

# Enabling Refactoring with HTN Planning to Improve the Design Smells Correction Activity

Javier Pérez

`jperez@infor.uva.es`

`www.infor.uva.es/~jperez`

Universidad de Valladolid

BENEVOL 2008

Dec 11-12 2008, Eindhoven

# Design Smell Correction

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

**Problems encountered in the software's structure (code or design)**, that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), **that can be detected statically**, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), that can be detected statically, that **do not produce compile or run-time errors**, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but **negatively affect software quality factors**. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - inconsistency management (detect and resolve inconsistencies)
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Object-Oriented Software Design Smells

## Design Smells

Problems encountered in the software's structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - **inconsistency management (detect and resolve inconsistencies)**
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

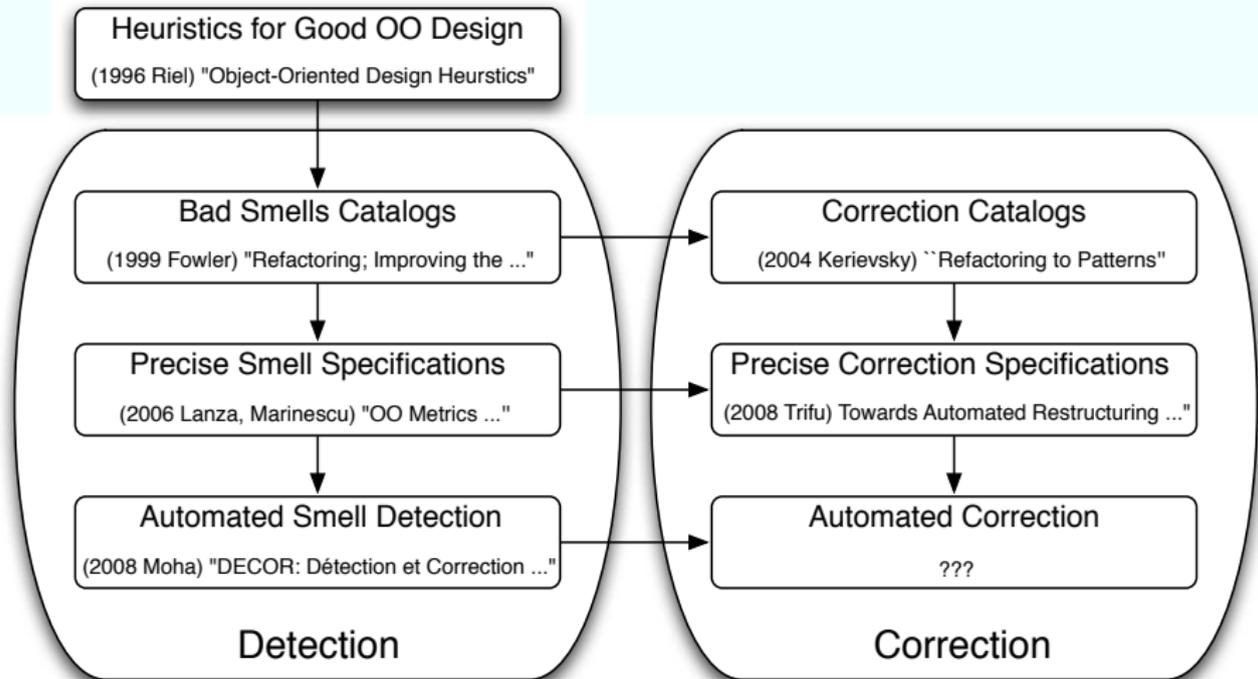
# Object-Oriented Software Design Smells

## Design Smells

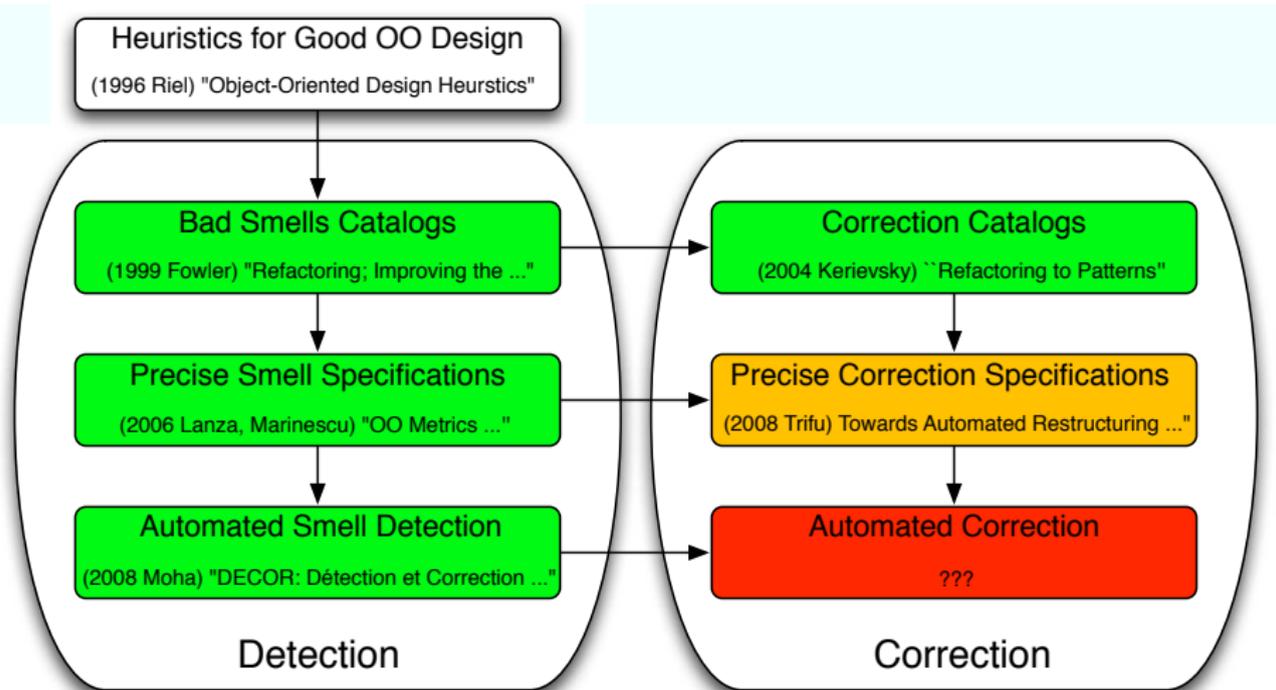
Problems encountered in the software's structure (code or design), that can be detected statically, that do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future.

