Using Evolutionary Computation to Automatically Refactor Software Designs to Include Design Patterns

Adam C. Jensen

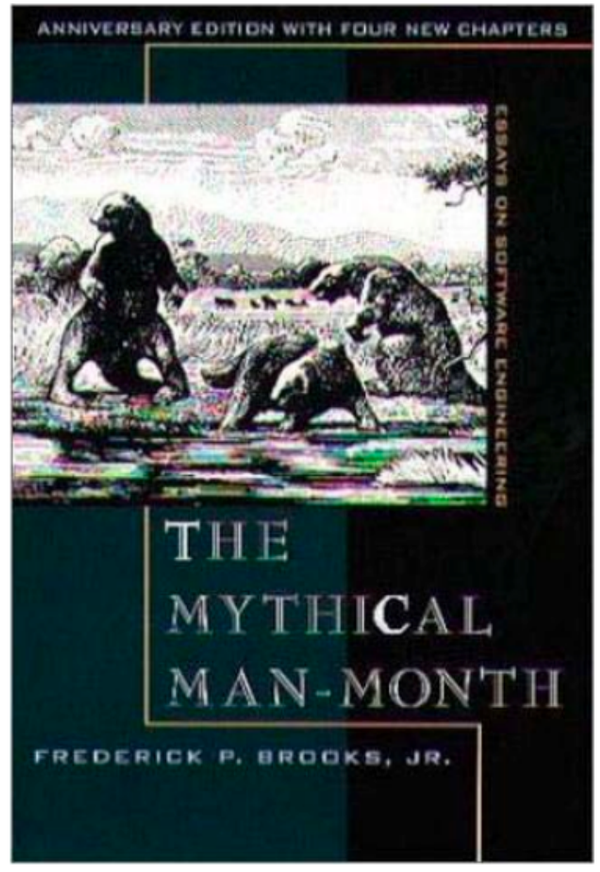
M.S. Thesis Defense

November 30, 2009

The Big Picture

"The total lifetime cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it."

Fred Brooks, *The Mythical Man Month*



The Big Picture

"... unless there is a clear reason to prefer the simpler solution, it is probably wise to choose the flexibility provided by the design pattern solution because unexpected new requirements often occur."

Prechelt *et al.*

L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering, pages 1134–1144, 2001.

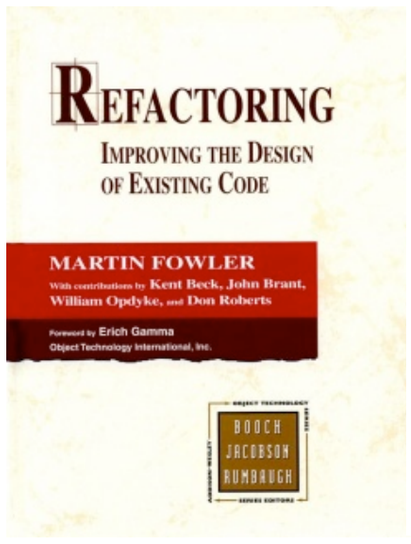
The Problem

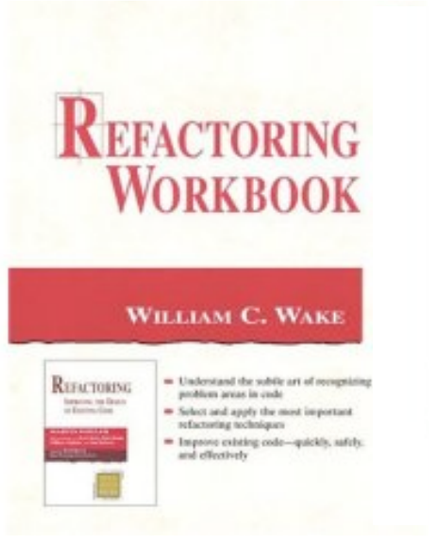
Refactoring is hard,

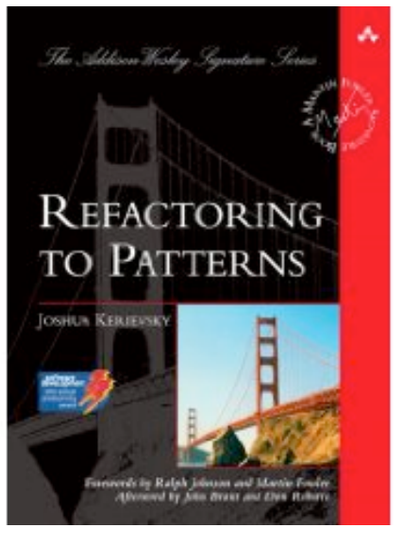
tedious,

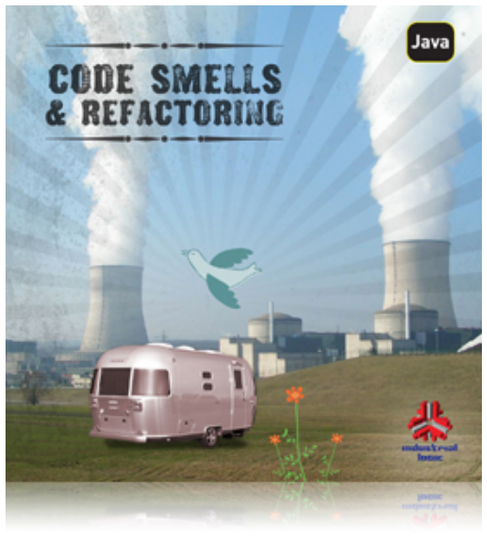
error-prone,

and *necessary*.





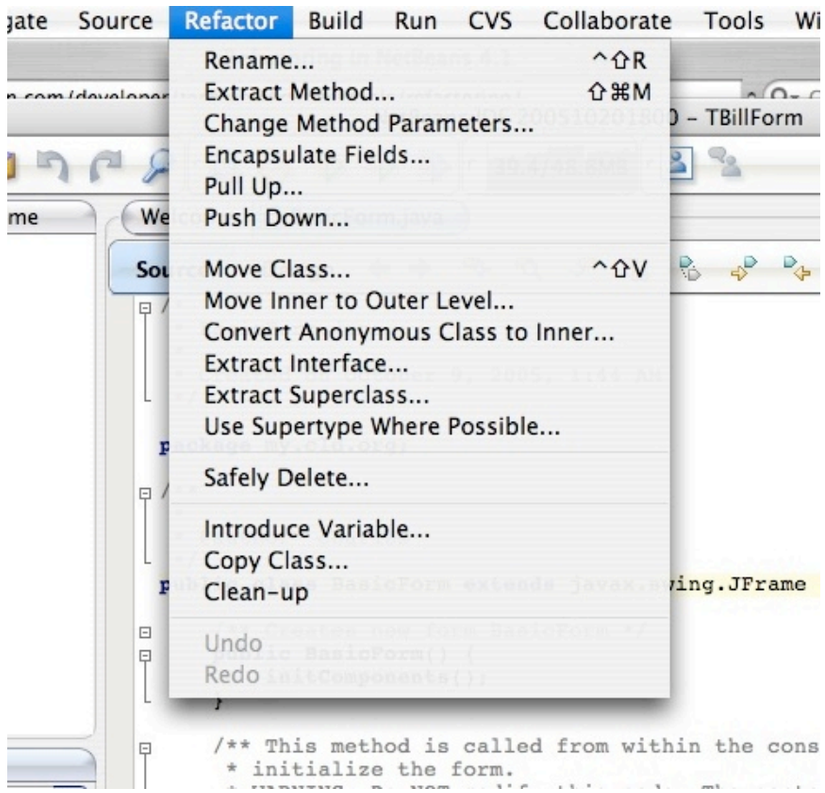




The Problem

- But there is hope. Automated refactorings include
  - Renaming classes
  - Generating get/set methods
  - Pulling up methods into a superclass
  - Generate an interface from a class

- What about larger refactorings?



Eclipse

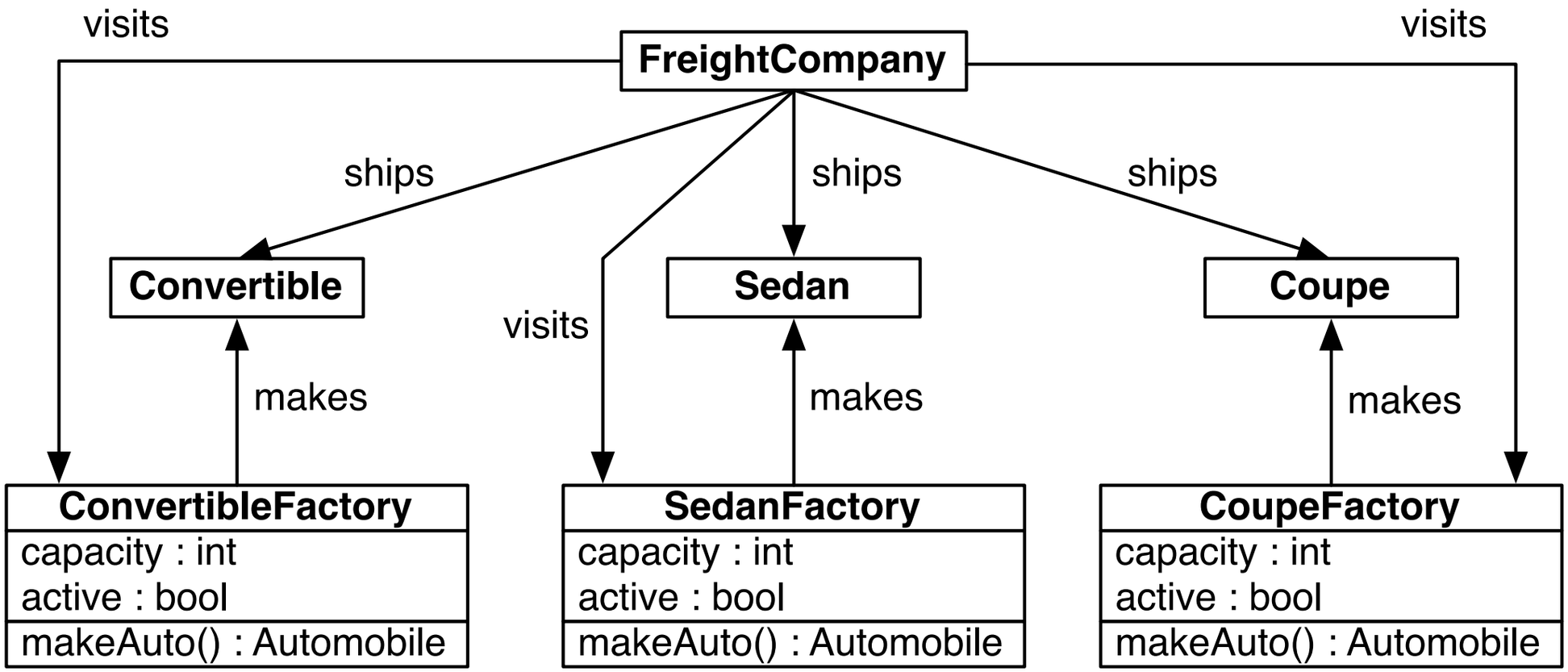
Design Patterns

A *design pattern* is a reusable solution to a common design problem that occurs in a particular context. [Gamma *et al.*]

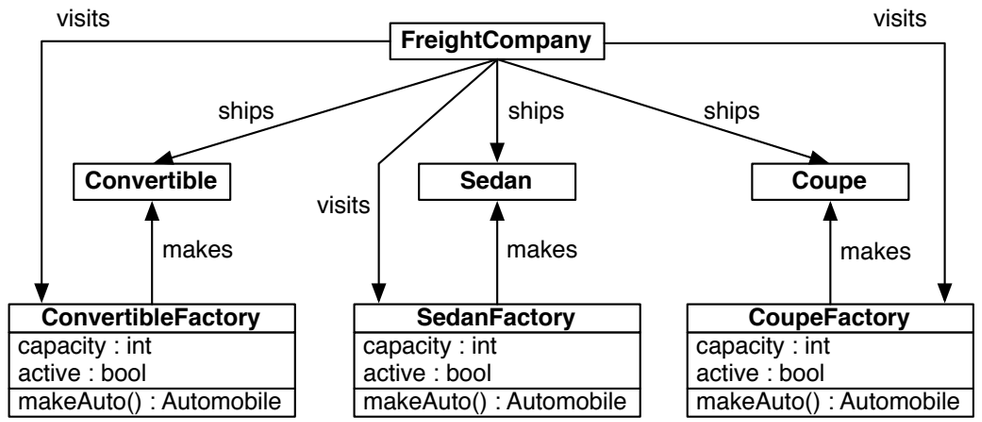
Benefits:

- Captures the experience of professional designers
- Enables designers to think at a higher level of abstraction
- Improves the maintainability and reusability of software designs

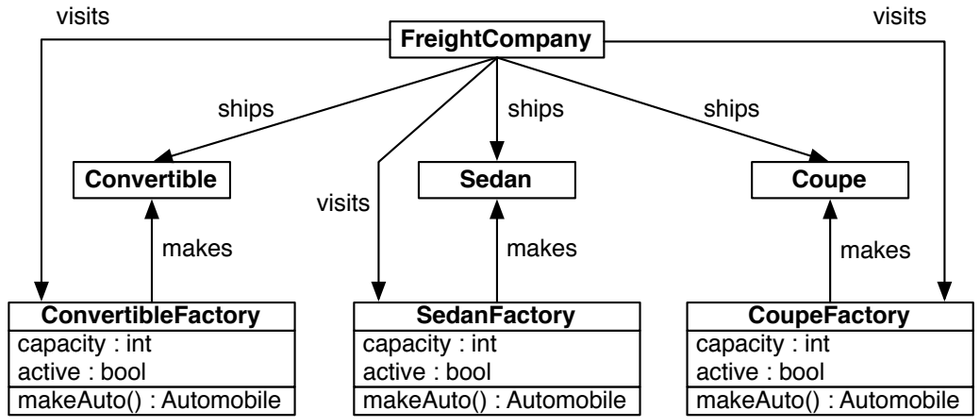
Example: Abstract Factory



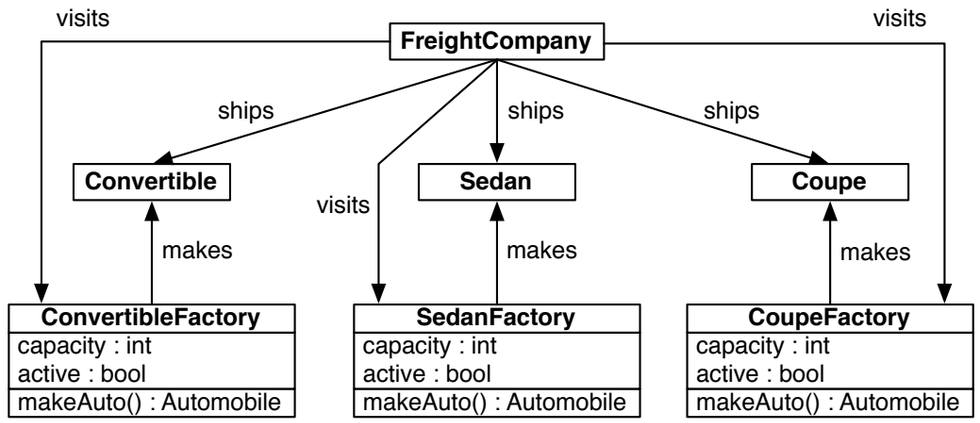
Example: Abstract Factory

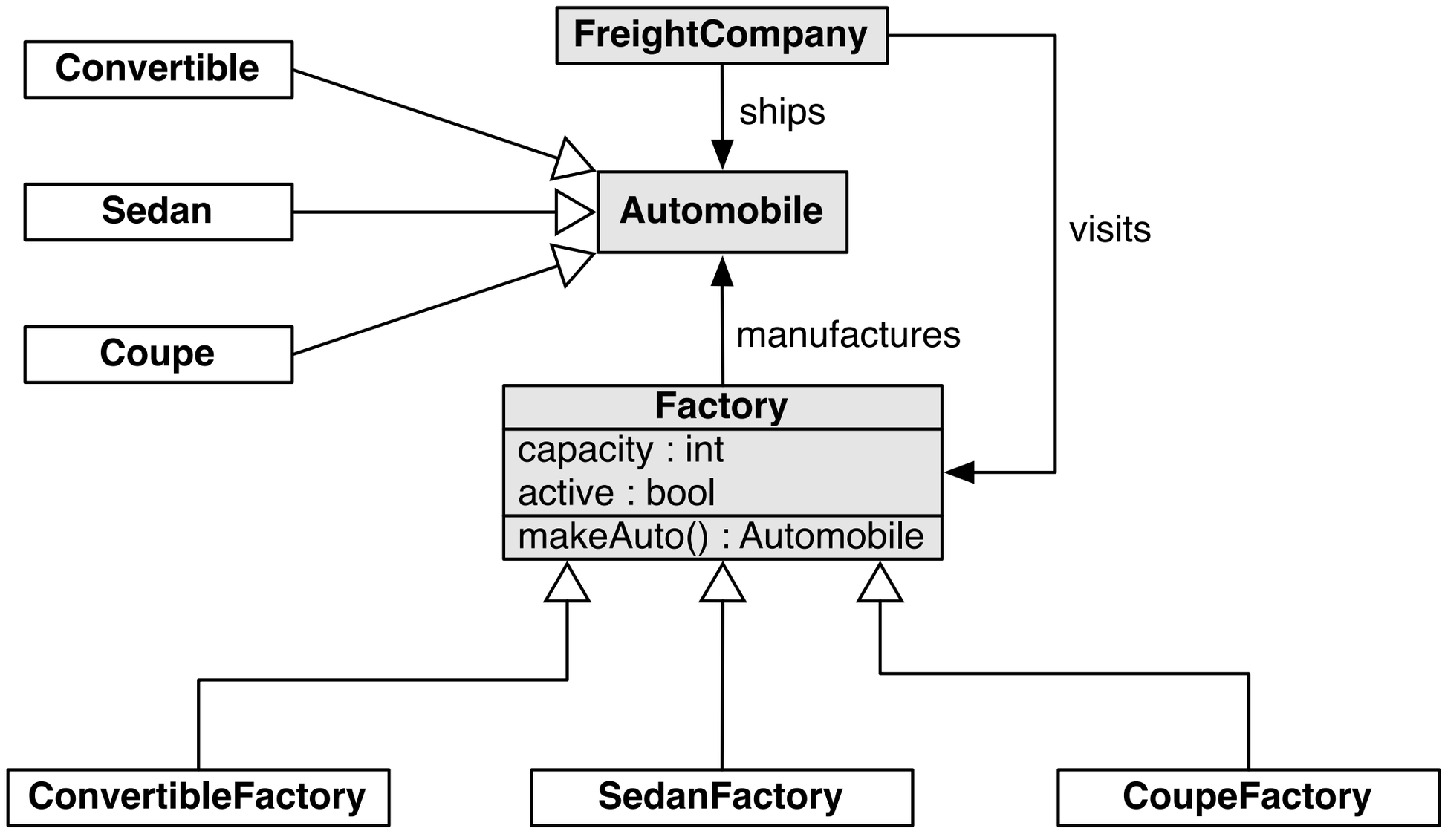


Example: Abstract Factory



Example: Abstract Factory





Thesis Statement

It is possible to use techniques from evolutionary computation, with guidance from software engineering metrics, in order to improve the design of existing software through the introduction of design pattern instances.

