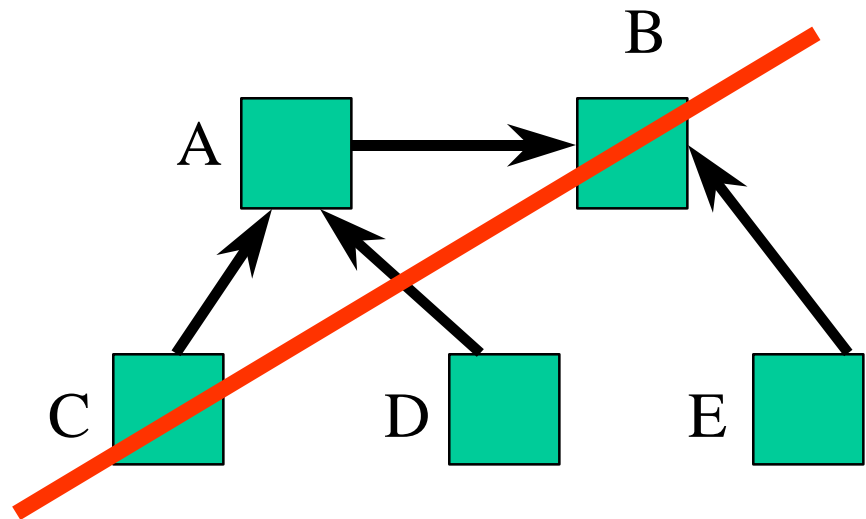


# Abstract Super Class Rule not needed

A, B now concrete  
A inherits from B  
not object-equivalent



A C D  
B C D E



# Class graph minimization

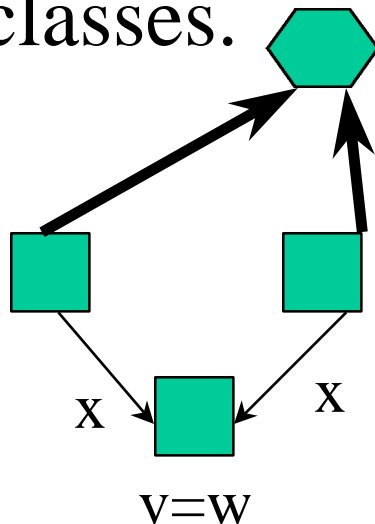
- Problem: NP-hard
  - input: class graph  $G$
  - output: class graph  $H$  of minimum size and object-equivalent to  $G$
- Problem: Polynomial
  - input: class graph  $G$
  - output: class graph  $H$  with minimum number of construction edges and object-equivalent to  $G$

# Class graph minimization

- Achieve in two steps:
  - Minimize number of
    - construction edges (polynomial)
    - alternation edges (NP-hard)

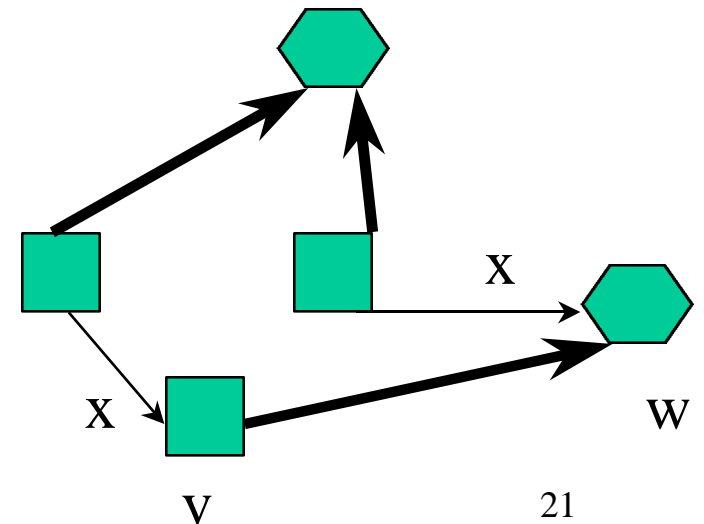
# Class graph minimization/ construction edge minimization

- A construction edge with label  $x$  and target  $v$  is redundant if there is a second construction edge with label  $x$  and target  $w$  such that  $v$  and  $w$  have the same set of subclasses.