- In the context of software inconsistencies:
  - consistency maintenance (keeping models consistent)
  - **inconsistency management (detect and resolve inconsistencies)**
  - co-evolution (manage consistency between different artefacts)
- Design smells are corrected with refactorings

# Brief History of Design Smell Management



# Brief History of Design Smell Management



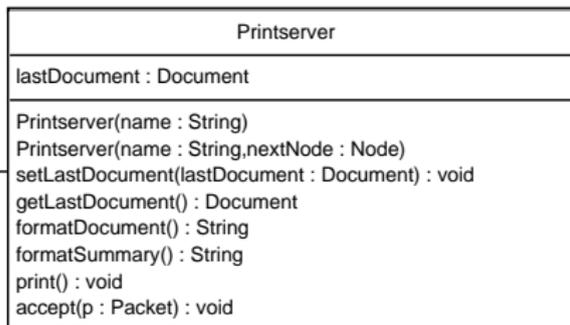
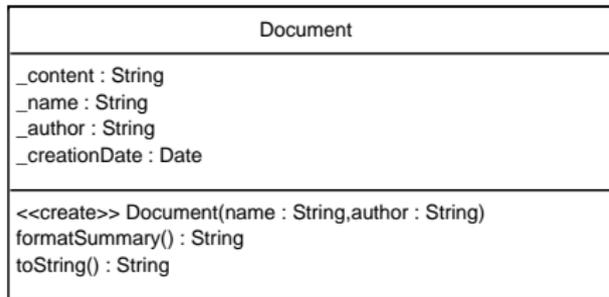
# Smell Example: Feature Envy

# Feature Envy

## Feature Envy

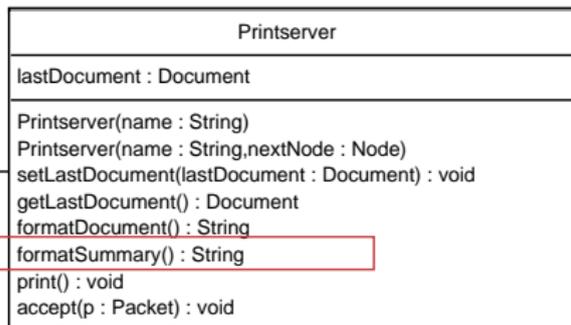
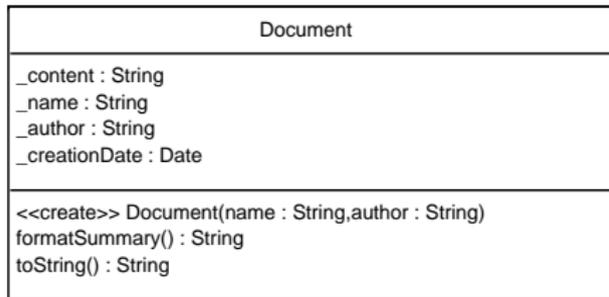
“... a method that seems more interested in a class other than the one it actually is in.” (Fowler *et al.*, 1999)

# Feature Envy Example



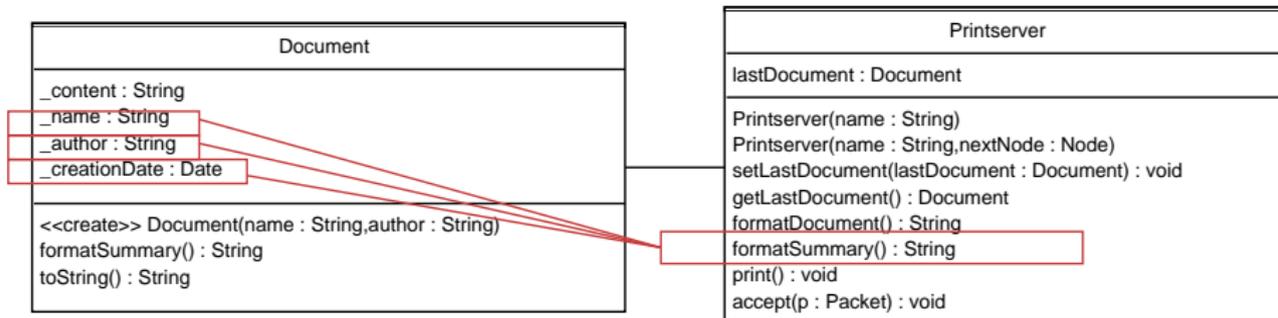
- `formatSummary()` uses many attributes from `Document` and none from its own class.
- The strategy is to **move** the **method** to `Document` but a method with the same signature already exists.

# Feature Envy Example



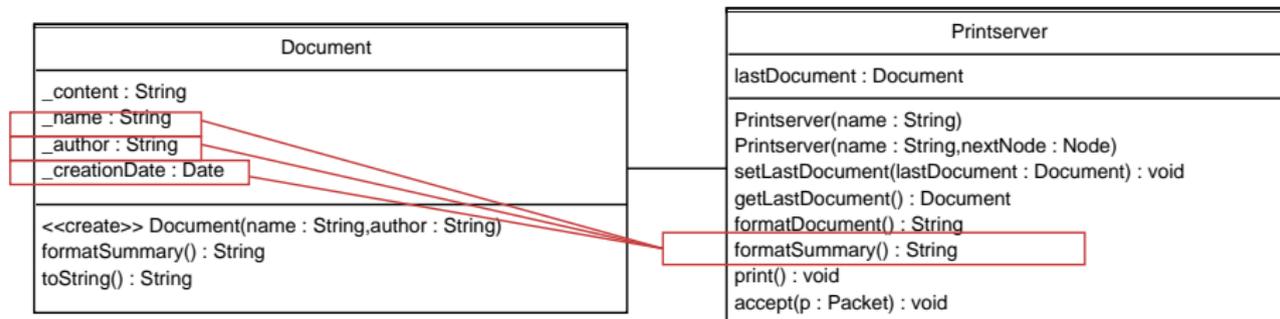
- `formatSummary()` uses many attributes from `Document` and none from its own class.
- The strategy is to **move** the **method** to `Document` but a method with the same signature already exists.

# Feature Envy Example



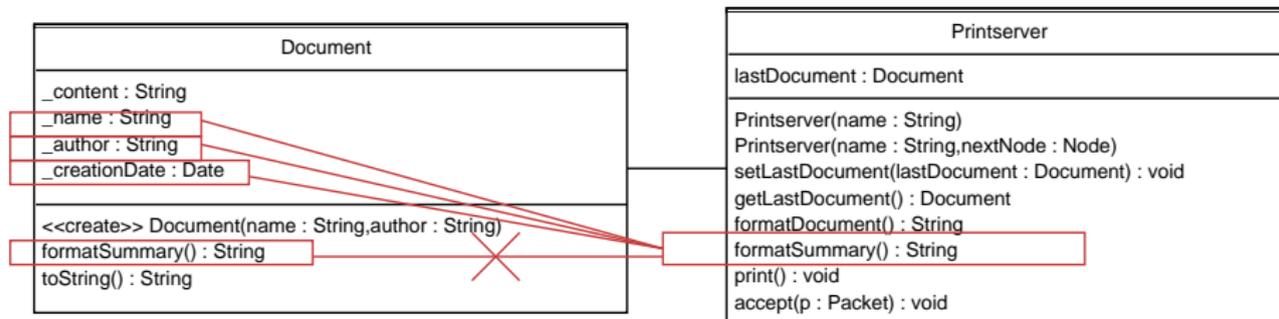
- `formatSummary()` uses many attributes from `Document` and none from its own class.
- The strategy is to **move** the **method** to `Document` but a method with the same signature already exists.