Contributions

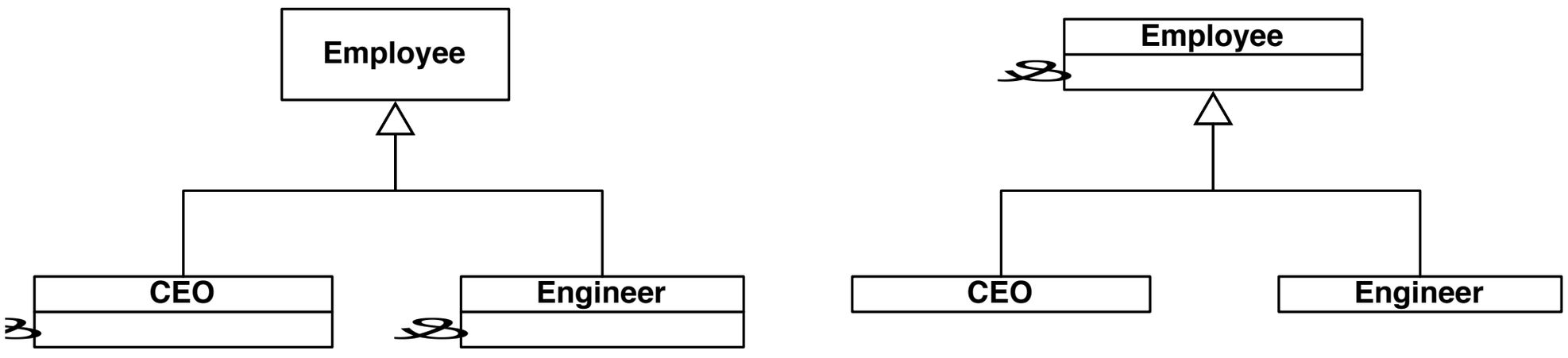
1. Novel GP encoding for refactoring software designs.

2. Process for automatically instantiating design patterns in existing software designs.

3. Step-by-step traceability for realizing a suggested design refactoring to enable manual or automated application.

Related Work

- [Seng *et al.*]: GA for optimizing a class hierarchy
- Key insight:
  - Moving operations between classes in a hierarchy can improve design quality
  - Evolution can efficiently explore sequences of refactorings to apply
  - Example: promoting a common operation to a superclass



Before

After

Related Work

- [O'Keeffe & Ó Cinnéide]: "search-based refactoring"
- Key insight:
  - Perhaps other (non-GA) evolutionary approaches are more suitable for search-based refactoring
- Study compared evolutionary approaches
  - Genetic Algorithm, Simulated Annealing, and Hill Climbers
  - Objective: to choose the optimal sequence of 14 refactoring steps, using software engineering metrics to evaluate solution quality
  - Conclusion: Multiple-restart hill-climber is most effective.

Existing approaches are effective for automating simple, incremental changes.

- No support for *composition* of multiple changes.
- They do not address larger refactoring strategies, such as design patterns.

Our Approach

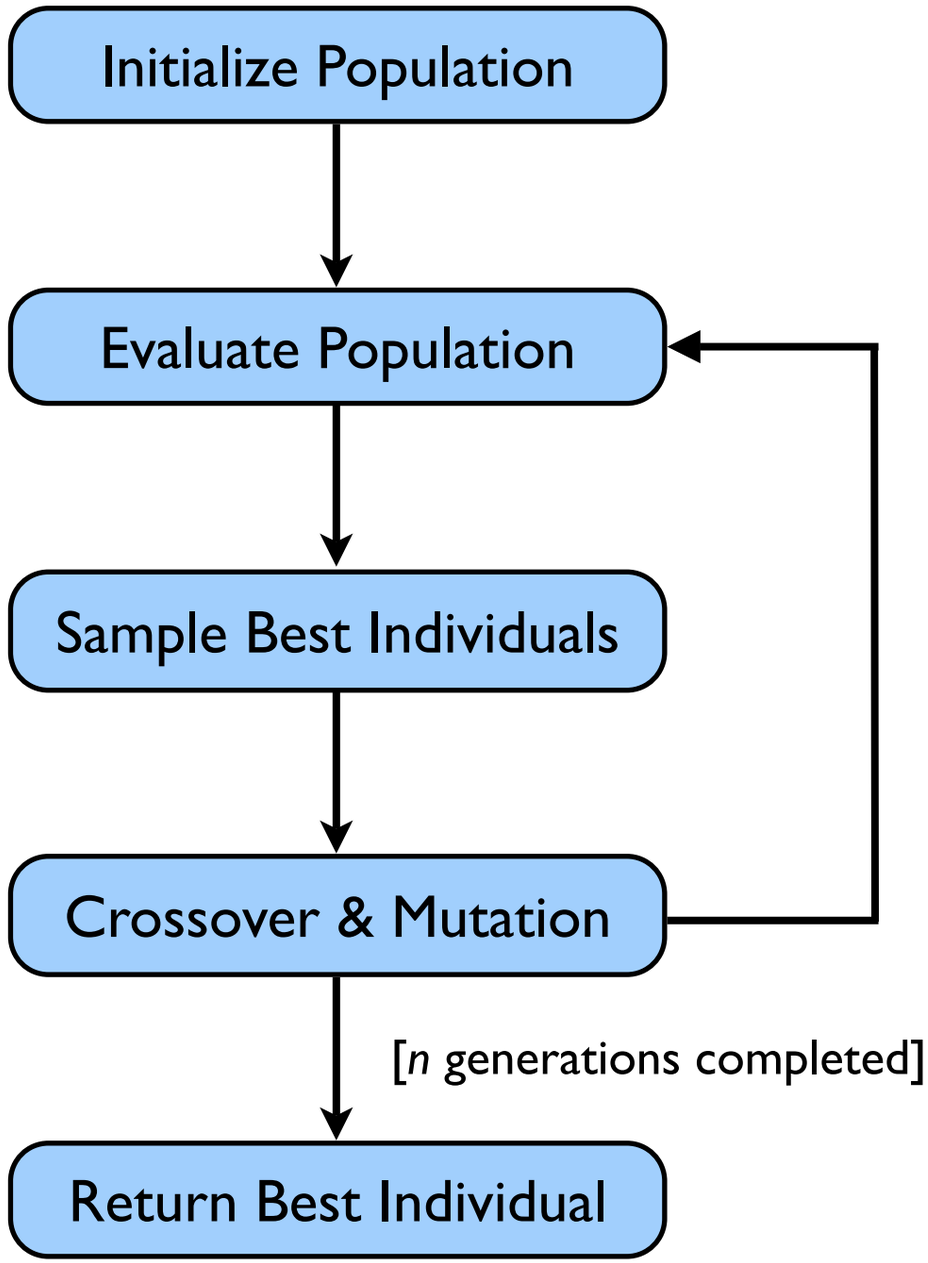
- Focus on the introduction of design patterns.
- Use genetic programming (GP), with a tree-based genotype, to leverage composition of mutation operators.
- Use metrics to evaluate fitness of modified designs.

Design Patterns

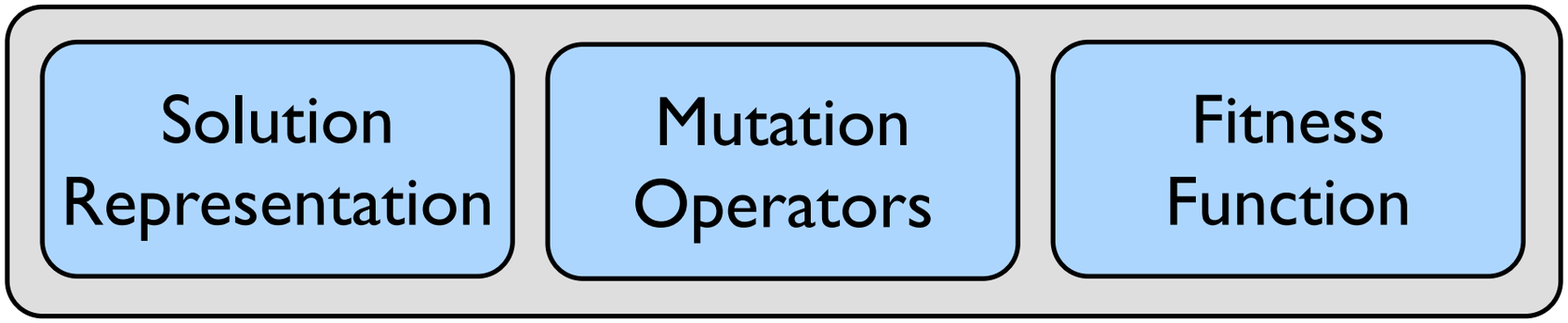
This approach supports seven Gamma design patterns:

- Abstract Factory
- Adapter
- Bridge
- Composite
- Decorator
- Prototype
- Proxy