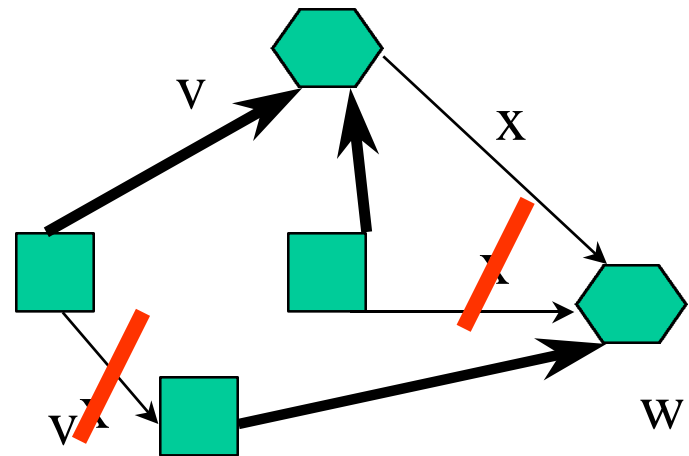
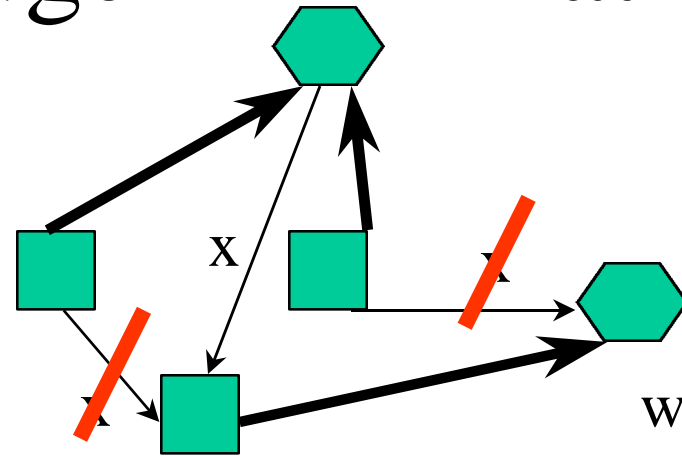
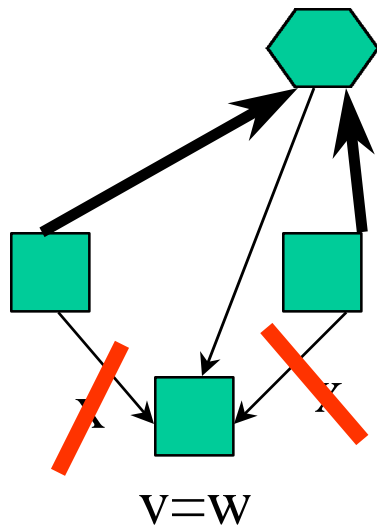
11/19/98

AOO/Demeter

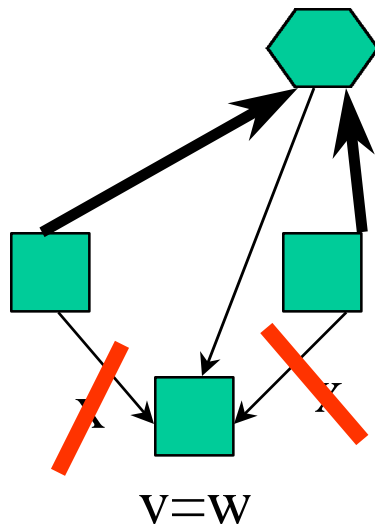


21

# Class graph minimization/ construction edge minimization



# Class graph minimization/ construction edge minimization



Abstraction of common parts  
solves construction edge  
minimization problem:

Are there any redundant parts?

- yes: attach them to an abstract class, introduce a new one if none exists.
- no: minimum achieved

# Recall: Class graph minimization

- Definition: size of class graph =  
    number of construction edges +  
     $1/4$  \* number of alternation edges
- Why  $1/4$ ?
  - encourage use of inheritance, factoring out commonality is good!
  - simplifies algorithm, any constant  $< 1/2$  also works.

# Recall: Class graph minimization

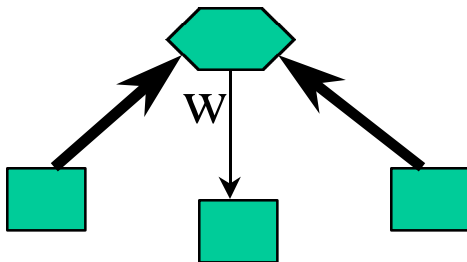
- Fruit : Apple | Orange

\*common\* Weight.