# Feature Envy Example



- `formatSummary()` uses many attributes from `Document` and none from its own class.
- The strategy is to **move the method** to `Document` but a method with the same signature already exists.

# Feature Envy Example



- `formatSummary()` uses many attributes from `Document` and none from its own class.
- The strategy is to **move** the **method** to `Document` but a method with the same signature already exists.

# Problems in Automated Correction

- Which is the strategy to correct a smell?
  - Feature Envy ( $m$ )  $\Rightarrow$  move method  $m$  close to data, from class  $s$  to class  $t$
- Which is the precise strategy instance to use?
  - Feature Envy (`formatSummary`)  $\Rightarrow$  move method **`formatSummary`** from **`Printserver`** to **`Document`**
- How to apply the strategy instance?
  - move method `formatSummary` from `Printserver` to `Document`  $\Rightarrow$ 
    - 1 first **remove `formatSummary` in `Document`** or **rename `formatSummary` in `Document`** or **rename `formatSummary` in `Printserver`**
    - 2 then **move method `formatSummary` from `Printserver` to `Document`**

# Refactoring Planning

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system's behaviour, in order to obtain that desirable system redesign.

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system's behaviour, in order to obtain that desirable system redesign.

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

**Specification of a refactoring sequence** which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system's behaviour, in order to obtain that desirable system redesign.

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which **matches a system redesign proposal**, and that can be immediately executed to modify the system, without changing the system's behaviour, in order to obtain that desirable system redesign.

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that **can be immediately executed** to modify the system, without changing the system's behaviour, in order to obtain that desirable system redesign.

# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed **to modify the system, without changing the system's behaviour**, in order to obtain that desirable system redesign.

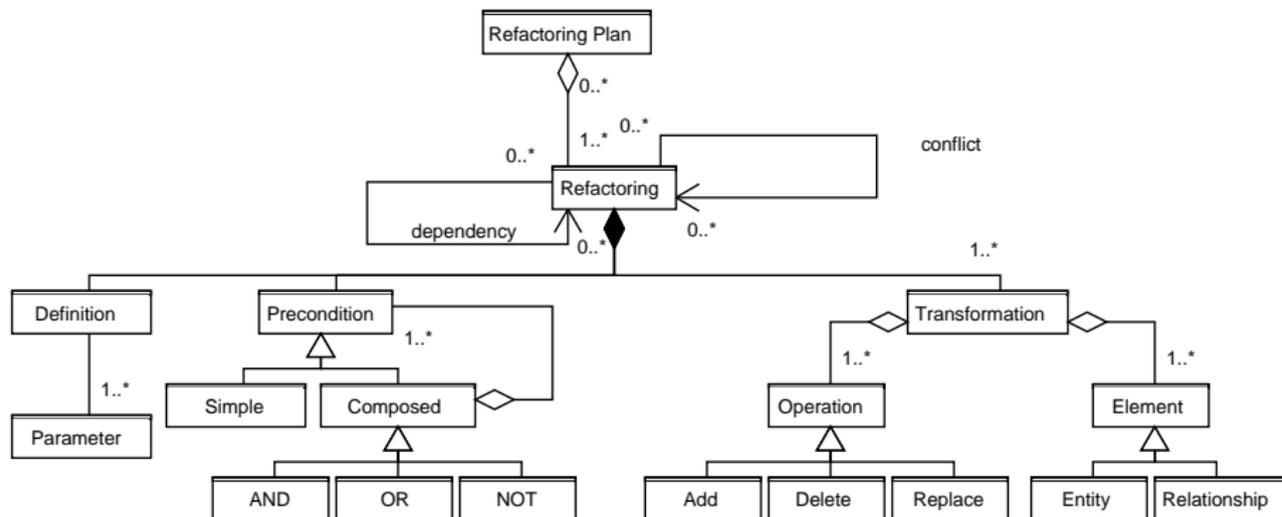
# Refactoring Plans

- The objective: Instantiate smell correction strategies into a correction plan which could be effectively applied, or at least could guide the developer through the process.

## Refactoring Plan

Specification of a refactoring sequence which matches a system redesign proposal, and that can be immediately executed to modify the system, without changing the system's behaviour, in order **to obtain that desirable system redesign**.

# Refactoring Model



- Refactoring plans can be computed with automated planning



# Automated Planning

## Definition

**Automated planning** is an artificial intelligence technique to generate sequences of actions that will achieve a certain goal when they are performed.

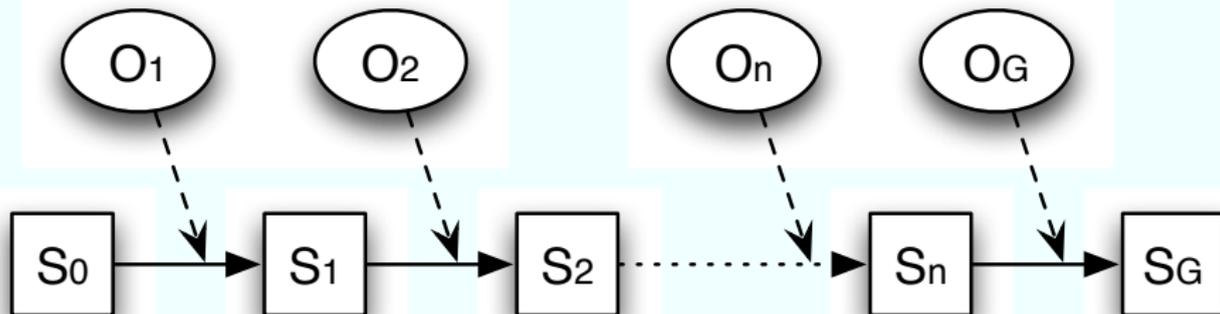
- Example: Getting apples and a book.