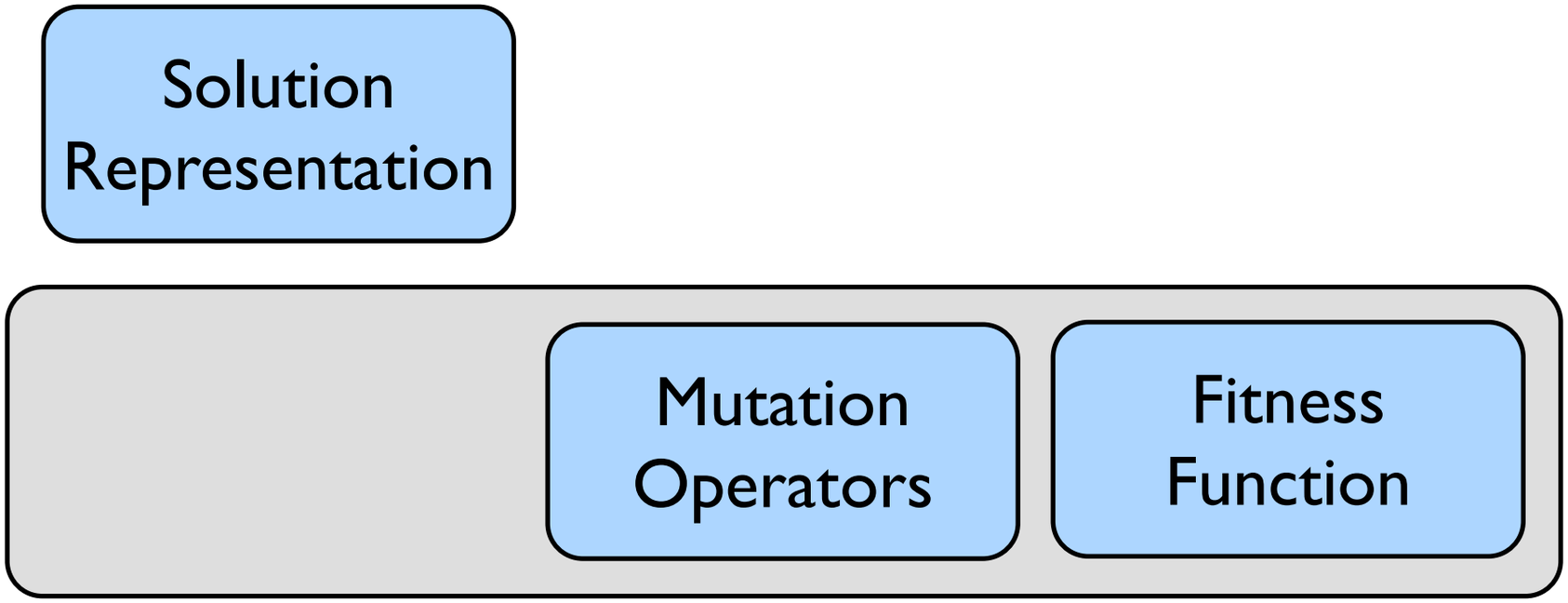
Evolutionary Approach



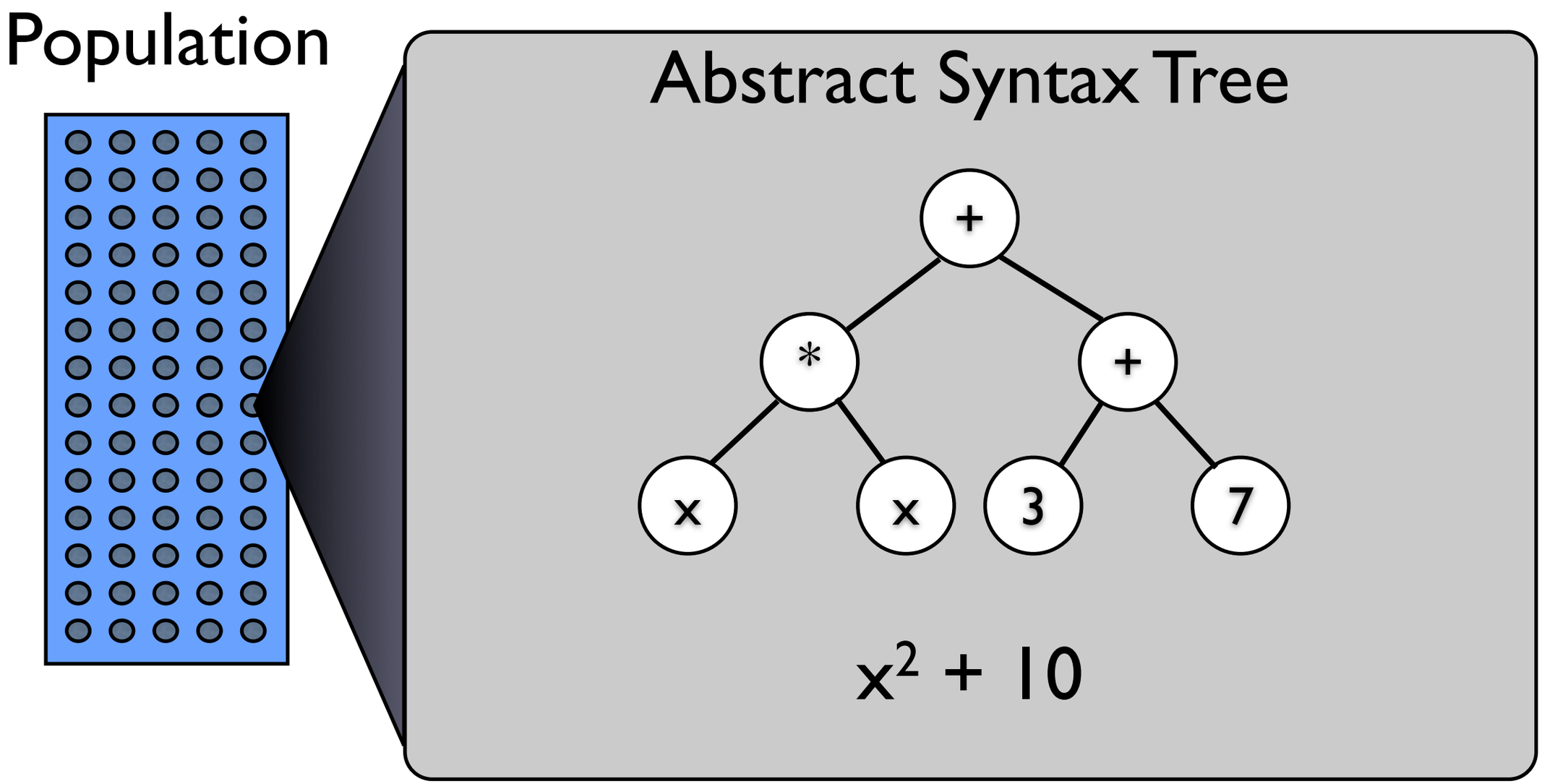
Evolutionary Approach



Evolutionary Approach

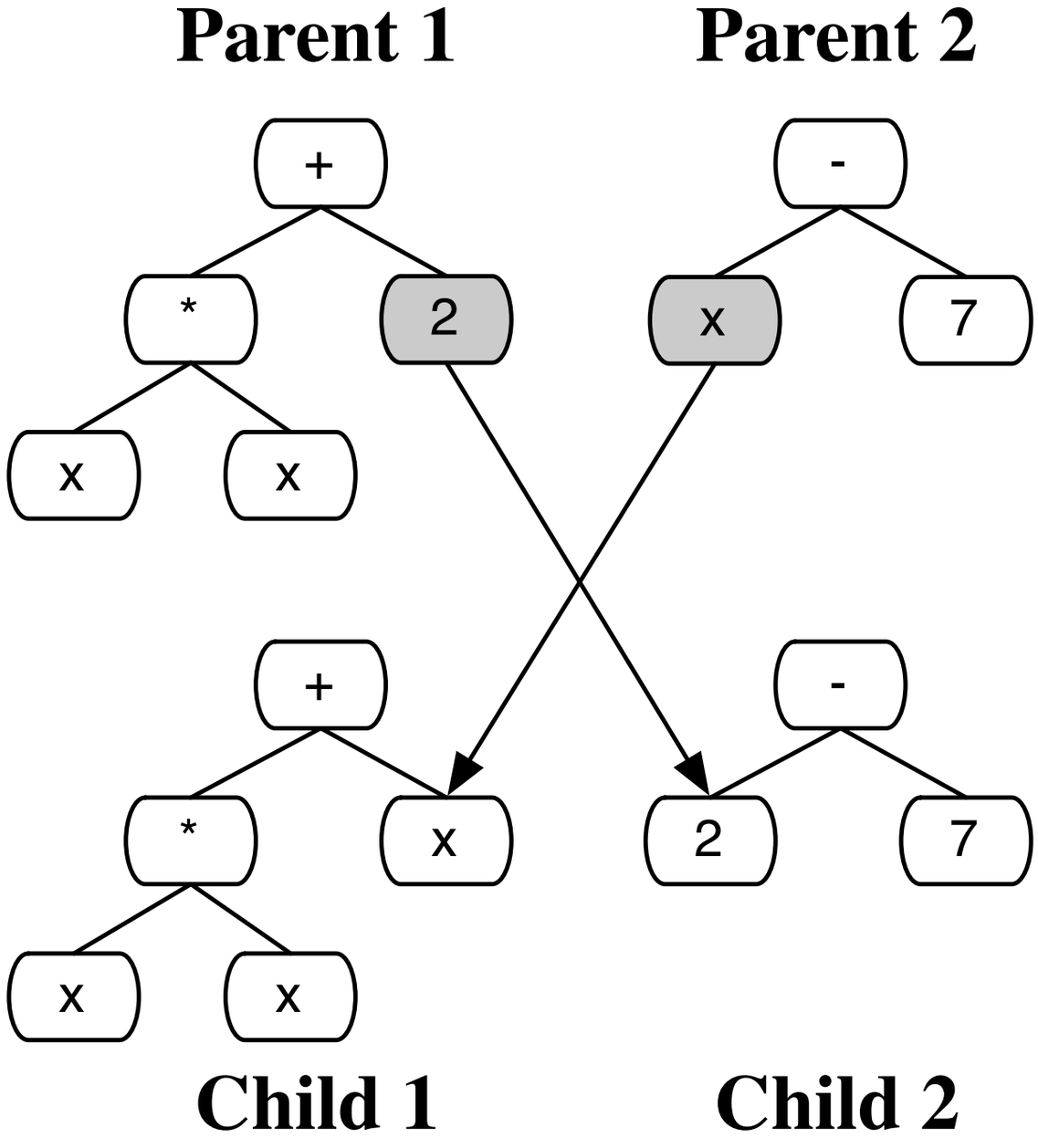


Traditional GP

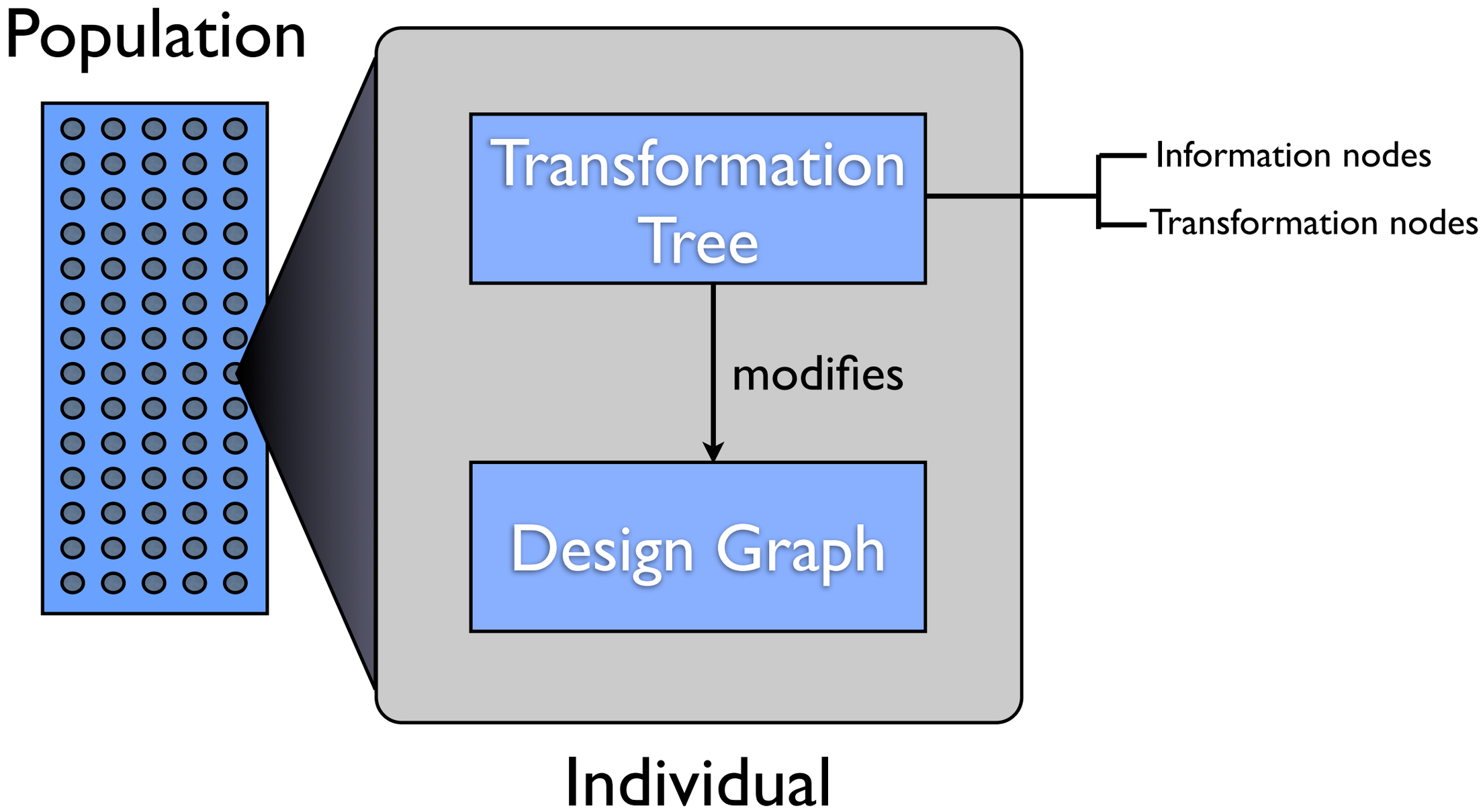


Individual

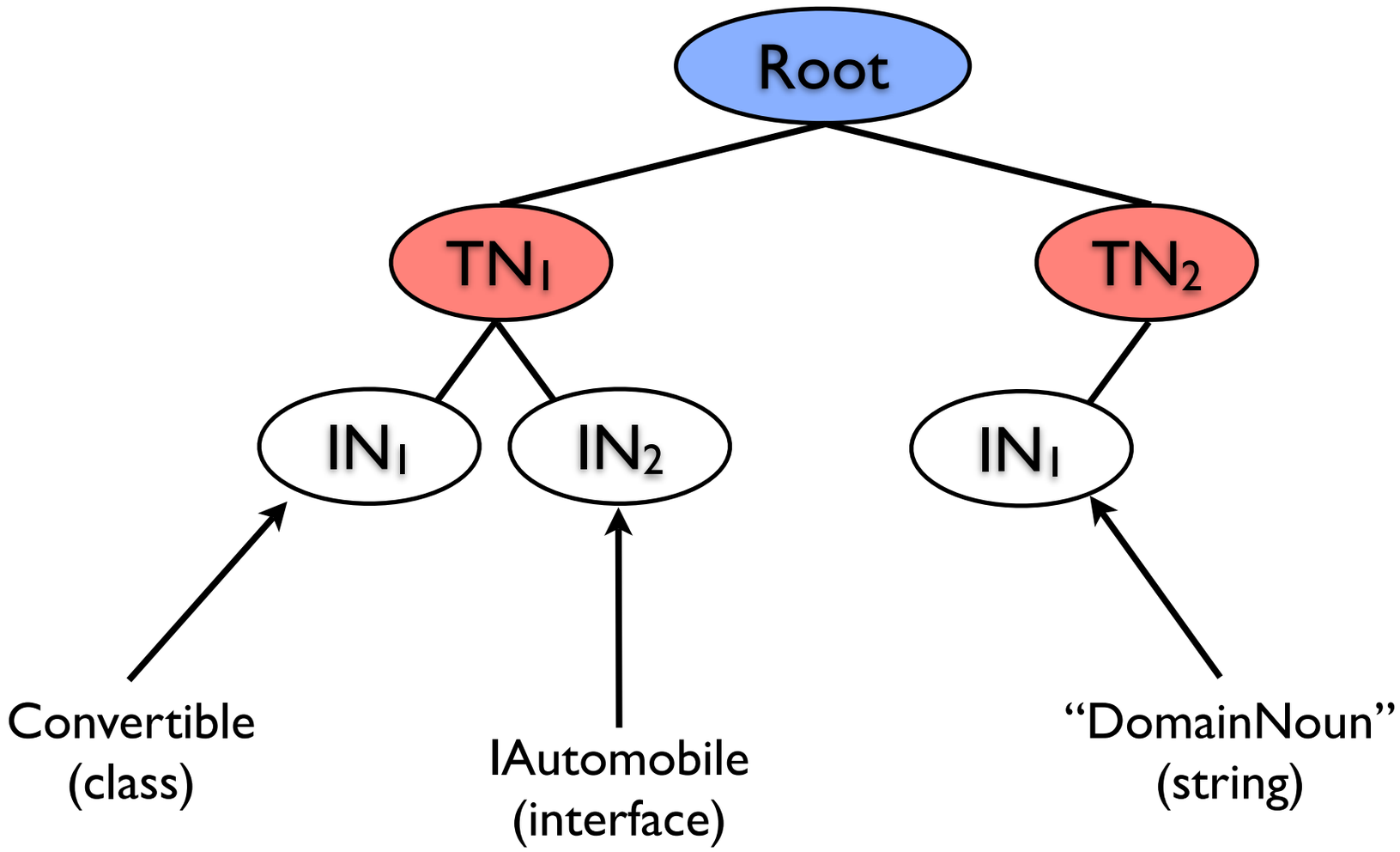
Subtree Crossover



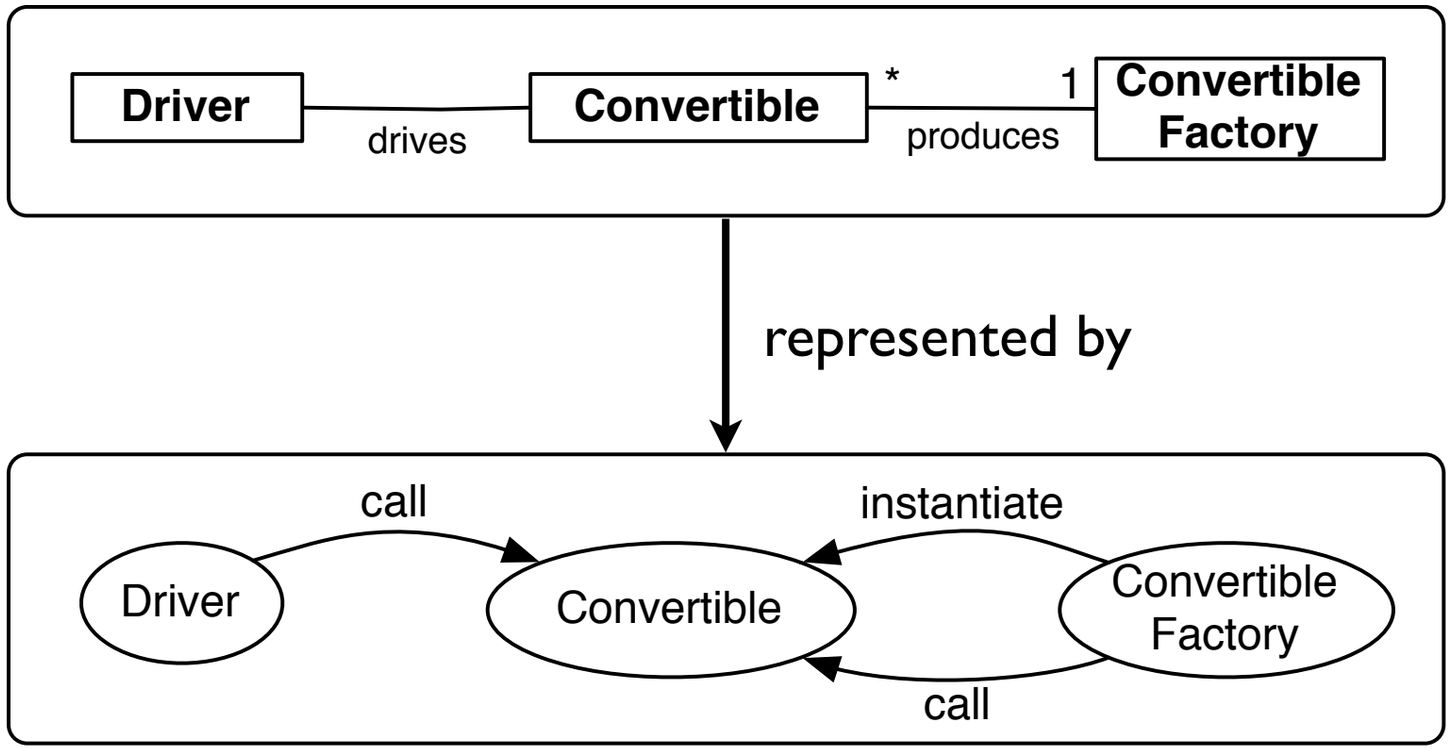
Extended GP


Population
Transformation Tree
Information nodes
Transformation nodes
modifies
Design Graph
Individual

Transformation Tree

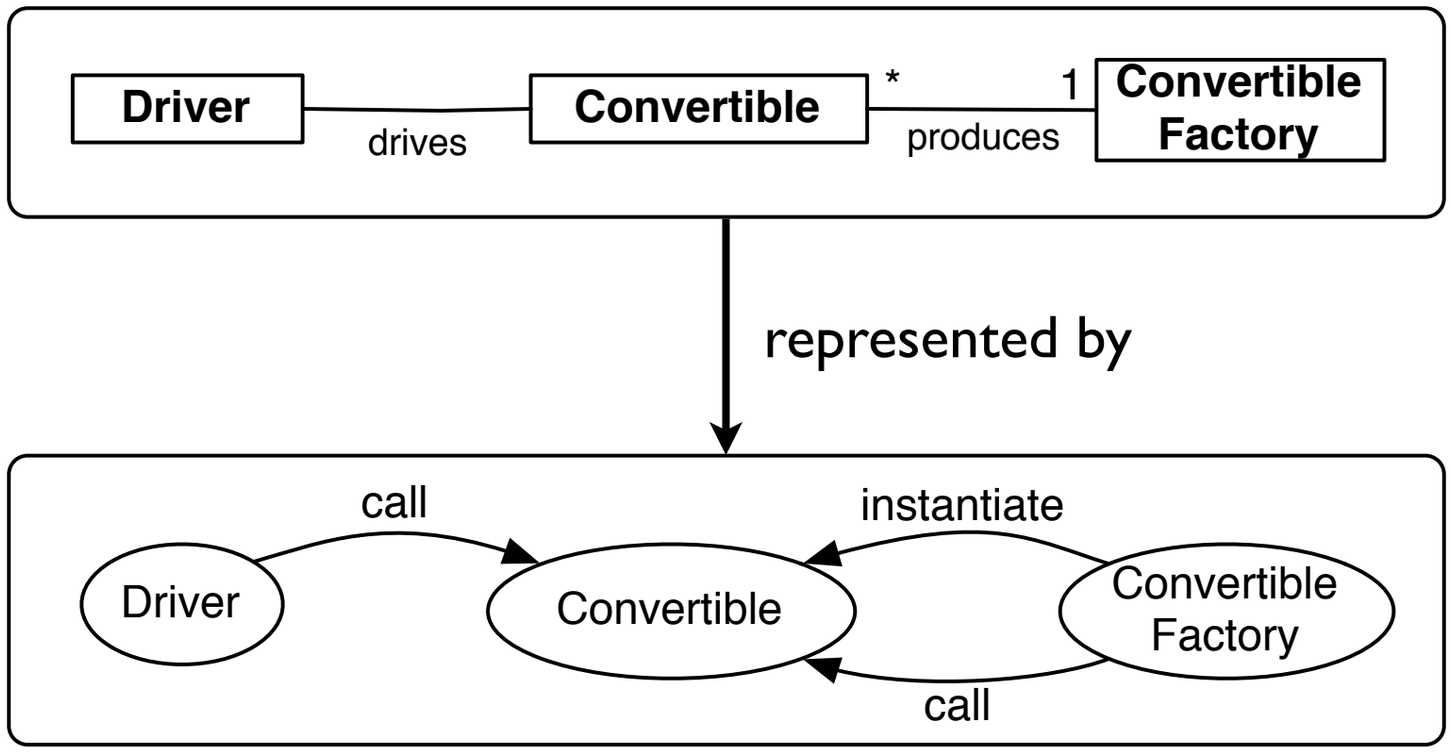


Software Design



Design Graph

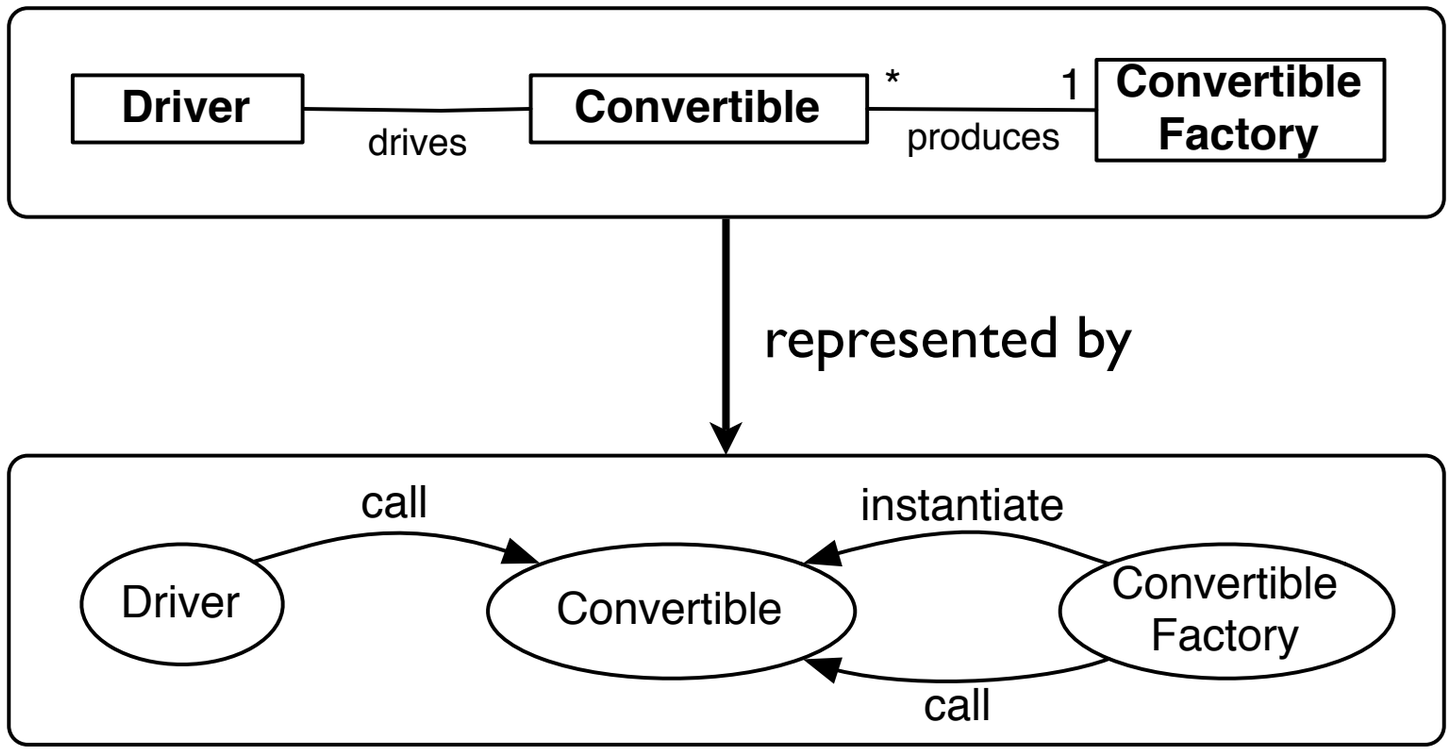
Software Design



Design Graph

Design Graph Elements:

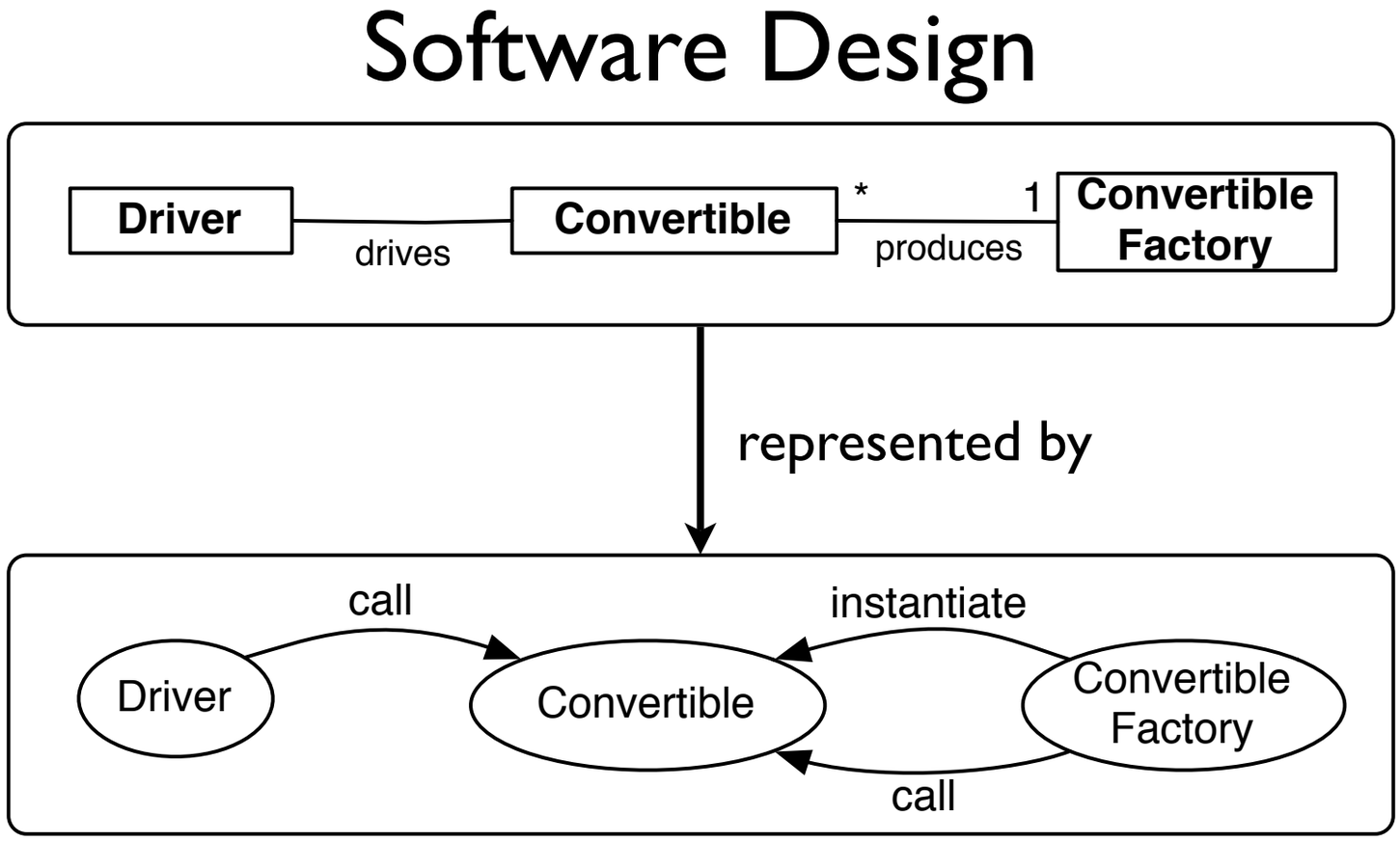
Software Design

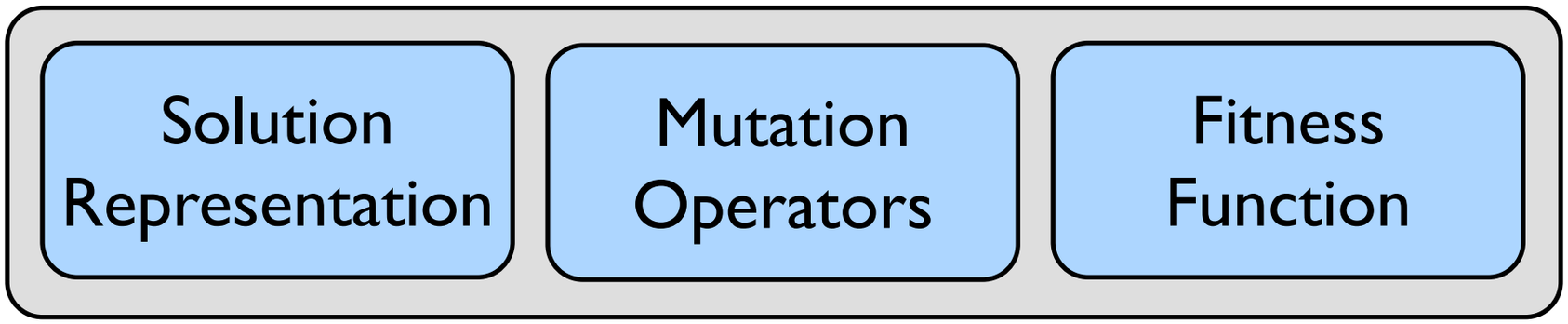


Design Graph

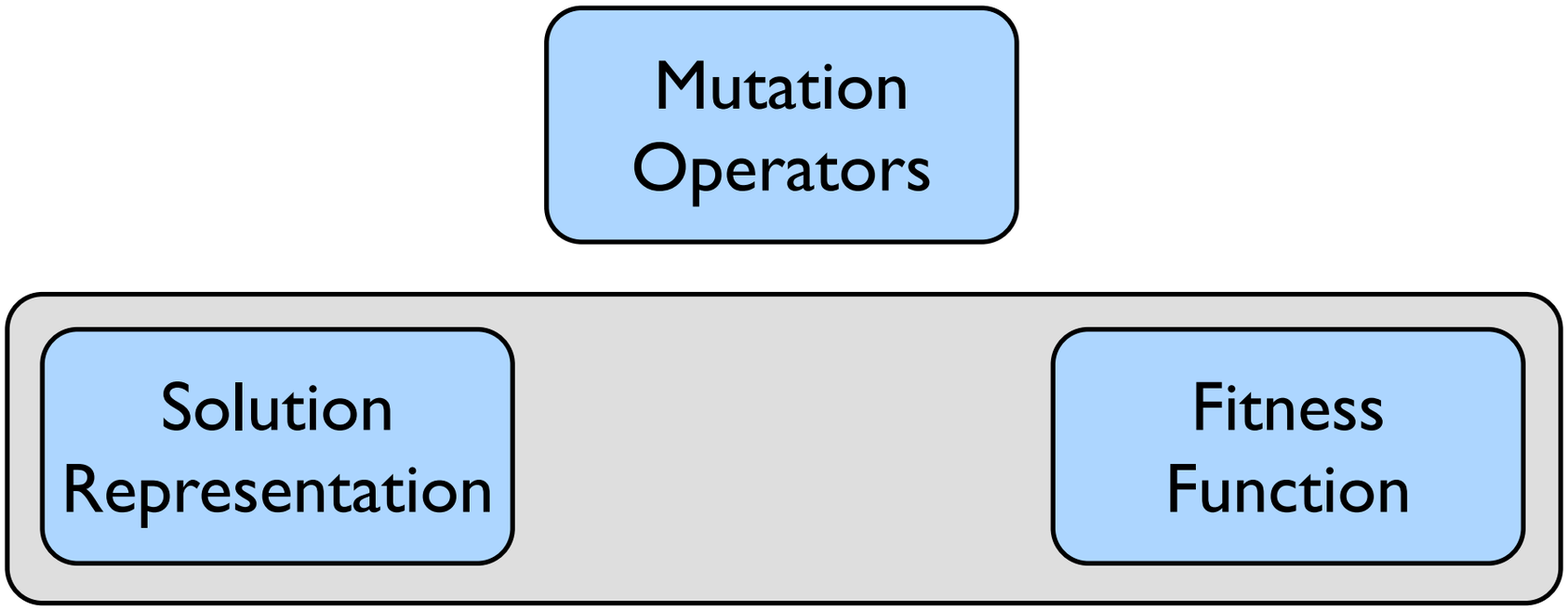
Design Graph Elements:

- Vertices:
  - Classes
  - Interfaces
  - Operations
- Edges
  - Aggregations
  - Associations
  - Function calls
  - Inheritance
  - Instantiations