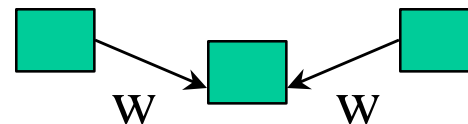
Apple = .      Orange = .       $1 + 0.5 = 1.5$  ( $1+2=3$ )

- Apple = Weight.

Orange = Weight. <sub>2</sub>



11/19/98



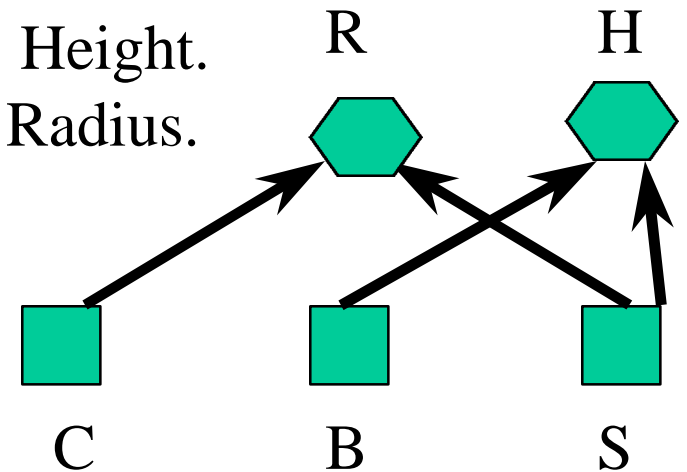
AOO/Demeter

25

# Problem:

- Redundant part elimination (= abstraction of common parts) may lead to multiple inheritance.

HeightRelated : Brick | Sphere \*common\* Height.  
RadiusRelated : Coin | Sphere \*common\* Radius.



# Class graph minimization

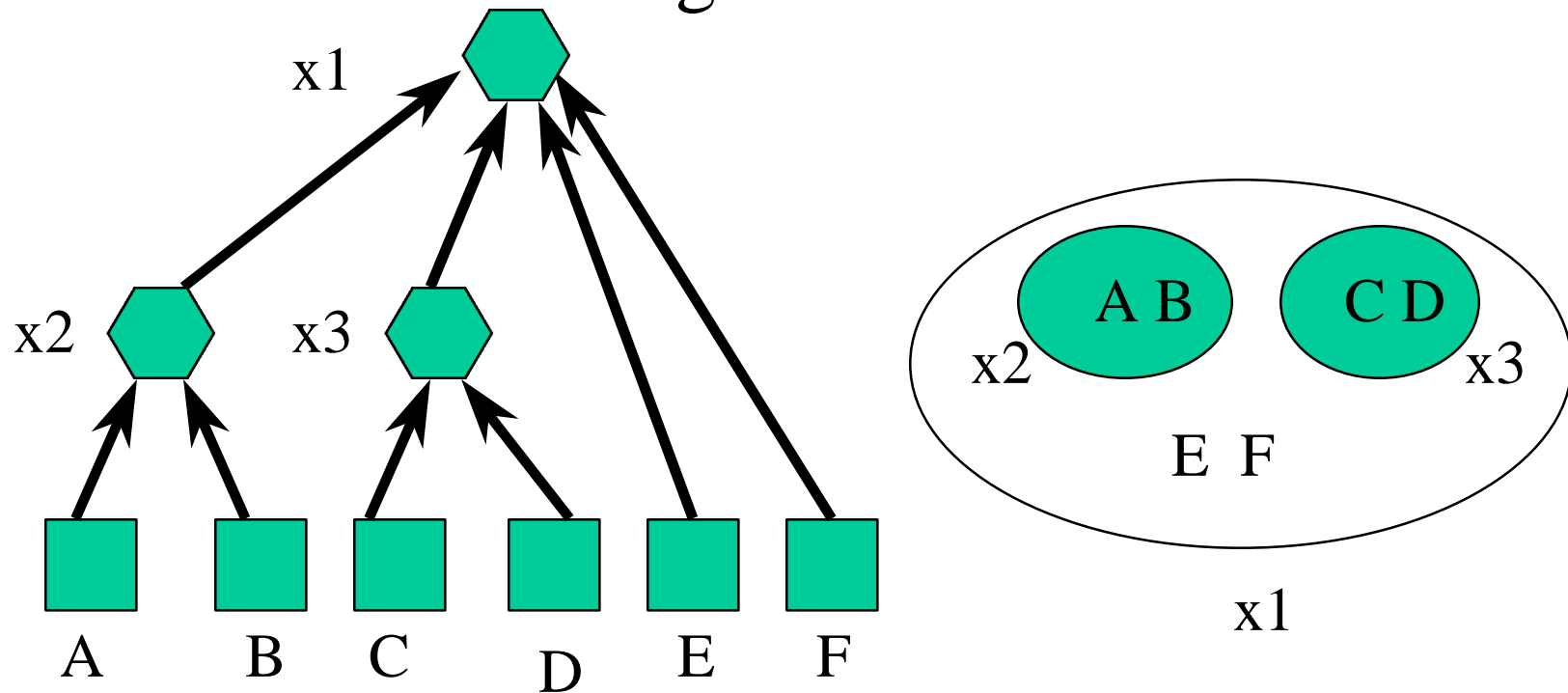
- Problem: NP-hard
  - input: class graph  $G$
  - output: class graph  $H$  of minimum alternation-edge size and object-equivalent to  $G$
- Problem: Polynomial
  - input: class graph  $G$  with tree property
  - output: class graph  $H$  with minimum number of alternation edges and object-equivalent to  $G$ .  
construction edges  $G =$  construction edges  $H$ .