**The state of the world:** at (grocery) AND not (have (apples))

**Actions:** buy (apples); moveTo (bookstore)

**Goals:** have (book) AND have (apples)

## Plan



# Classical Planning Operators (STRIPS)

- **World's state:** list of terms
- **Operators:**
  - definition: name + arguments
  - precondition
  - effect list (add): terms to add to the state
  - effect list (deletes): terms to remove from the state
- **Problem:**
  - initial state
  - goal: list of terms
- **General planning approach:** chain operators by matching their effects and preconditions

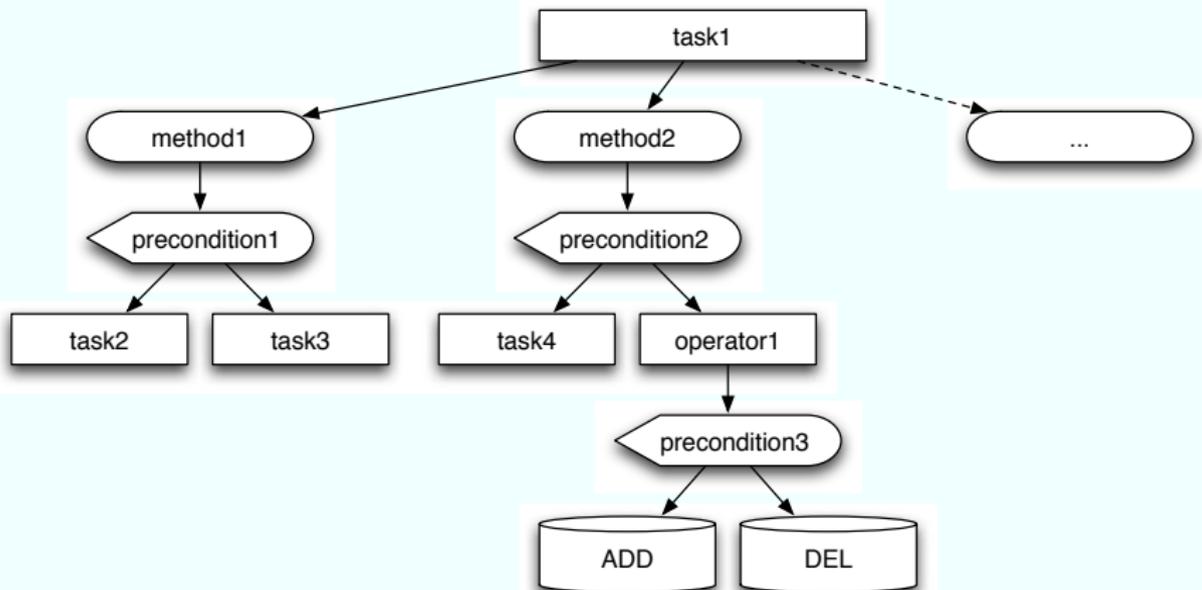
# Some Types of Planners

- Depending on the **planning space**:
  - state space planning
  - plan space planning
- Depending on the **search direction**:
  - forward searching
  - backwards searching
- Depending on when the **operator ordering** is committed:
  - total-order planning
  - partial-order planning
- I'm using **Hierarchical Task Network (HTN) planning**.

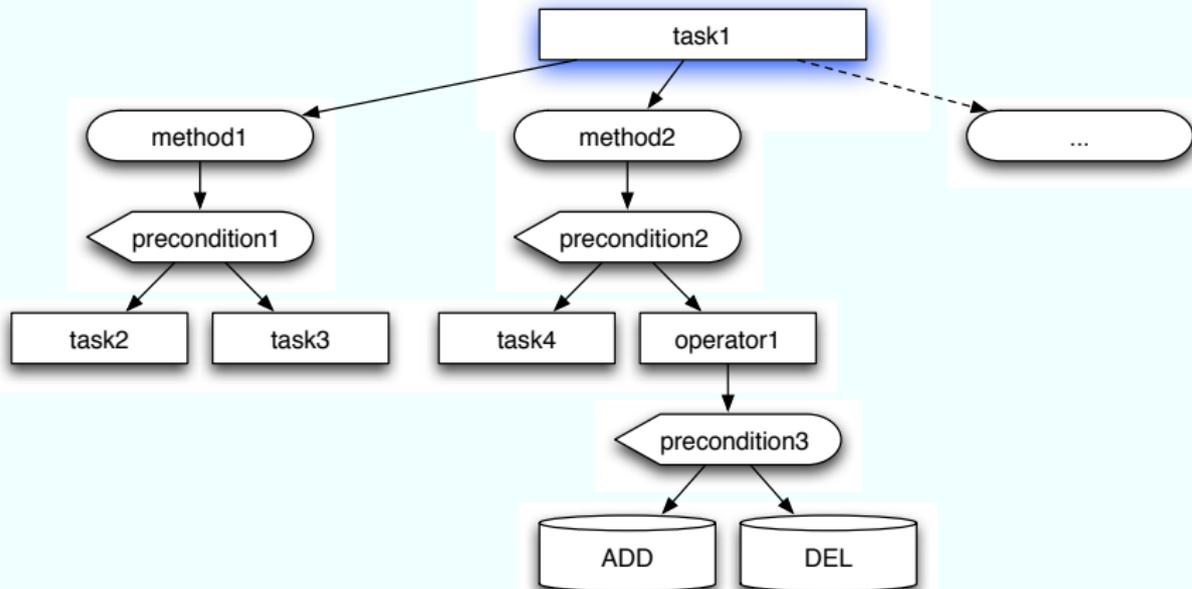
# Some Types of Planners

- Depending on the **planning space**:
  - state space planning
  - plan space planning
- Depending on the **search direction**:
  - forward searching
  - backwards searching
- Depending on when the **operator ordering** is committed:
  - total-order planning
  - partial-order planning
- I'm using **Hierarchical Task Network (HTN) planning**.

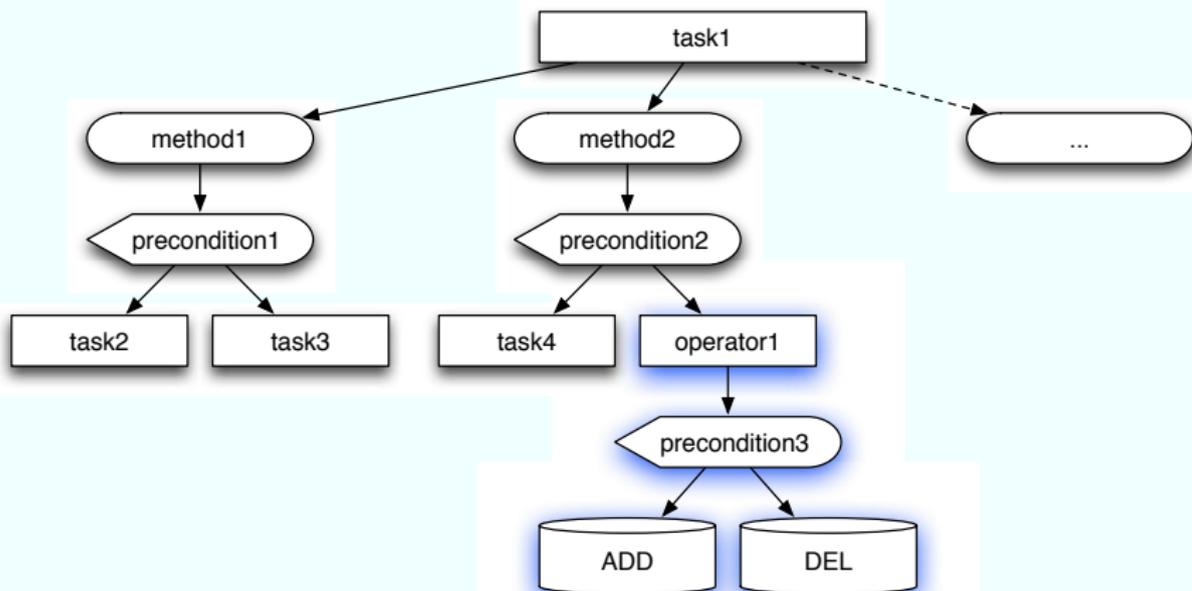
# Hierarchical Task Network (HTN) Planning