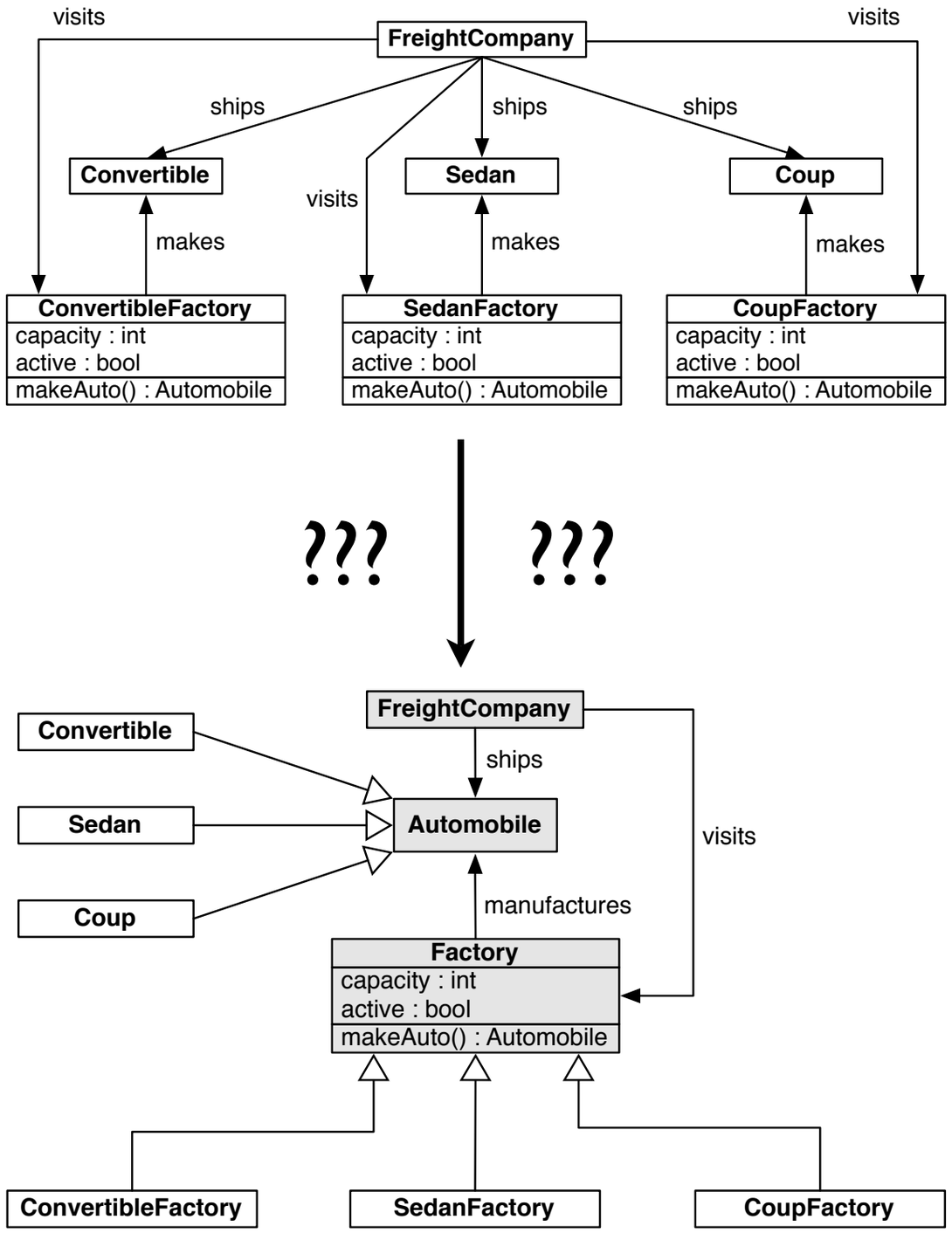


Evolutionary Approach

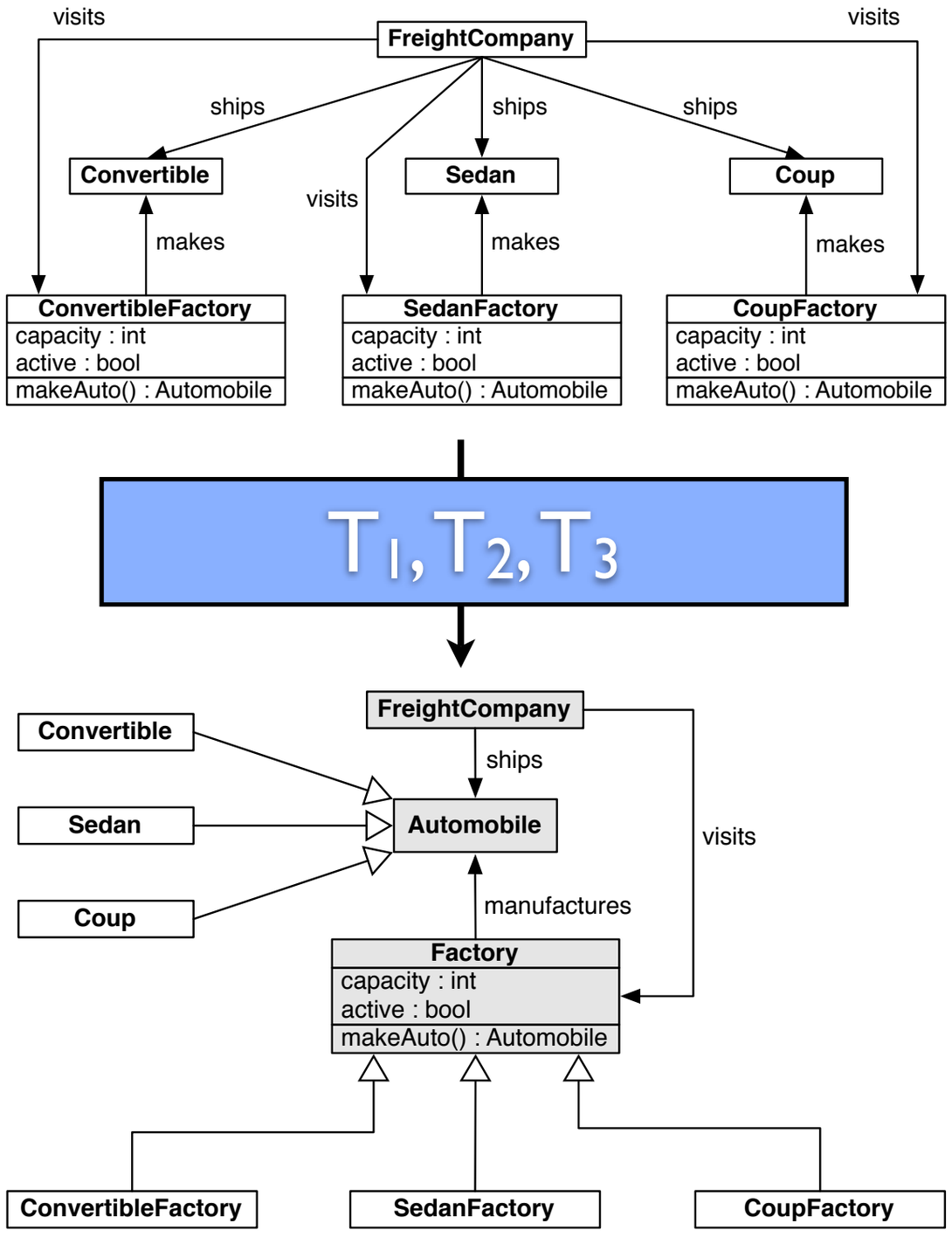


Evolutionary Approach

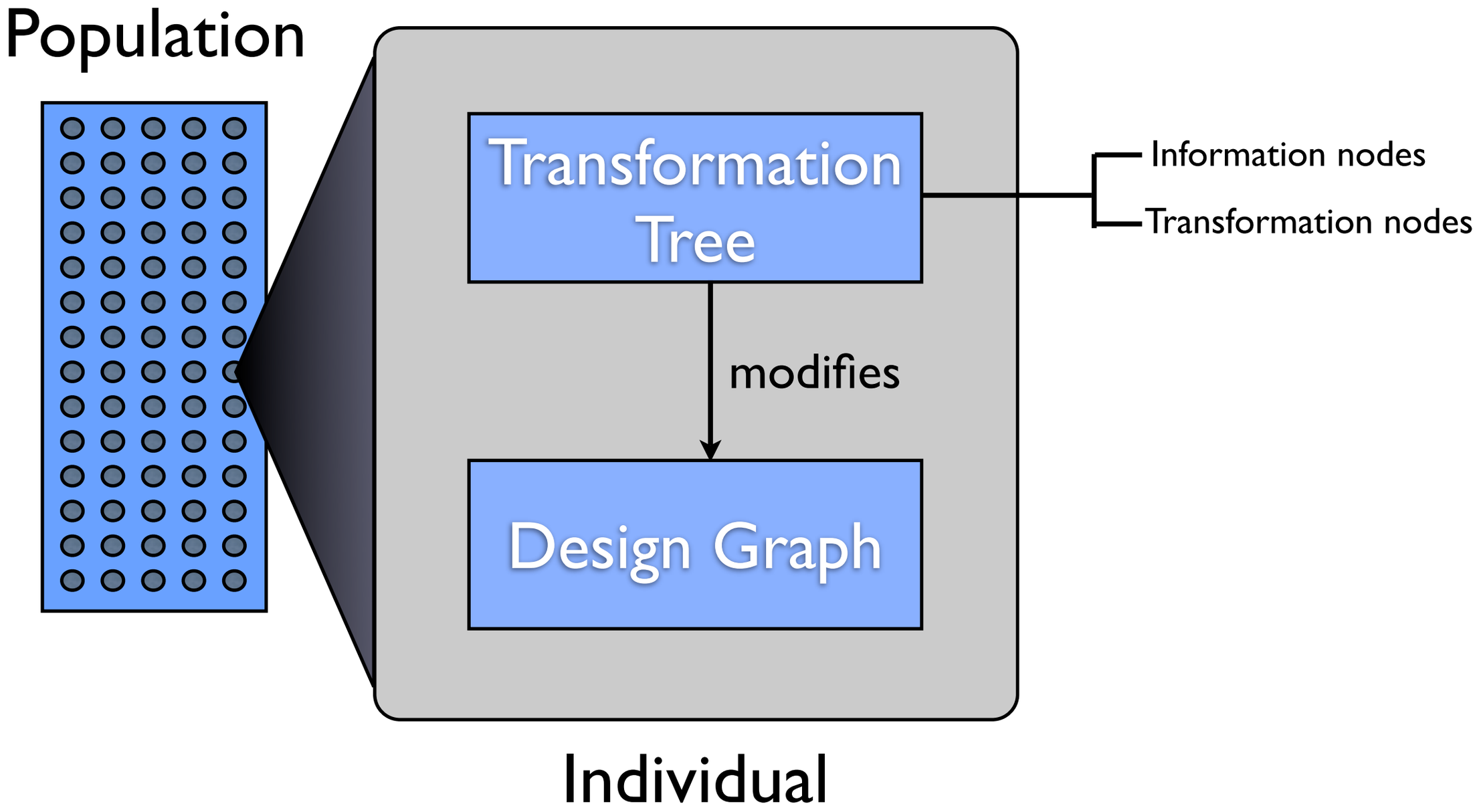
Course-Grained Refactorings



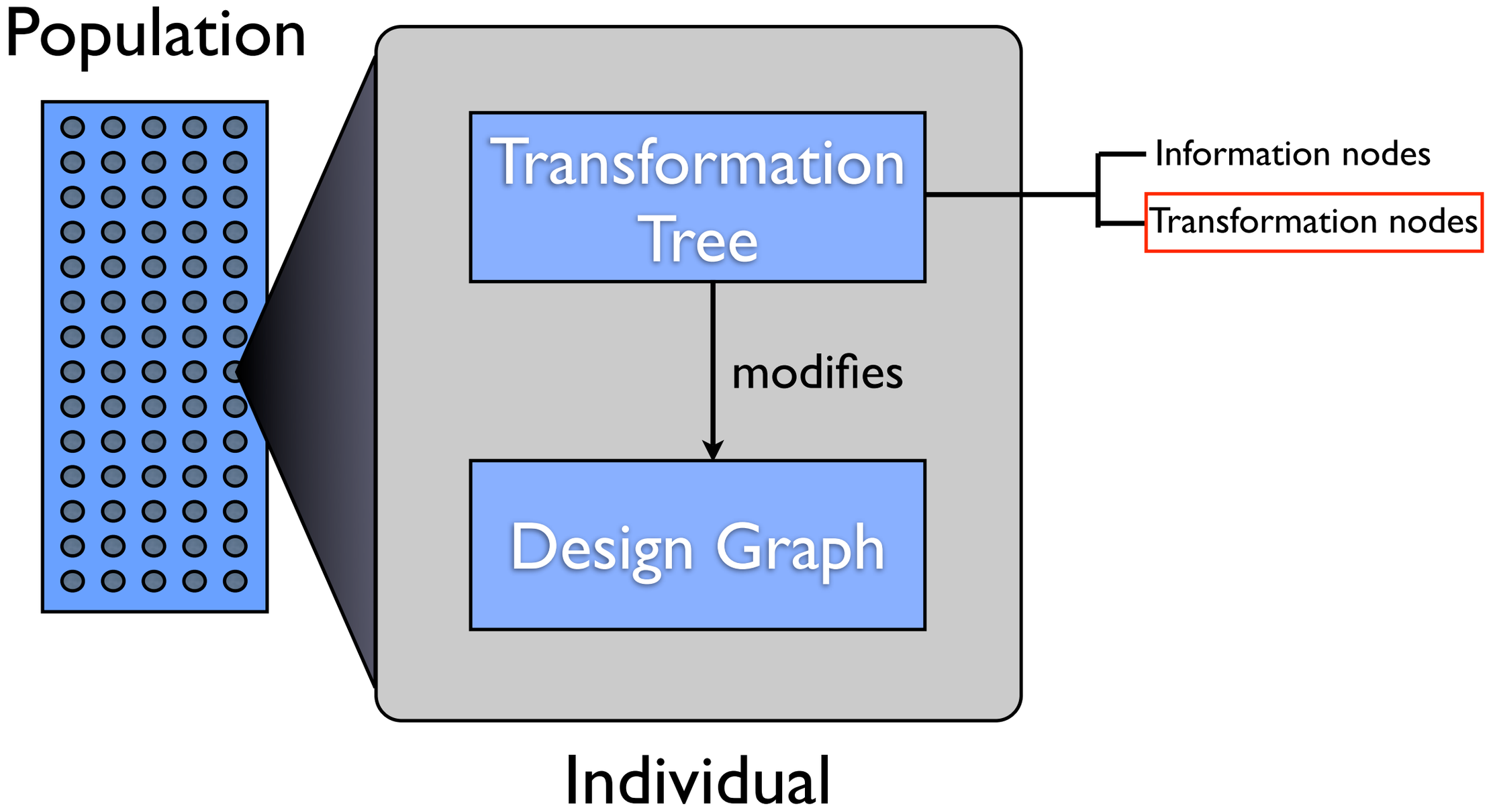
Course-Grained Refactorings



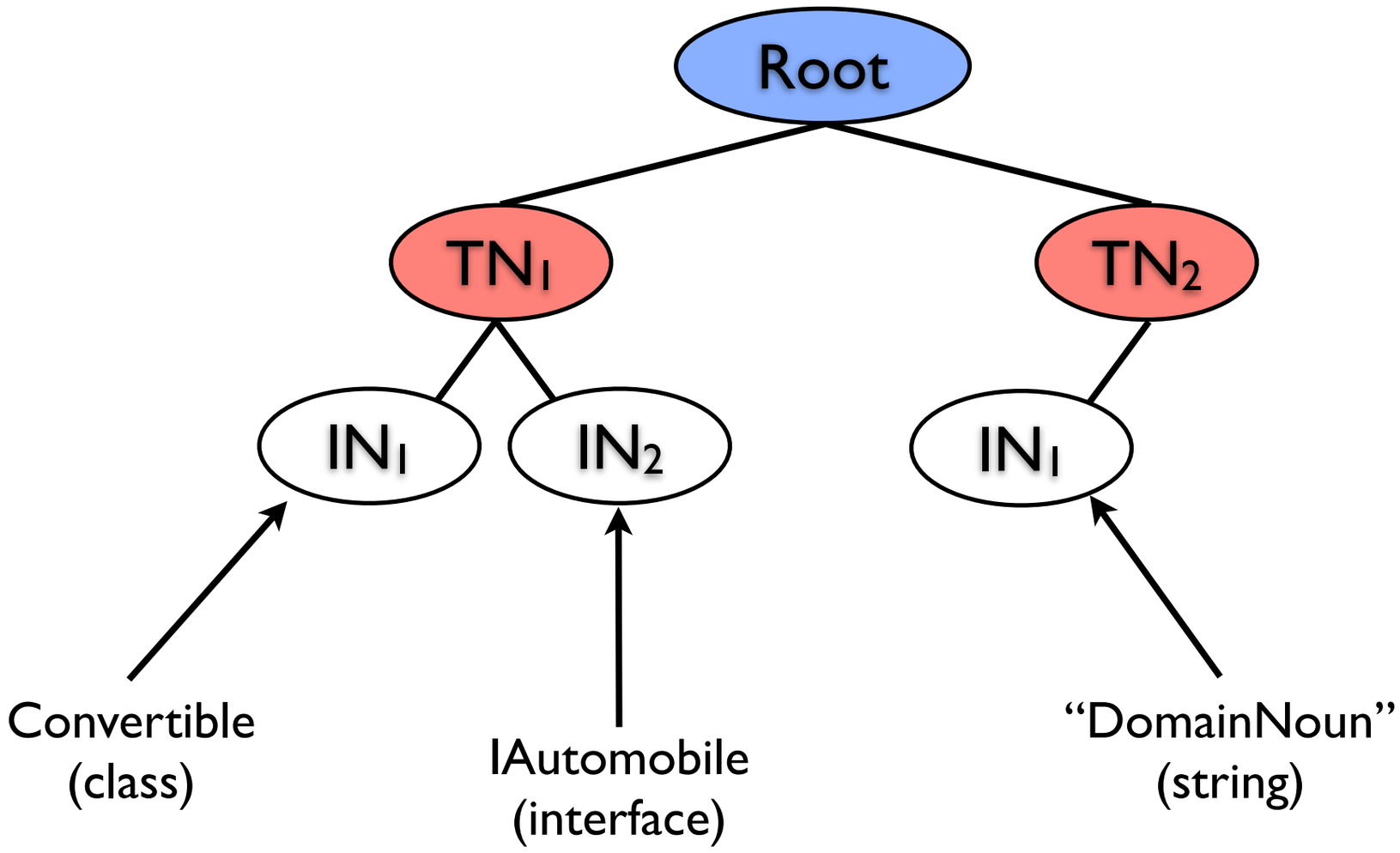
Extended GP


Population
Transformation Tree
Information nodes
Transformation nodes
modifies
Design Graph
Individual

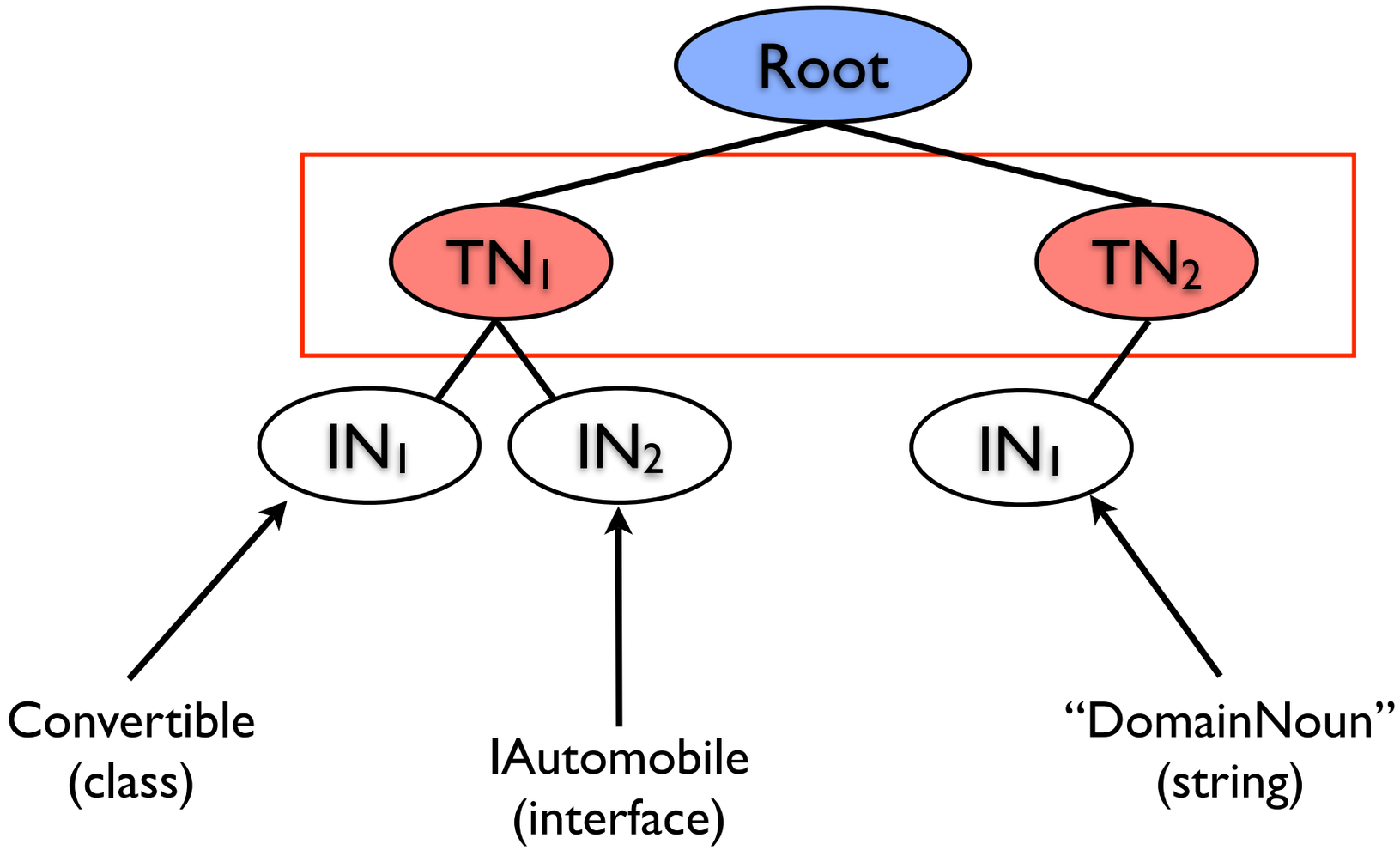
Extended GP


Population
Transformation Tree
Information nodes
Transformation nodes
modifies
Design Graph
Individual

Transformation Tree



Transformation Tree



Minitransformations

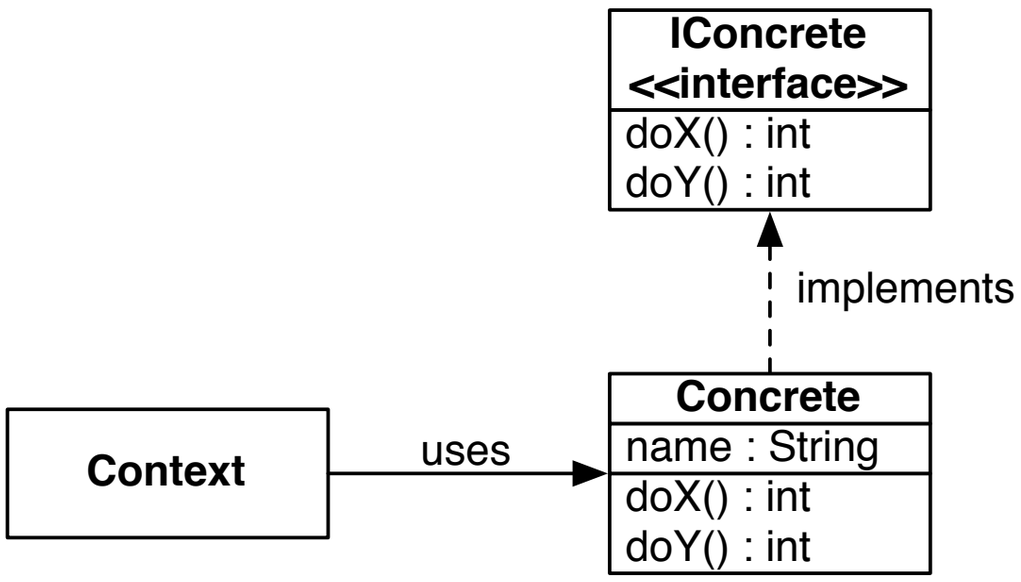
- Six mini-patterns, or *minitransformations:*

1. *Abstract Access*
2. *Abstraction*
3. *Delegation*
4. *Encapsulate Construction*
5. *Partial Abstraction*
6. *Wrapper*

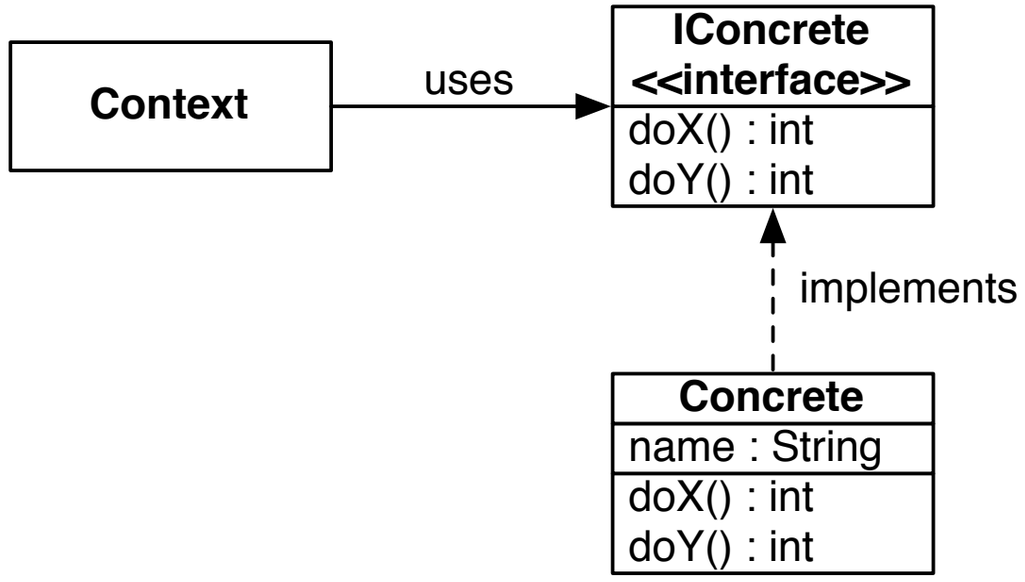
Abstract Access Transformation

Purpose: to modify a class *Context* to access another class *Concrete* more abstractly through an interface *IConcrete*.

Before



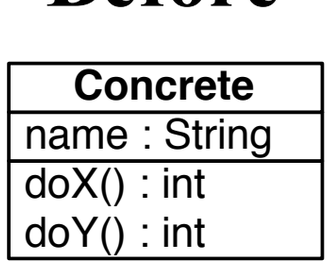
After



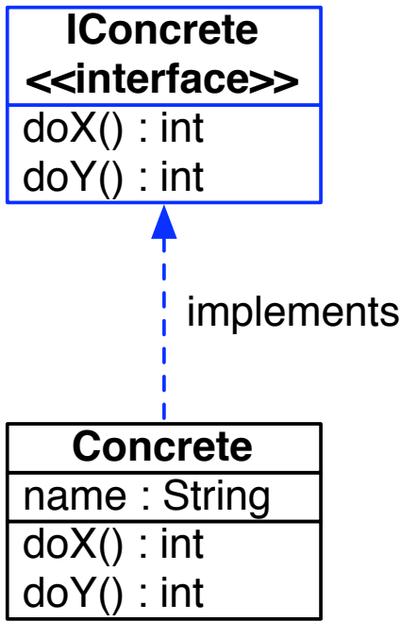
Abstraction Transformation

Purpose: to derive an interface from an existing class *Concrete*, enabling other classes to view *Concrete* more abstractly.