# Algorithm

- Construct collection of concrete class subsets
- Subset containment relationships determine inheritance structure
- Minimum graph will be single inheritance

# Algorithm

- Containment relationships between subclass sets determine single inheritance structure.



# Summary: class graph min.

- Simple algorithms for
  - minimizing construction edges
  - finding object-equivalent single inheritance class graph
  - minimizing alternation edges provided tree-property holds
- Those algorithms can be easily applied manually during OOD.

# Summary: class graph min.

- Problem to watch out for: minimizing construction edges may introduce multiple inheritance
- Multiple inheritance can always be eliminated by introducing additional classes

# Change of Topic

- Context objects

# Evolution of object behavior

- What is behind design patterns like: Bridge, Chain of Responsibility, Decorator, Iterator, Observer, State, Strategy, Visitor
- Dynamic variation of behavior
- Need patterns since there are not adequate language constructs in programming languages

# Context relation

- Supports dynamic behavioral evolution while maintaining safety and performance benefits
- Context relation orthogonal to inheritance relation
- Dynamically alter single object or a class

# Context Relation

- Design patterns
  - Creational
    - abstract instantiation process
  - Structural
    - abstract object composition
  - Behavioral
    - abstract object communication and responsibilities
- Structural and behavioral benefit

# Context Relation

- Dynamically alter an object
  - Bridge, Chain of Responsibility, Strategy, etc.
- Dynamically alter a class
  - Iterator, Visitor

# Context Relation

- Three basic concepts to safely achieve dynamic behavior at both the object and class level:
  - Instance-stored versus class-stored specification
  - Dynamic specification
  - Dynamic update

# Instance-stored versus class-stored

- Many OO languages distinguish between
  - class variables and methods (static)
    - methods invoked through the class, no implicit `this`
  - instance variables and methods
    - methods invoked with class instance, implicit `this`

# Instance-stored versus class-stored

- Both instance and class methods can be considered class-stored: each class has conceptually one virtual method table
- Java currently only supports class-stored methods
- Need variations on a per-object basis.

# Instance-stored versus class-stored

- Instance methods may vary on a per-object basis: instance-stored methods
- Instance methods may vary on a per-class basis: class-stored methods
- Class methods may vary on a per-class basis

# Method declarations

| Method                | Keyword               | <code>this</code> | Virtual table   |
|-----------------------|-----------------------|-------------------|-----------------|
| <code>class</code>    | <code>class</code>    | no                | class-stored    |
| <code>instance</code> |                       | yes               | class-stored    |
| <code>instance</code> | <code>instance</code> | yes               | instance-stored |