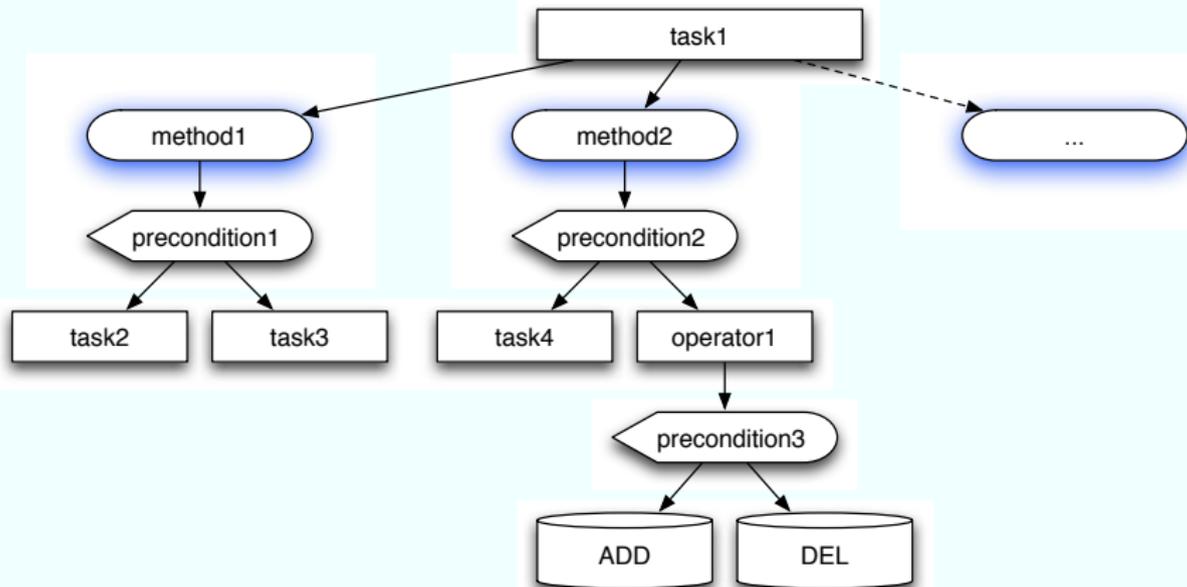
# Hierarchical Task Network (HTN) Planning



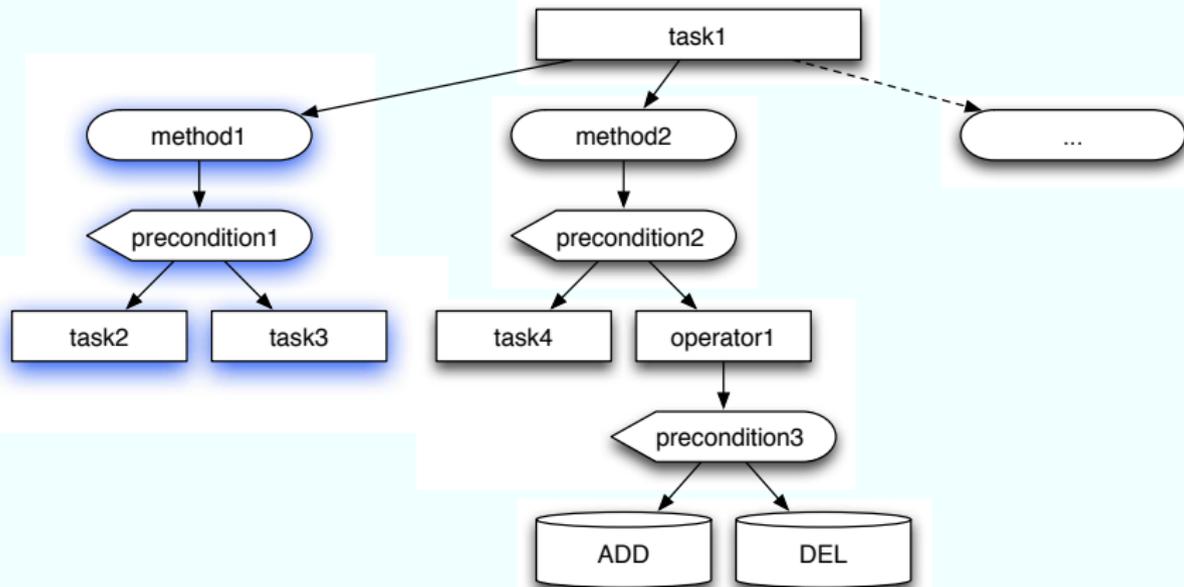
# Hierarchical Task Network (HTN) Planning



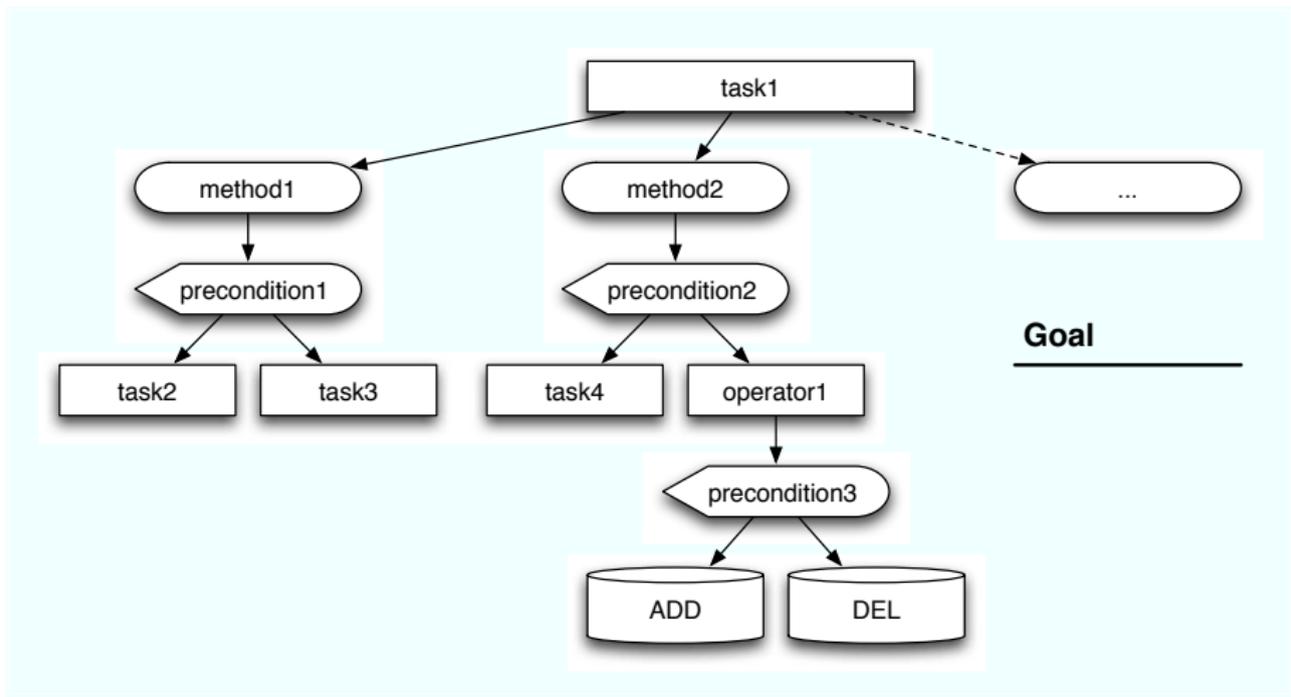
# Hierarchical Task Network (HTN) Planning