Before



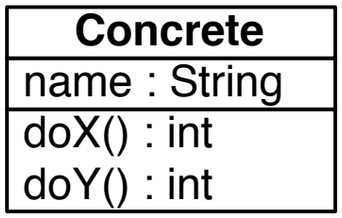
After



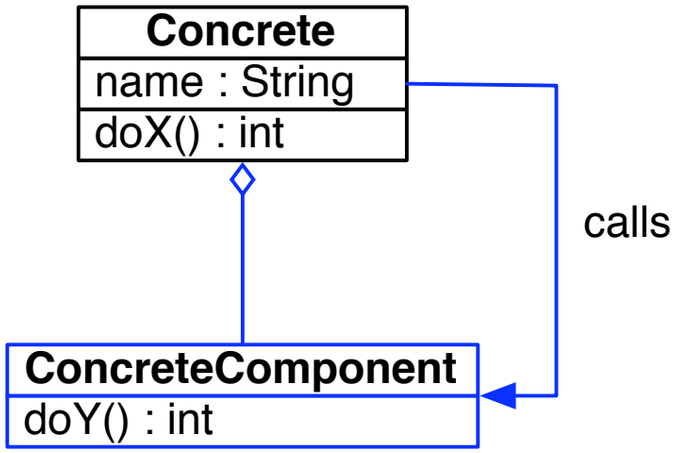
Delegation Transformation

Purpose: to delegate a subset of the functionality of one class to another class.

Before



After



Encapsulate Construction Transformation

Purpose: to localize class instantiations into a dedicated operation.

Before

ShapeManager
numShapes : int
CopyShape() : Shape RandomizeShape() : Shape TransformShape() : Shape

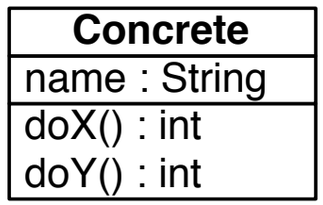
After

ShapeManager
numShapes : int
CopyShape() : Shape RandomizeShape() : Shape TransformShape() : Shape CreateShape() : Shape

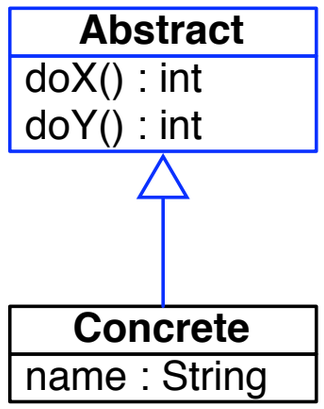
Partial Abstraction Transformation

Purpose: to create an abstract class with the same interface as an existing class.

Before



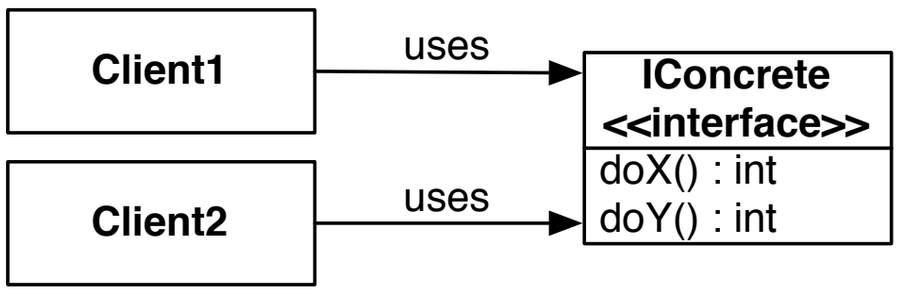
After



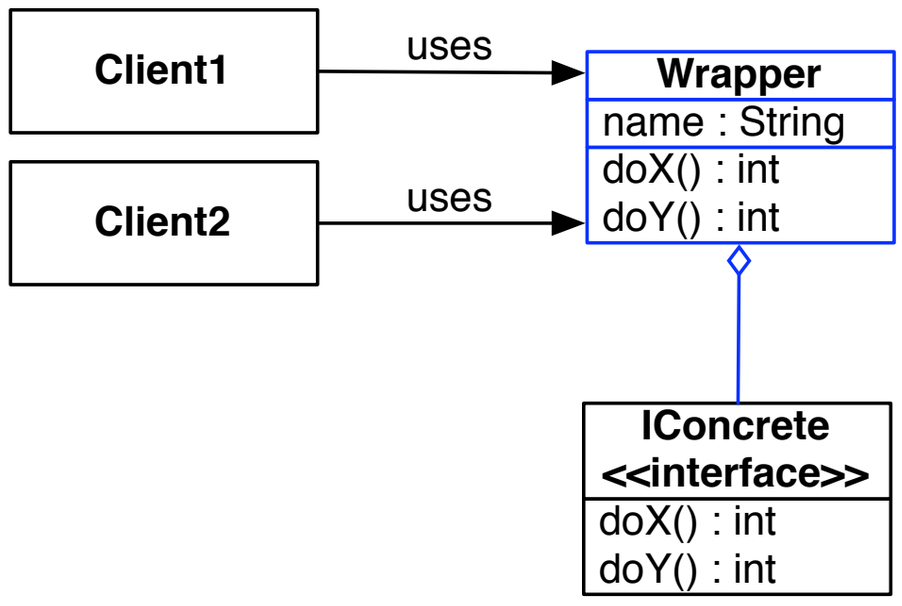
Wrapper Transformation

Purpose: to wrap a class with another, thus enabling run-time replacement of the wrapped class.

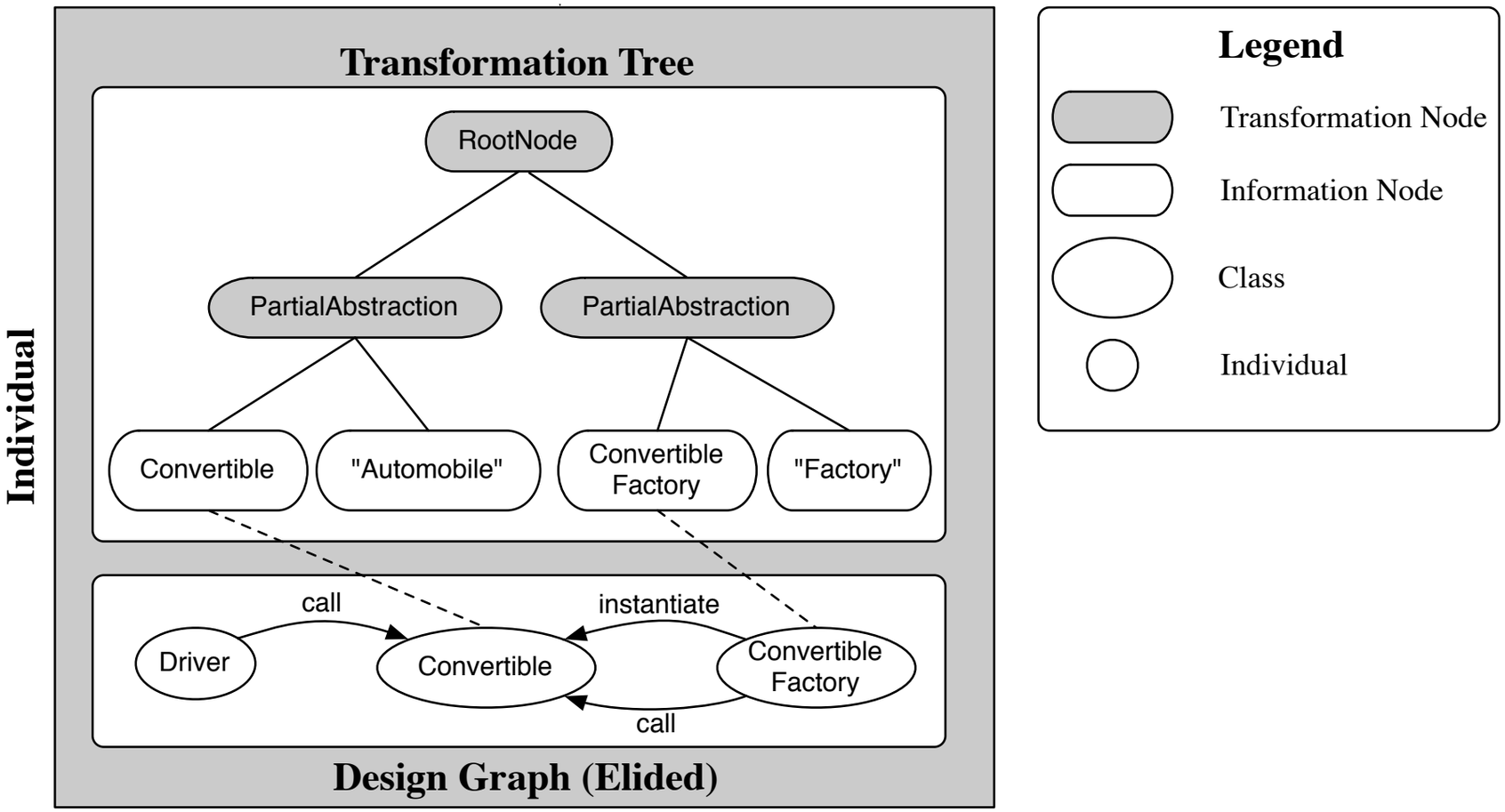
Before

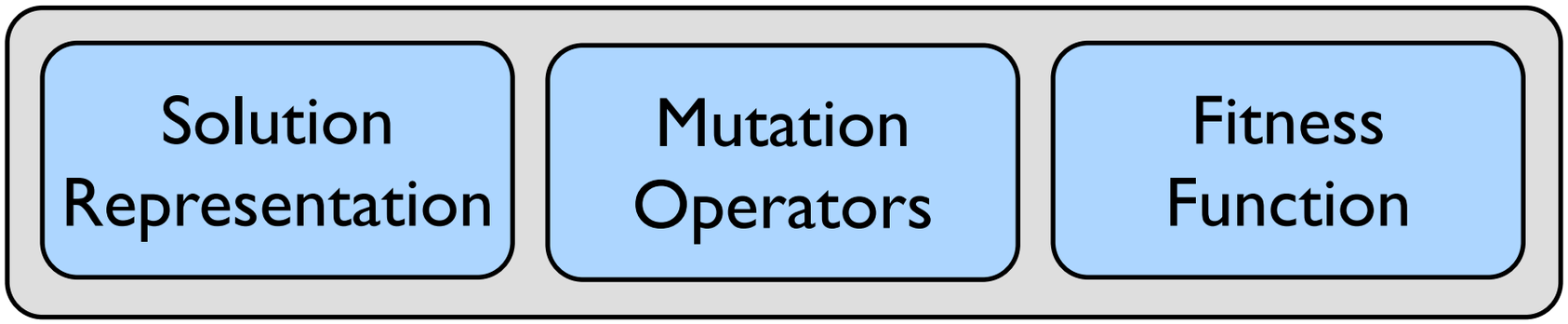


After

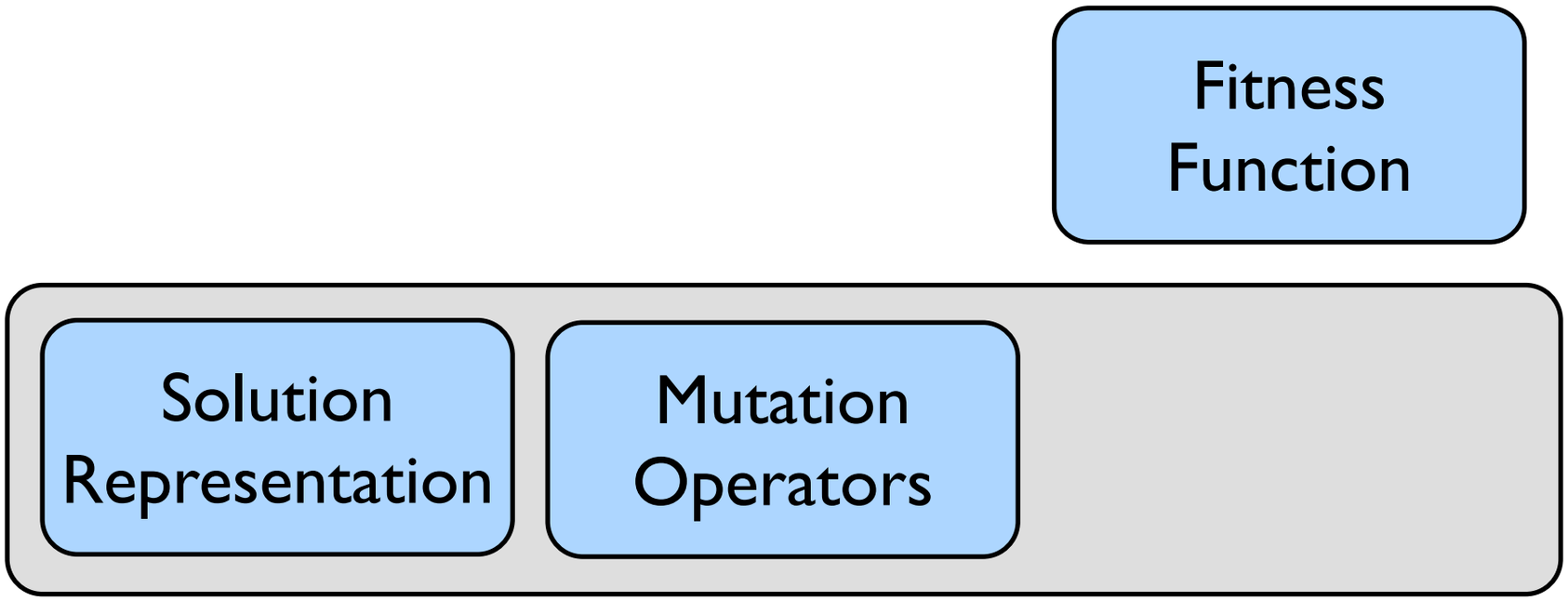


Minitransformations in Context





Evolutionary Approach



Evolutionary Approach

Fitness Function

Two initial terms:

1. Quality of evolved software design

    - Metrics

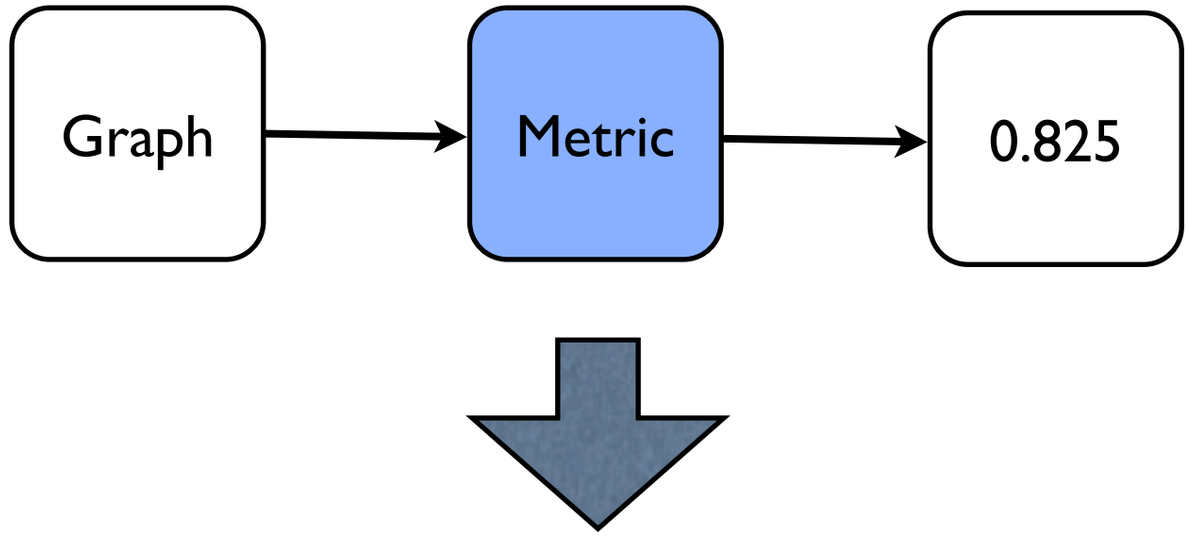
2. Presence of design patterns

    - Design pattern detector

Metrics

- Hierarchical Metrics for Object-Oriented Design Quality [Bansiya & Davis]
  - Known as "QMOOD"
  - Comprises 11 metrics
  - Evaluates multiple characteristics: cohesion, coupling, design size, etc.
  - Amenable to automation
  - Implemented naturally as a set of graph algorithms

Metrics



Coupling = $0.5*M_1 + 0.25*M_2 - 0.5*M_3$

Overall = 0.15*Cohesion - 0.15*Coupling + ...