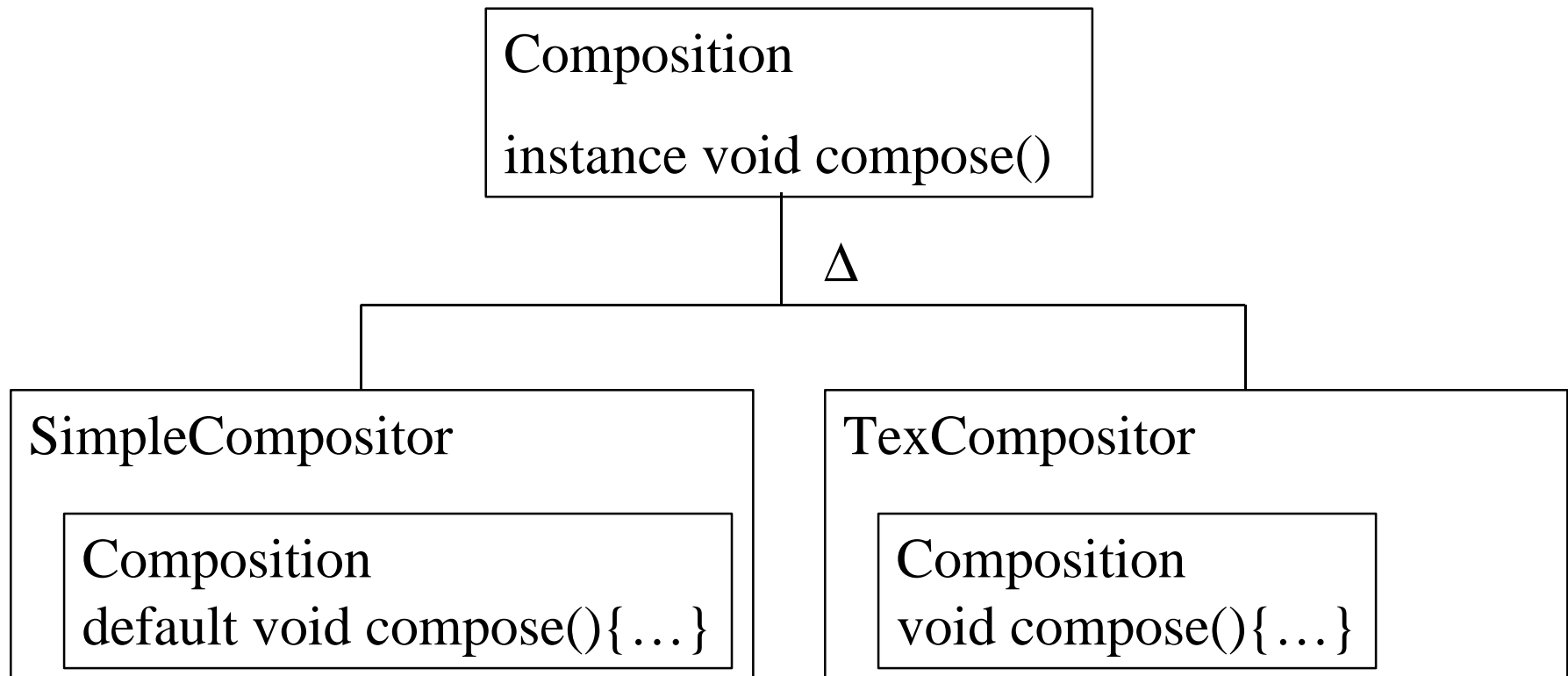
# Examples

- `class void f()`  
    `{ /* no this */ }`
- `void f() { .. this .. }`
- `instance void f()`  
    `{ .. this .. }`