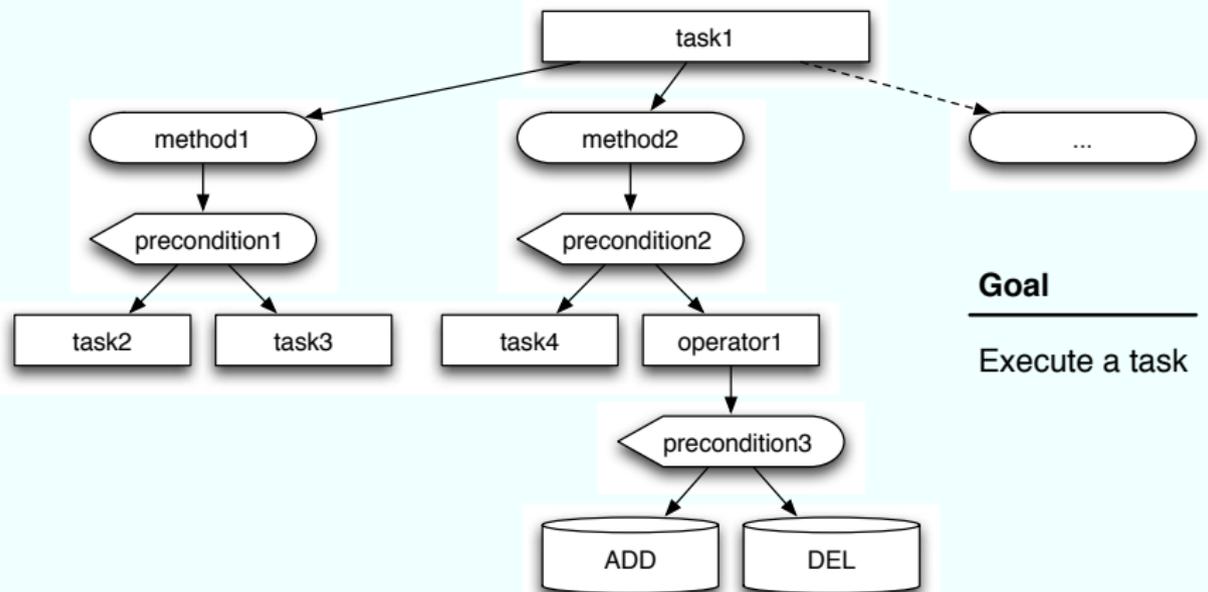
# Hierarchical Task Network (HTN) Planning