Design Pattern Detection

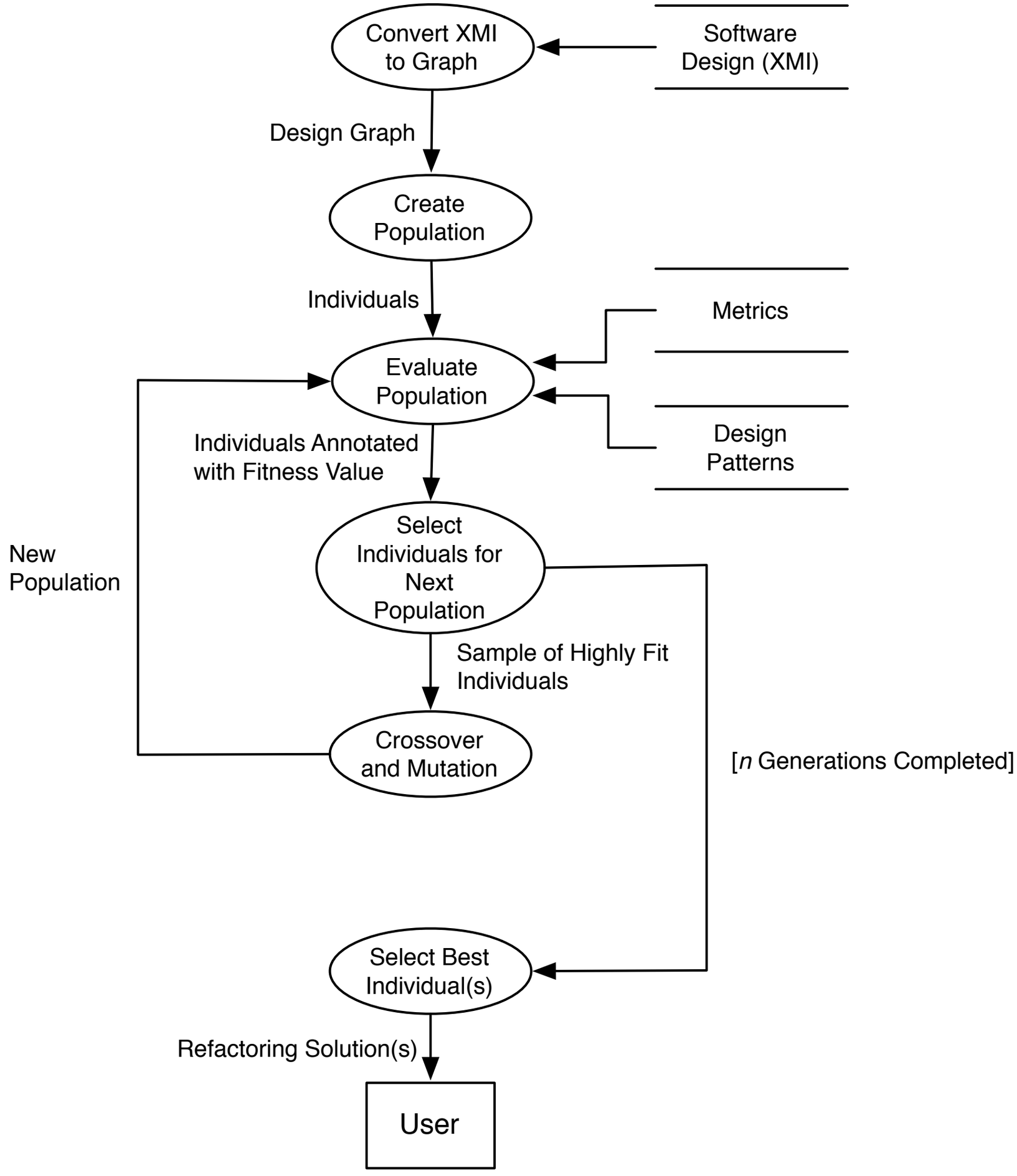
- Design patterns have a characteristic *signature*
  - Can be detected using a logic or query language (e.g., Prolog or SQL)
  - Example Prolog query for the Abstract Factory pattern:

```
abstract_factory(AFact,CFact,AProd,CProd,Client) :-
  cls(AFact), cls(CFact), cls(AProd), cls(CProd),
  cls(Client), inherit(CFact,AFact),
  inherit(CProd,AProd), instantiate(CFact,CProd),
  fcall(Client,CProd), AFact \= CFact,
  AProd \= CProd, AFact \= Client,
  CFact \= Client, AProd \= Client,
  CProd \= Client, AFact \= AProd,
  AProd \= CFact.
```

Fitness Function

F= Metrics -
C_1*nodeCountPenalty +
C_2*patternReward +
C_3*matchingSequencesReward

Data Flow Diagram



Implementation

- Input model format: ArgoUML XMI

- GP Framework: Evolutionary Computation for Java (ECJ)

- Graph support: jGraph

- Detection of Design Patterns: jLog and HSQLDB

Validation

5 experiments

- 4 strategies for parameter tuning
- 1 large case study

Parameter Tuning

1. Plain vanilla
2. Constraining the number of transformation nodes
3. Rewarding design pattern *presence*
4. Rewarding known sequences of transformation nodes

Setup

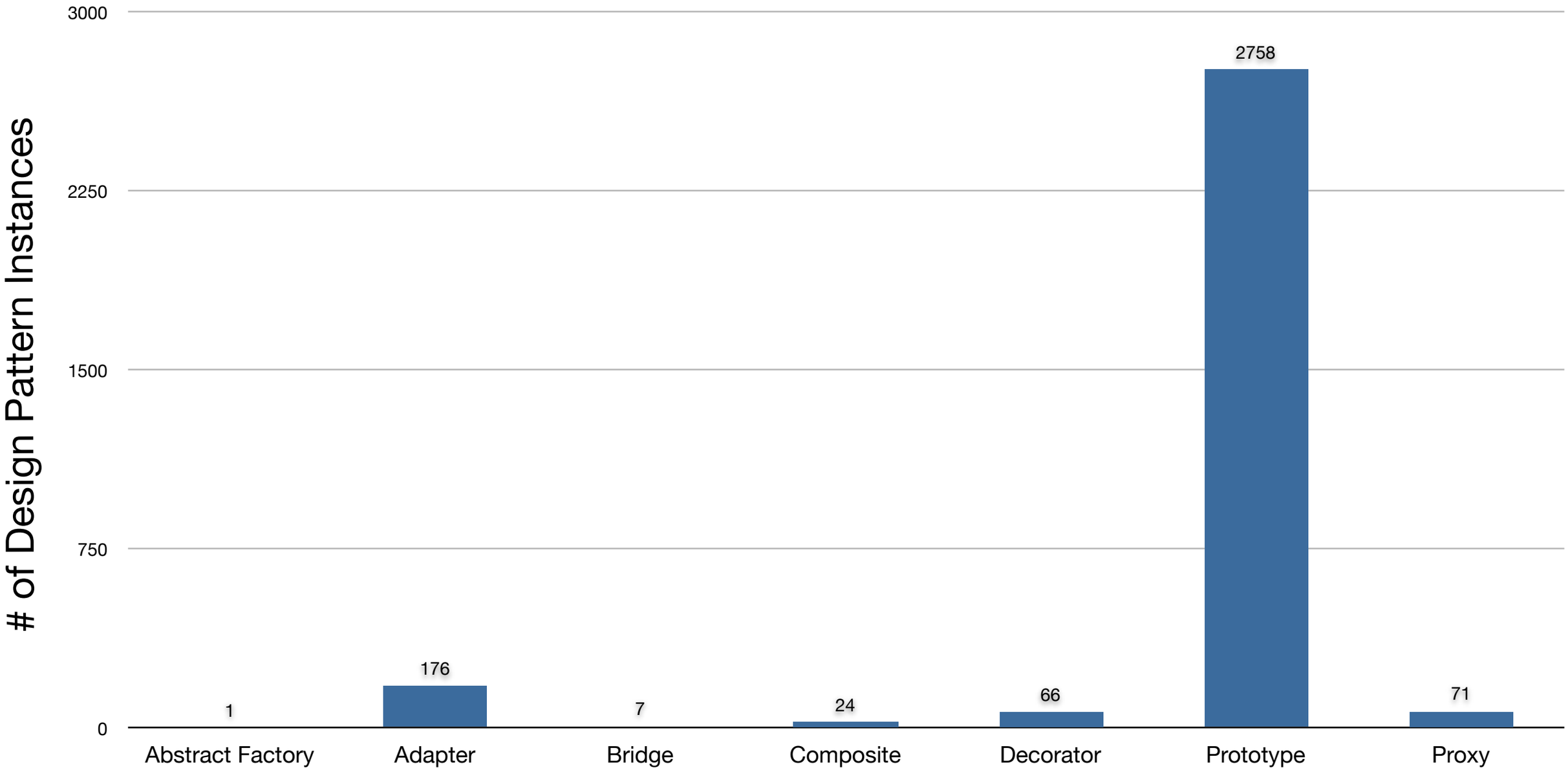
- Parameters
  - 100 individuals
  - 100 generations
  - Tournament selection with tournament size = 7
  - 90% crossover+mutation; 10% reproduction
- Experiments I-IV
  - 290 software designs
  - 5 trials per design, each with a unique random seed

Experiment I

- Purpose: to establish a baseline.
- No rewards or penalties; only metrics are considered
- Fitness function:
  - F = Metrics + 0.0*nodeCountPenalty 0.0*patternReward + 0.0*matchingSequenceReward
- Hypothesis: at least one *new* design pattern instance will evolve in each of the 290 models.
- Result: hypothesis validated.

Experiment I

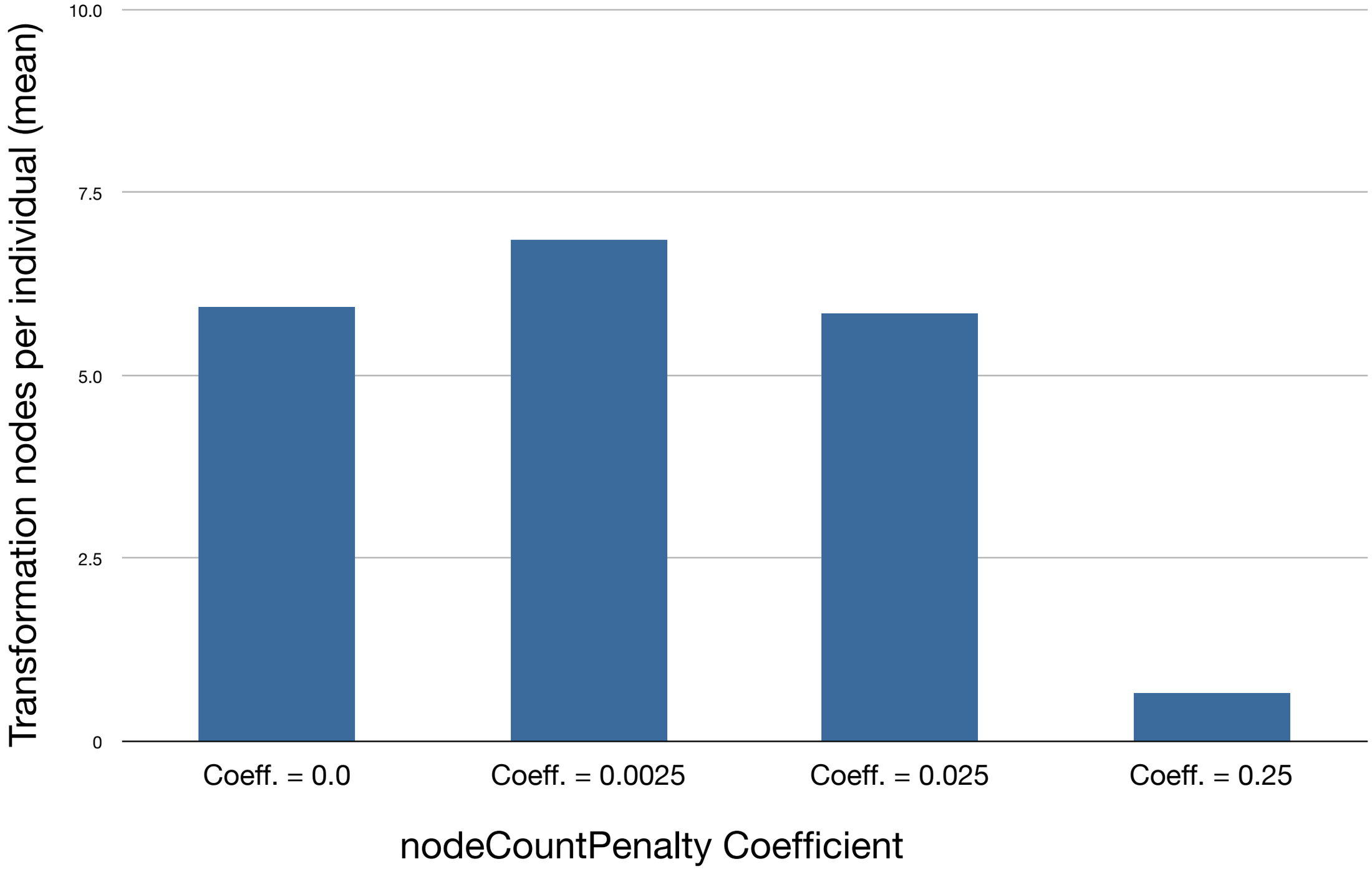
Mean # of transformation nodes: 5.94
New design pattern instances per model: 10.7



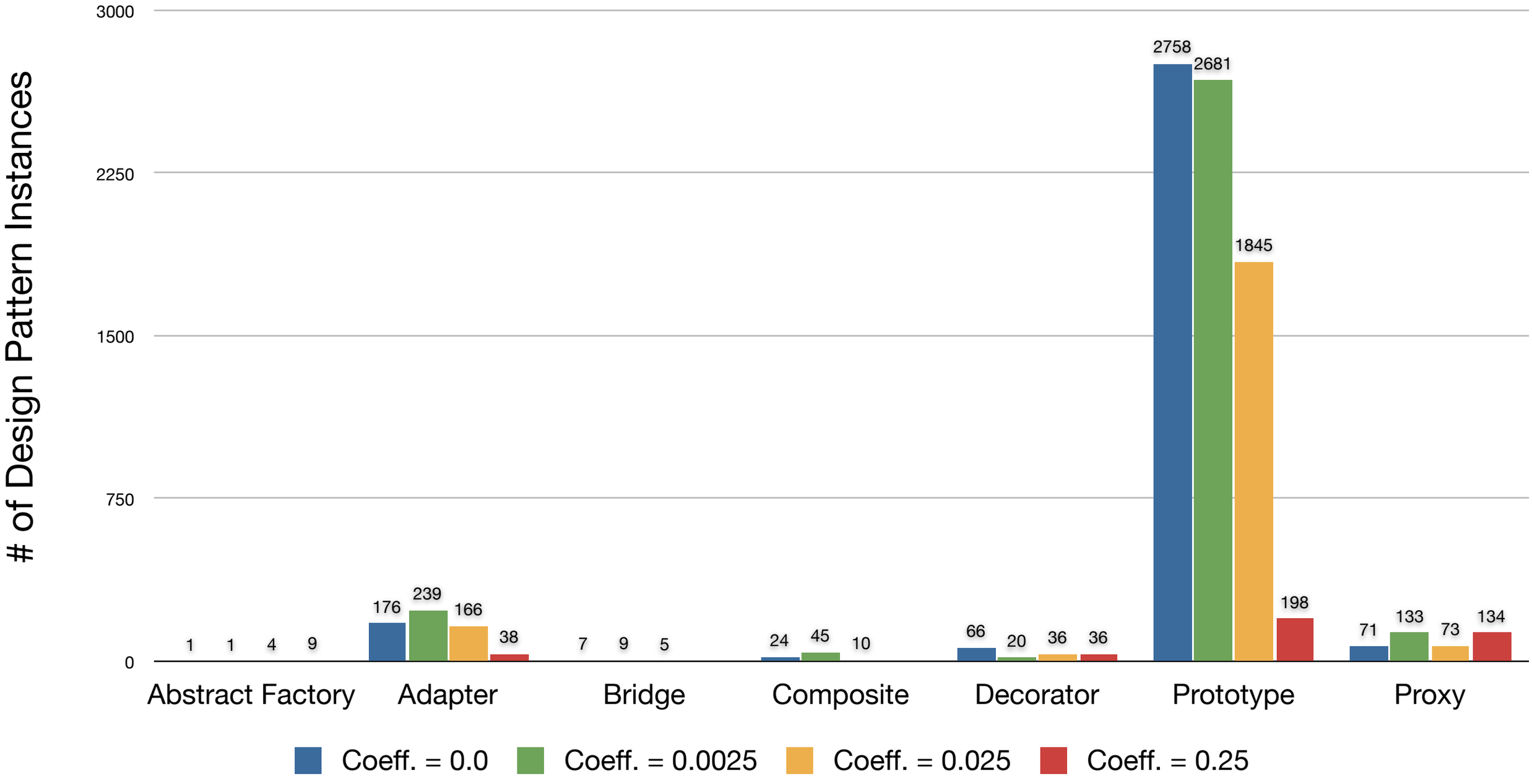
Experiment II

- Purpose: to penalize large numbers of transformations
- New term in fitness function: nodeCountPenalty
  - We vary the coefficient on this term: 0.0025, 0.025, 0.25, 0.5
- Fitness function:
  - F = Metrics + **X***nodeCountPenalty + 0.0*patternReward + 0.0*matchingSequencesReward
- Hypothesis: a larger coefficient will reduce the number of transformations in the average individual
- Result: hypothesis validated.

Experiment II



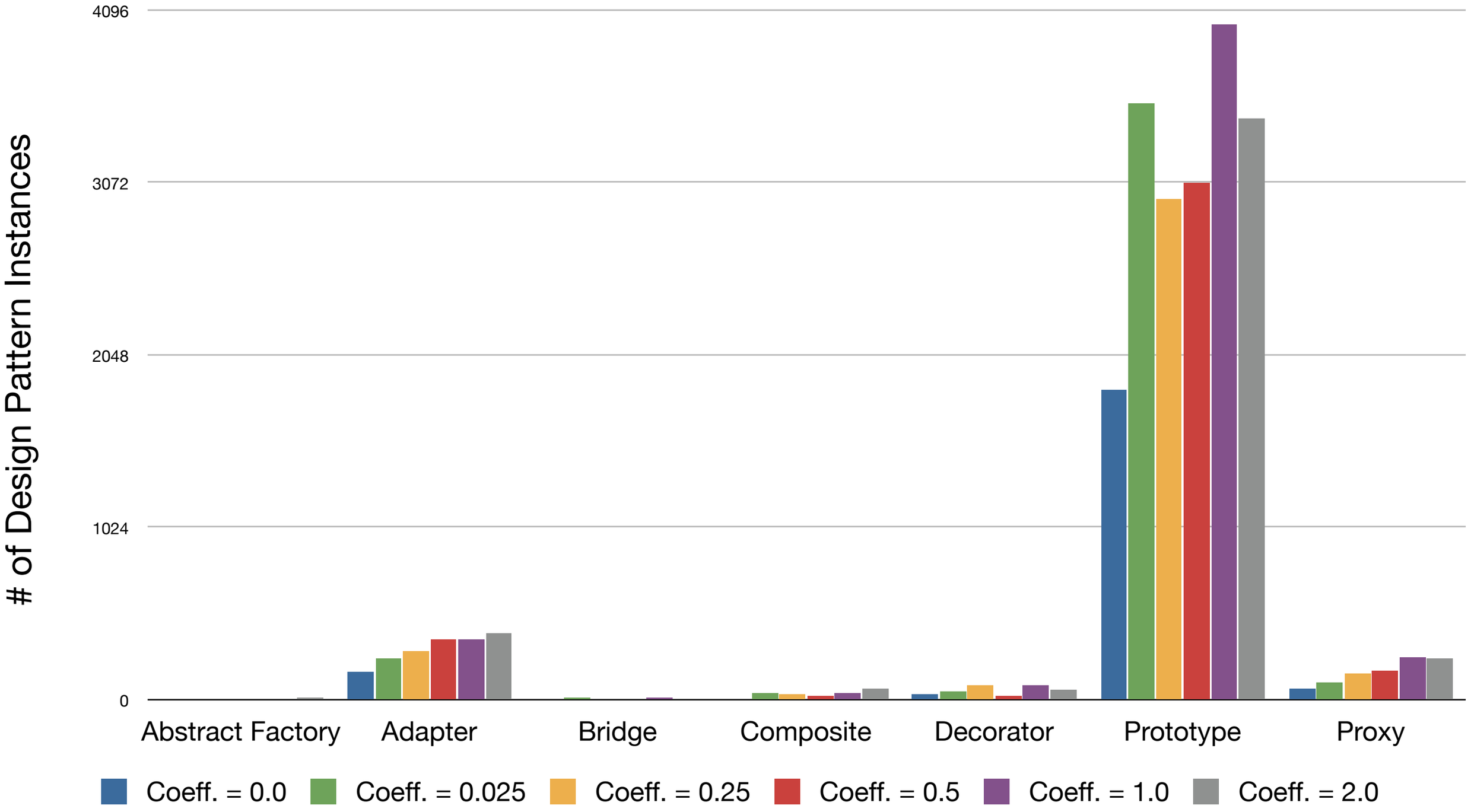
Experiment II



Experiment III

- Purpose: to reward creation of design pattern instances
- New term in fitness function: patternReward
  - We vary the coefficient on this term: 0.025, 0.25, 0.5, 1.0, 2.0
- Fitness function:
  - F = Metrics + 0.025*nodeCountPenalty + **X***patternReward + 0.0*matchingSequencesReward
- Hypothesis: a larger coefficient will lead to a larger number of design pattern instances on average.
- Result: hypothesis validated.