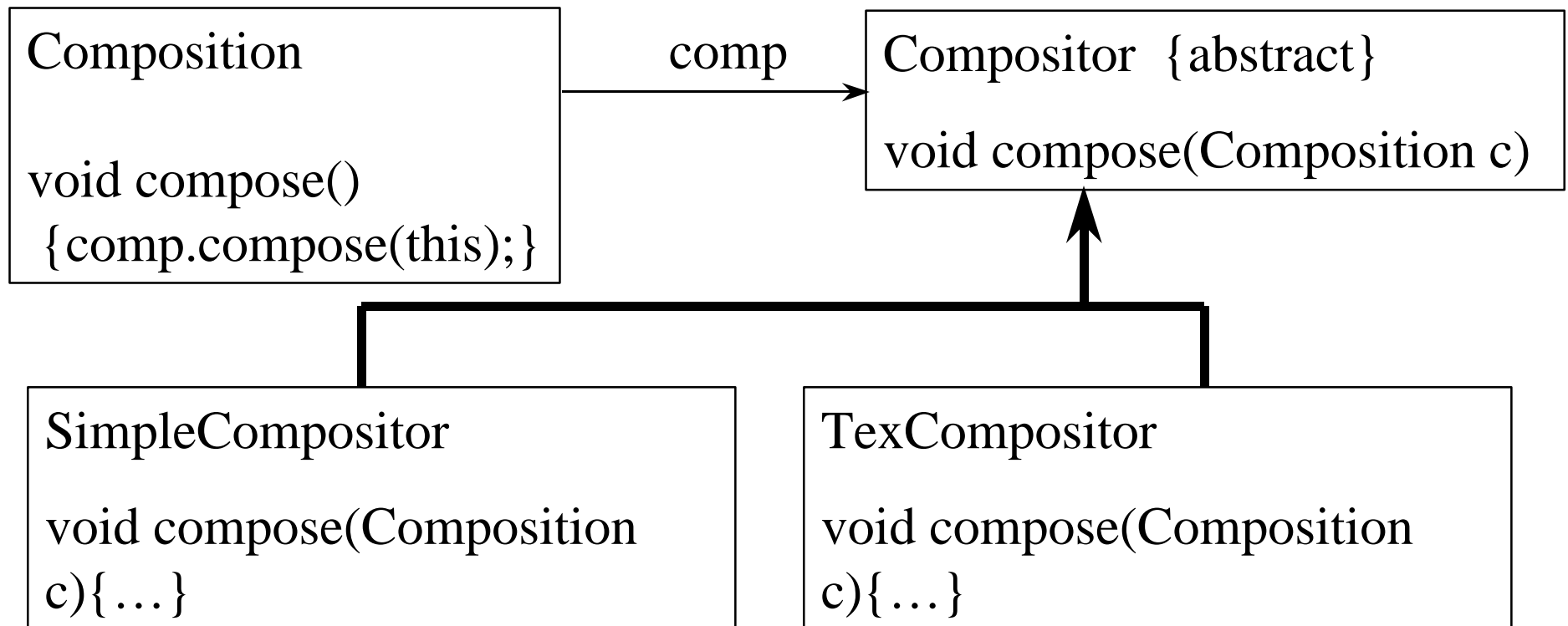
# Dynamic specification

- Specify with context relation
- Context relation links a class with its dynamic variations
- Use delta symbol
- Use Strategy Design pattern as an example (has nothing to do with traversal strategies)

# Context relation



# Without context relation



# Context classes

- A context class does not inherit from a base class, nor is it considered a subclass
- It is also not considered an abstract super class
- A context class defines instance variables and methods of its instances: it is a class

# Design language for Java

| Java extension  | Purpose               |
|---|-----------------------|
| <code>class C<br/>  {update class<br/>    B {} }</code> | dynamic specification |
| <code>instance</code>                                   | instance-stored       |
| <code>default</code>                                    | default impl.         |
| <code>context</code>                                    | meta variable ref.    |
| <code>::=</code>  | context attachment    |

# Dynamic update

- Update the methods for an object or class
- Requires virtual method tables to be dynamic
- To alter method tables: context attachment
- attach context object to object or class

# Other Design Patterns

- State
  - an object in different states may react differently to the same message. Without context relation, need association and inheritance.
- Bridge, Chain of Responsibility, Decorator and Observer can also be simplified with context relation.

# Another use of context objects

- Modify a group of classes for the duration of a method invocation: attach context to method invocation
- Does this sound or look familiar?

# Visitors as special context objects

- `e.visit{inv}(); // inv: inventory visitor`
  - the visit method should be executed within the context of the inventory object `inv`. Updates the application classes for the duration of invocation
- `e.visit(inv); // in Demeter/Java`

# Alternative view

- In Demeter/Java we have adaptive methods:  
$$A \{ R \ f() \ \text{to} \ X \ (V1, V2) ; \dots \}$$
- This seems to be the preferred way of programming by the Demeter/Java team.
- This style is about class level behavior modifications and does not have to use visitor objects for implementation.

# How can we improve adaptive methods?

- Need mechanism to communicate between visitors.

# Current issues in Software Engineering: The year 2000 crisis

- Problem: two digit date fields will brake programs
- Problem goes deeper: related to problems with abstraction, information hiding, modularity and reuse
- Opportunity: rewrite the old systems to make them more flexible: Rewrite them in Demeter/Java or in a traversal visitor style?

# The millennium opportunity

- Identify abstractions. Extract from the legacy code not just data, but more ambitious abstractions like scheduling policies or lending guidelines.
- Enforce OO principles. Make no concessions regarding information hiding.
- Leave hooks, places where new mechanisms can be plugged in later.

# The millennium opportunity

- Be dogmatic about reuse. Repetition is one of the worst enemies of software. Whenever you spot duplication, kill it immediately. How do you kill duplication of structure? Use class graphs and strategies.
- Ref: Adapted from IEEE Computer, Nov. 1997, pages 137-138.

# The End

# Style rules

- class graphs
  - Inductiveness, kinds of class dictionaries
  - Boyce-Codd normal form
  - Buffer-Rule
- methods
  - Law of Demeter