# Hierarchical Task Network (HTN) Planning



# Hierarchical Task Network (HTN) Planning

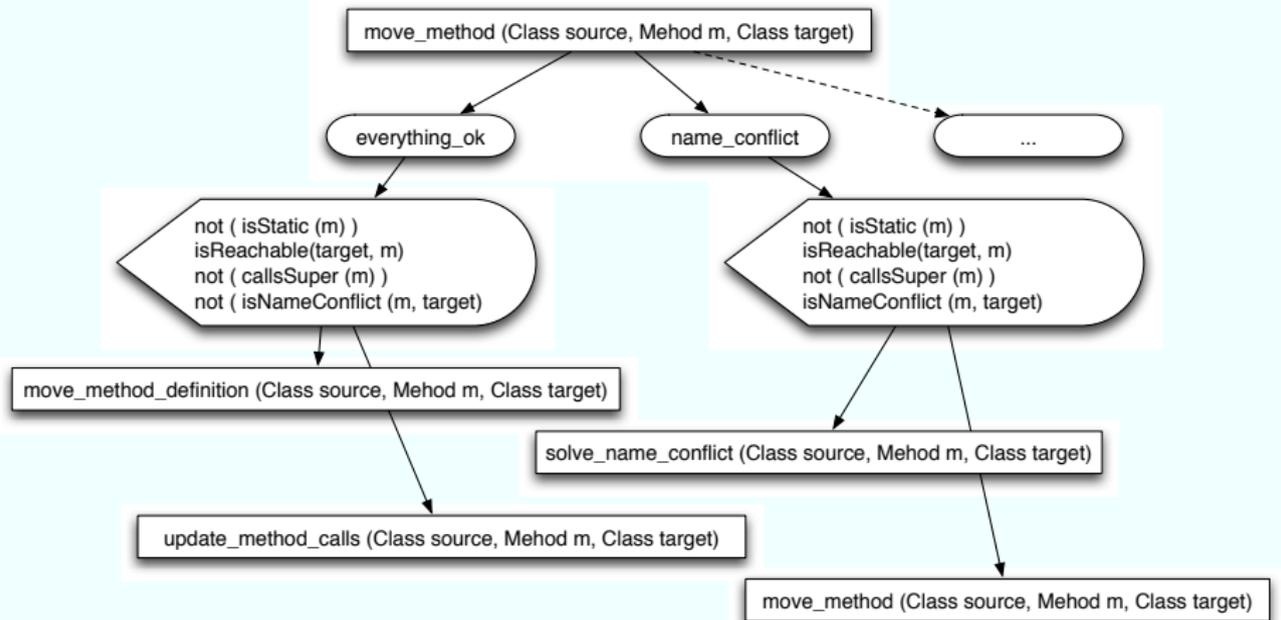


# Smell Correction with HTN Planning

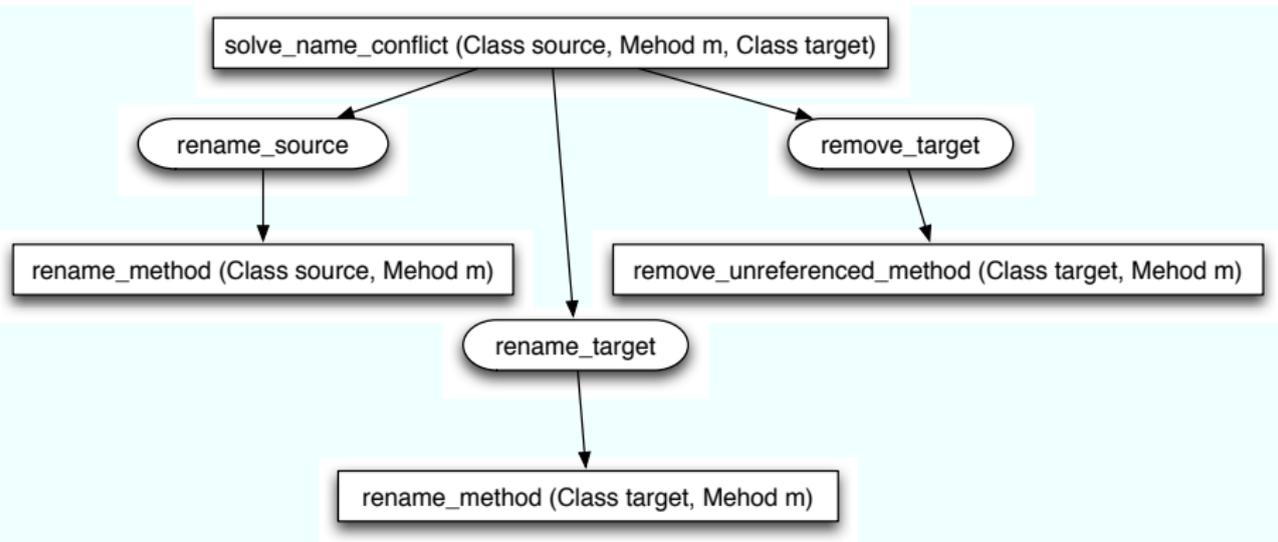
- **World's state:** AST represented by first order logic formulas
- **Operators:** refactoring substeps
- **Tasks:**
  - refactorings strategies
  - smell correction strategies
- **Goals:** Execute a smell correction strategy
- **Planning Problem:** Execute a particular smell correction strategy over a particular version of a system

# Planning for “Feature Envy”

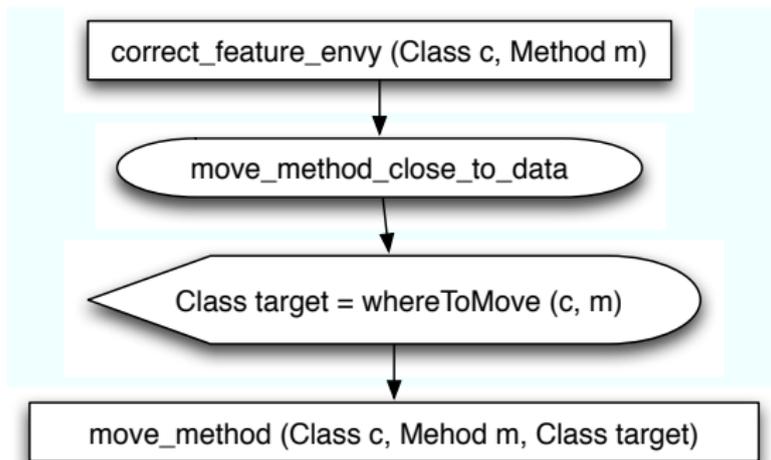
## HTN for "move method"



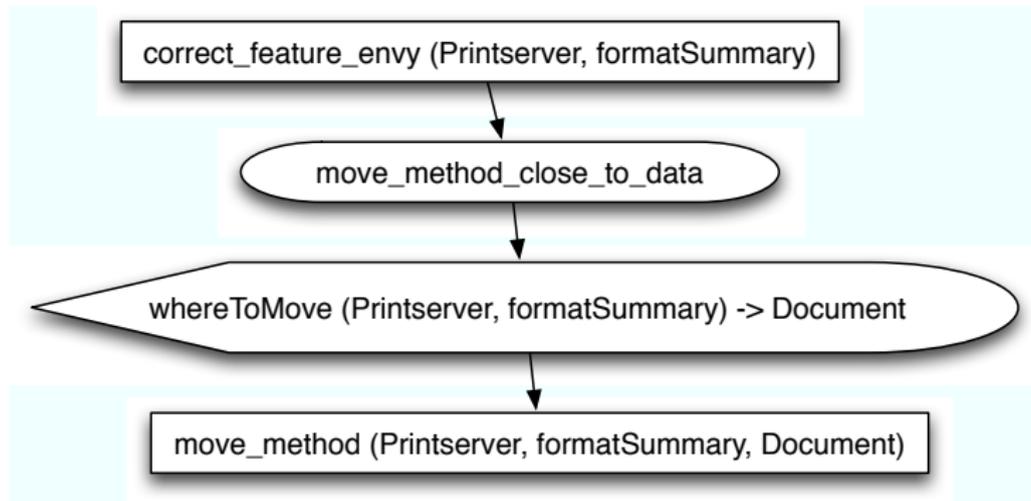
## HTN for "solve conflict"



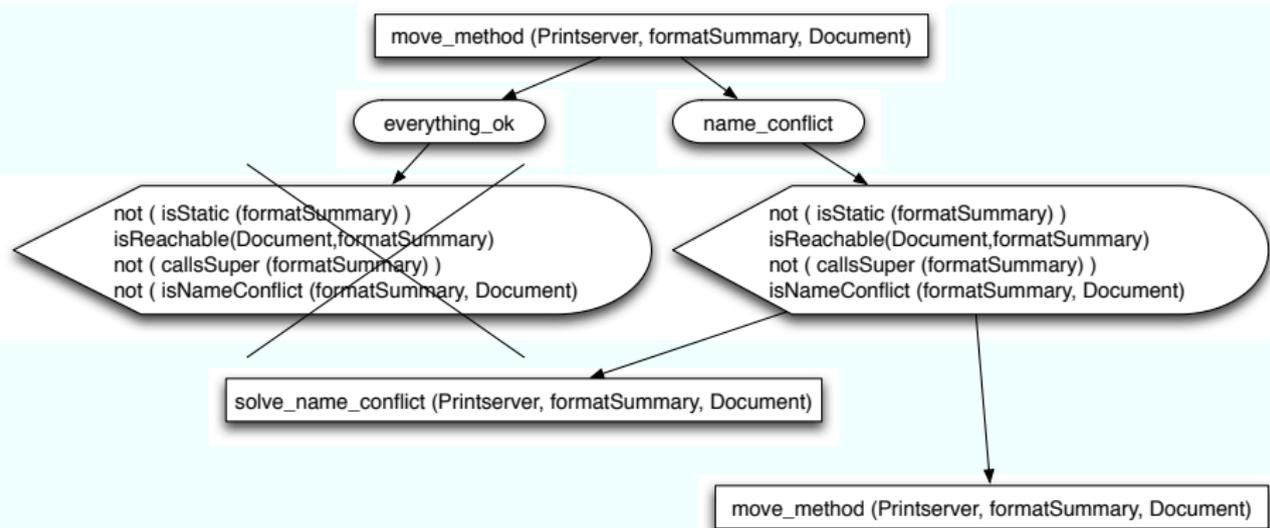
# HTN for "feature envy"