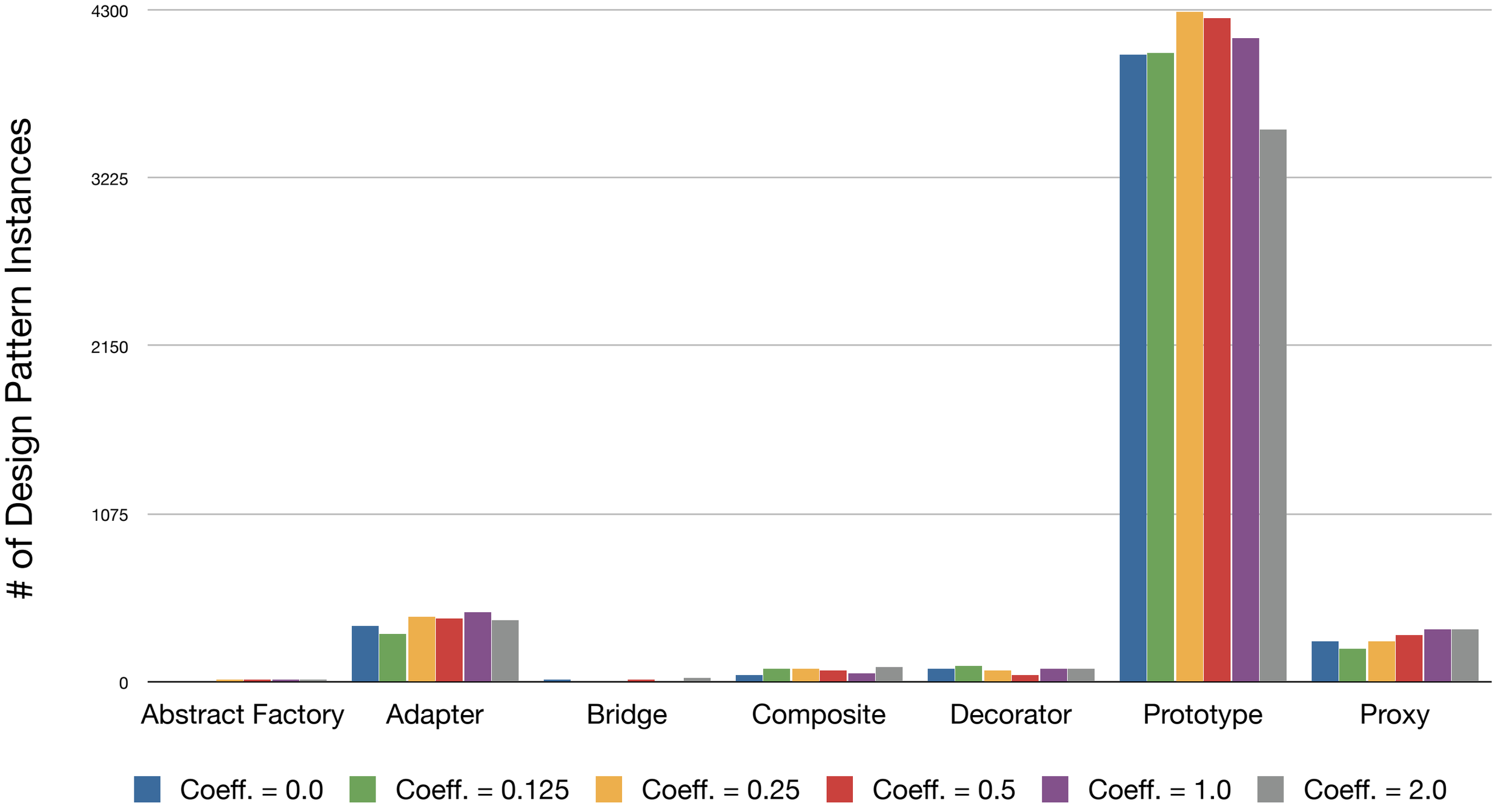
Experiment III



Experiment IV

- Purpose: to reward specific sequences of transformations
- New term in fitness function: matchingSequencesReward
  - We vary the coefficient on this term: 0.025, 0.25, 0.5, 1.0, 2.0
- Fitness function:
  - F = Metrics + 0.025*nodeCountPenalty + 1.0*patternReward + **X***matchingSequencesReward
- Hypothesis: a larger coefficient will lead to a larger number of design pattern instances on average.
- Result: hypothesis validated.

Experiment IV

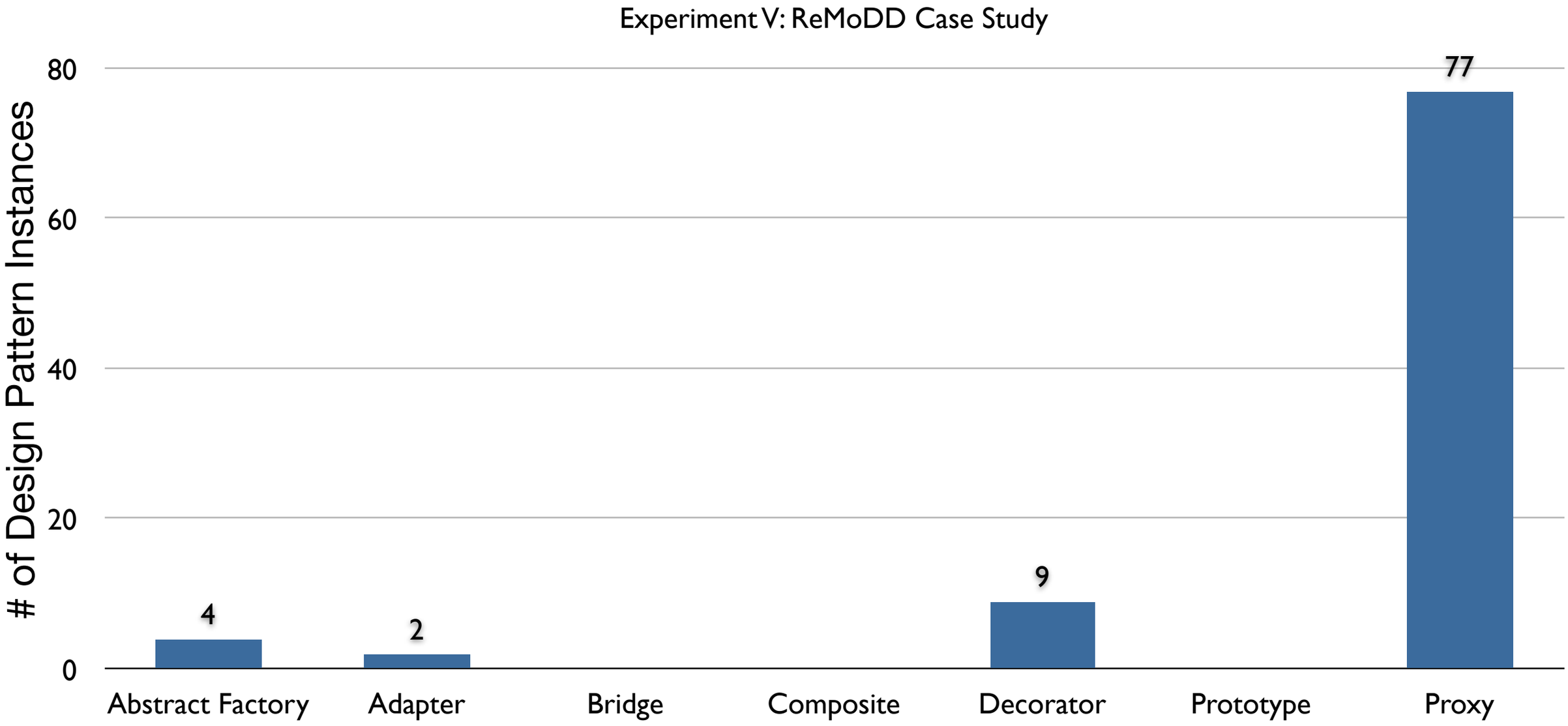


Experiment V

- Purpose: to test the approach on a large design.
- Parameters
  - 25 generations
- Fitness function:
  - F = Metrics + 0.025*nodeSizePenalty + 1.0*patternReward + 0.5*matchingSequencesReward
- Hypothesis: at least one instance of each supported design pattern type will evolve during the run.
- Result: **FIXME**.

Experiment V

- ReMoDD case study



Discussion

- Pattern complexity affects instance frequency
- Trends are not always monotonic
- Trials with no DP instances had higher quality than other trials for the same model
- In experiment IV, average design quality increased by 2.6

Conclusions

- EC can be used to support automated design refactoring
- Validated on large body of designs.
  - 290 models
  - real-world case study
  - exploration of several rewards and penalties
- Impact:
  - design assistance tools
  - step-by-step process for automated refactoring

Future Work

- Consider fitness for purpose of specific metrics and design patterns
- Explore synergy between metrics and DPs
- Look for other metric suites

Questions / Comments

References

[OCinneide2001] M. Ó Cinnéide. Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College, 2001.

[OKeeffe2008] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice, 20(5), 2008.

[Prechelt2001] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering, pages 1134–1144, 2001.

[Seng2006] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of ob ject-oriented systems. In Proceedings of the 8th annual conference on Genetic and evolutionary computation, pages 1909–1916. ACM New York, NY, USA, 2006.

Acknowledgements

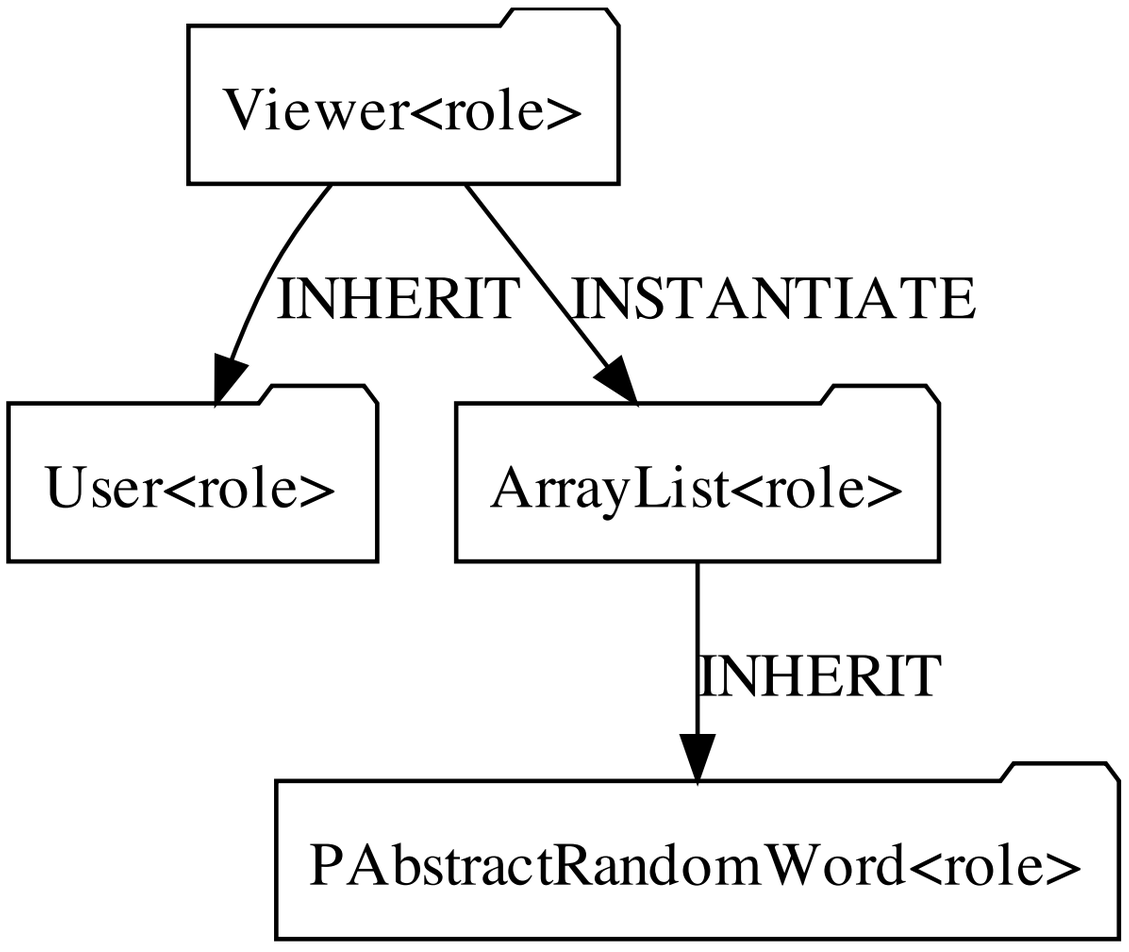
- Funding support from
  - Ford
  - QFC
  - NSF

QMOOD Metrics

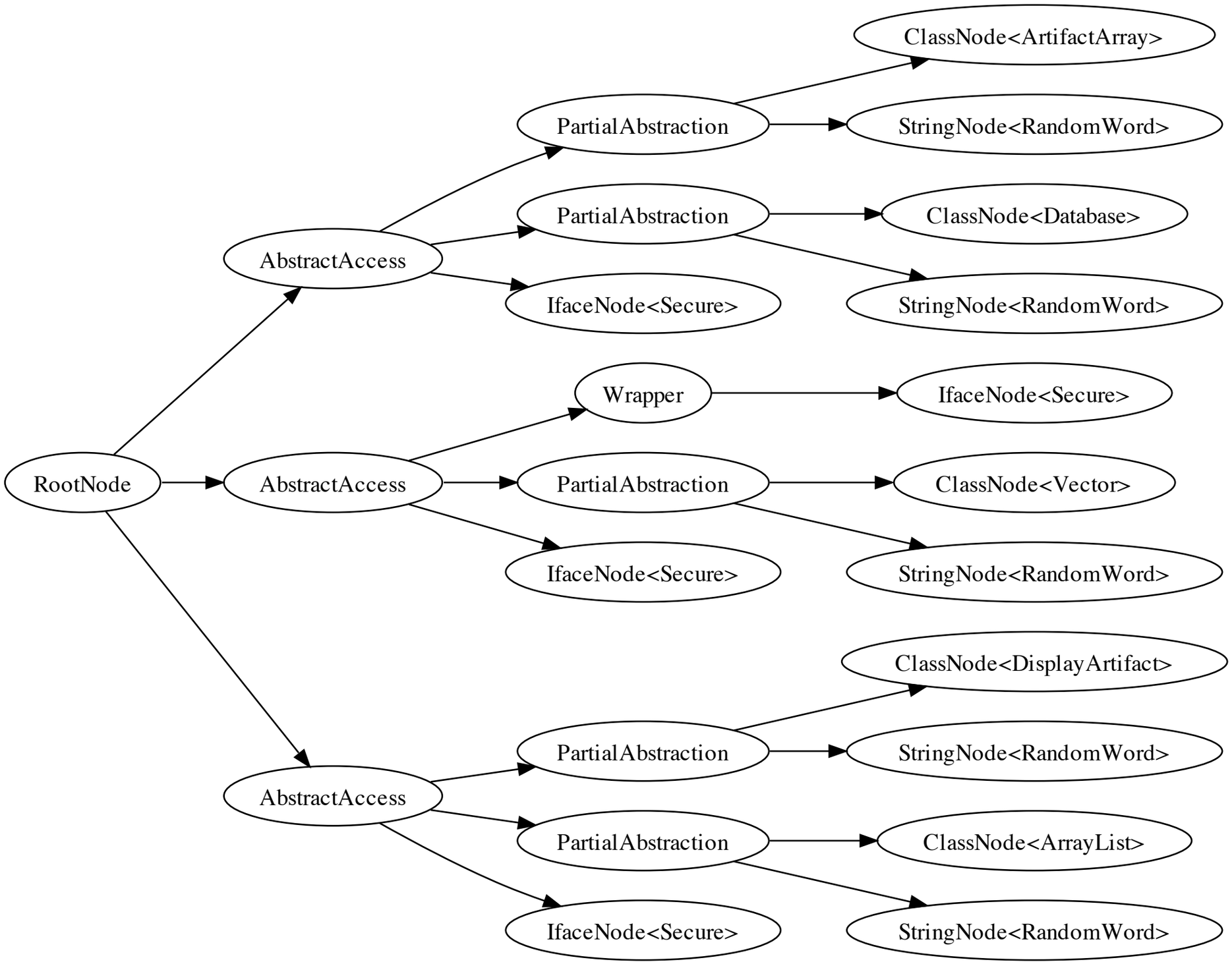
Bansiya & Davis

Metric Name and Abbreviation	Description
Design size in classes (DSIC)	The number of classes in the software design.
Number of hierarchies (NOH)	The number of class hierarchies in the software design.
Average number of ancestors (NOA)	The average number of other classes that a class inherits.
Number of polymorphic methods (NOPM)	The number of methods in the software design that exhibit polymorphic behavior.
Class interface size (CIS)	The average number of public methods in a class.
Direct class coupling (DCC)	A count of the number of classes that a given class is directly related to by attribute declaration or method return type.
Cohesion among methods of class (CAM)	The relatedness among methods of a class, computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class.
Number of methods (NOM)	The average number of methods in a class.
Data access metric (DAM)	The ratio of non-public (i.e., private or protected) attributes to the total number of attributes declared in a class. This is interpreted as the average of the ratios for all classes in the software design.
Measure of aggregation (MOA)	The average number of data declarations (e.g., fields) in a class whose data types are user-defined classes. We exclude classes that are part of the Java standard library and language.
Measure of functional abstraction (MFA)	The ratio of the total number of methods inherited by a class to the number of methods that are accessible by member methods of that class.

Evolved Design Fragment



Sample AST


RootNode
AbstractAccess
PartialAbstraction
ClassNode<ArtifactArray>
StringNode<RandomWord>
PartialAbstraction
ClassNode<Database>
IfaceNode<Secure>
StringNode<RandomWord>
Wrapper
IfaceNode<Secure>
AbstractAccess
PartialAbstraction
ClassNode<Vector>
IfaceNode<Secure>
StringNode<RandomWord>
AbstractAccess
PartialAbstraction
ClassNode<DisplayArtifact>
StringNode<RandomWord>
PartialAbstraction
ClassNode<ArrayList>
IfaceNode<Secure>
StringNode<RandomWord>