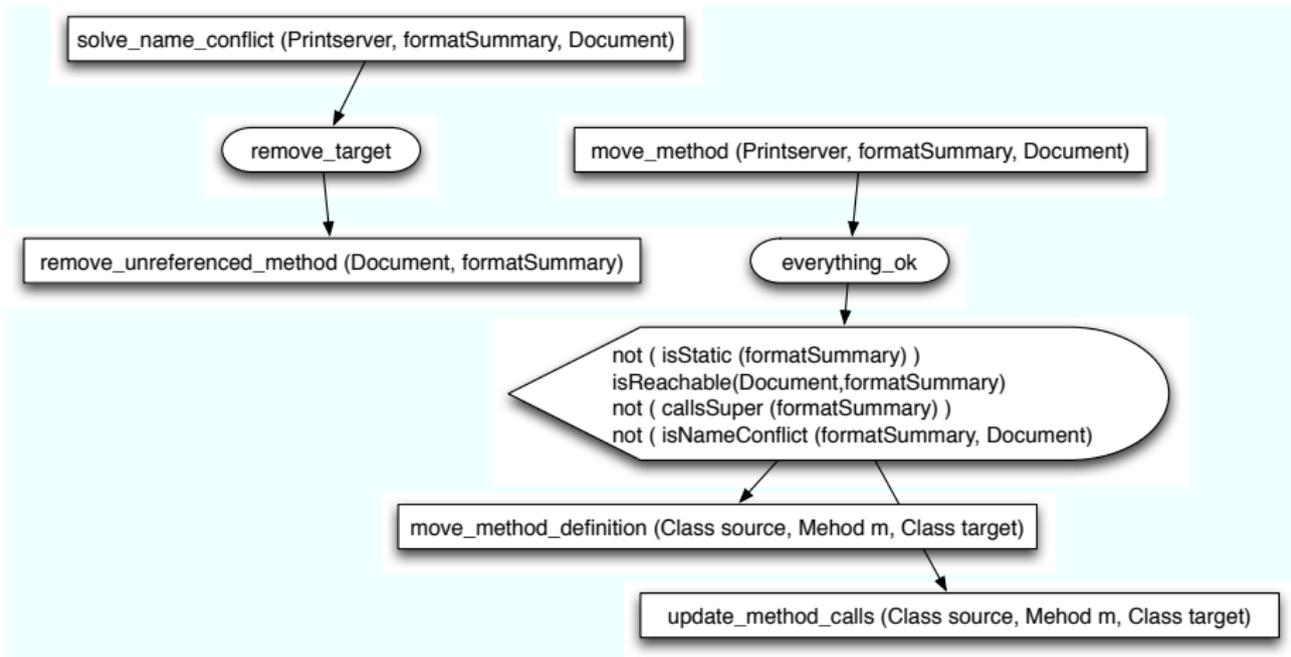
# Planning for "feature envy"



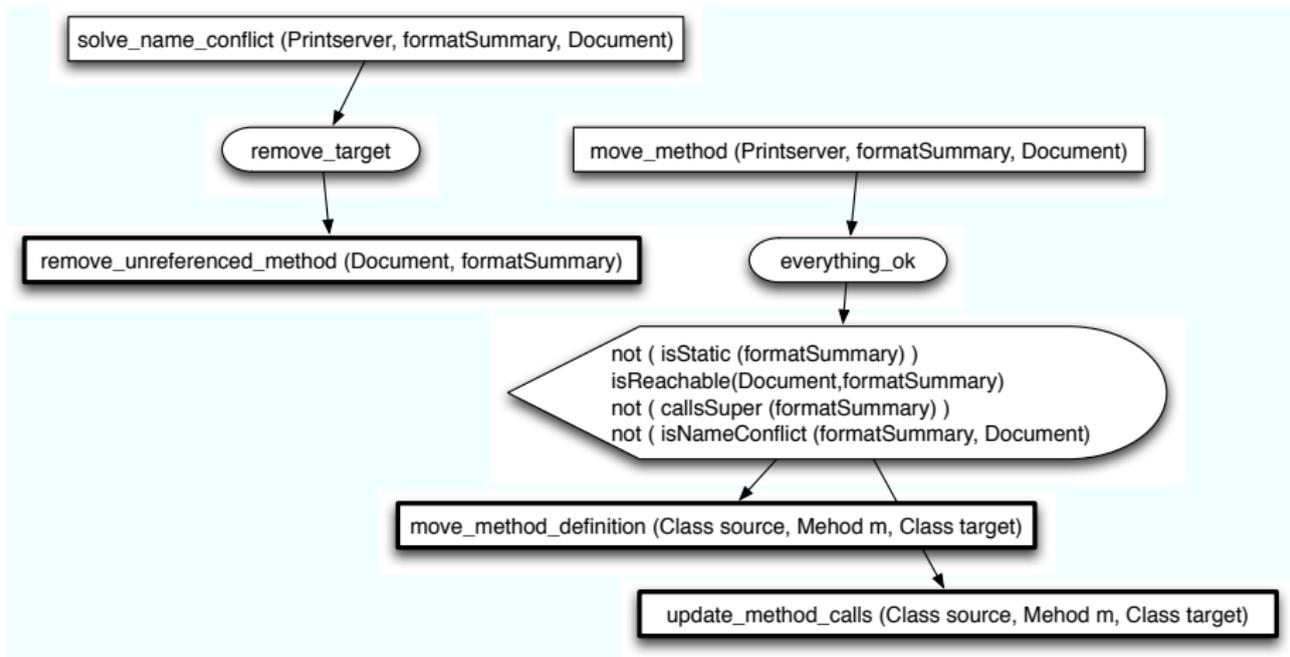
# Planning for "move method" 1



# Planning for "move method" 2



# Planning for "move method" 2



# Conclusions and Future Work

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Conclusions

- Design smell management can keep being improved, working on specification and automation of the correction activity.
- To do that, correction strategies must be planned ahead for each specific case.
- This can be done with automated planning and specifically with HTN planning:
  - HT networks can accommodate correction strategies, combining procedural and non-deterministic searching.
  - HTN planning offers good balance between procedural execution and non-determinism.
  - The planner can be incrementally extended, adding new methods and improving the existing ones.

# Future Work

- Implement refactoring specifications
- Implement design smell correction strategies
- Run experiments on real systems
- Integrate the planer with other tools for:
  - refactoring dependencies computation
  - metrics computation
  - ...

# Enabling Refactoring with HTN Planning to Improve the Design Smells Correction Activity

Javier Pérez

`jperez@infor.uva.es`

`www.infor.uva.es/~jperez`

Universidad de Valladolid

BENEVOL 2008

Dec 11-12 2008, Eindhoven