# GHOST

Generated by Doxygen 1.8.6

Tue Jun 24 2014 18:32:44

# Contents

# Chapter 1

# GHOST

GHOST documentation

**Author**

Florian Richoux

## Introduction

GHOST (General meta-Heuristic Optimization Solving Tool) is a C++11 library designed for StarCraft: Brood war. GHOST implements a meta-heuristic solver aiming to solve any kind of combinatorial and optimization RTS-related problems represented by a CSP. It is an generalization of the Wall-in project (see `github.com/richoux/Wall-in`).

The source code is available at `github.com/richoux/GHOST`, and the documentation pages at `richoux.github.io/GHOST`. GHOST is under the terms of the GNU GPL v3 licence.

### Scientific papers:

- Florian Richoux, Jean-François Baffier and Alberto Uriarte, GHOST: A Combinatorial Optimization Solver for RTS-related Problems (to appear).
- Florian Richoux, Alberto Uriarte and Santiago Ontañón, Walling in Strategy Games via Constraint Optimization (to appear in AIIDE 2014 proceedings).
- Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill and Mike Preuss, `A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft`, Transactions on Computational Intelligence and AI in Games, IEEE, 2013.

## A short CSP/COP tutorial

### Intuition behind CSP and COP

Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) are two close formalisms intensively used in Artificial Intelligence to solve combinatorial and optimization problems. They allow you to easily express what your problem is, and offer you a uniform way to solve all problems you can describe by a CSP or a COP.

The difference between a CSP and a COP is simple:

- A CSP models a satisfaction problem, that is to say, a problem where all solutions are equivalent, so you just want to find one of them. Example: find a solution of a Sudoku grid. Several solutions may exist, but finding one is sufficient, and no solutions seem better than another one.
- A COP models an optimization problem, where some solutions are better than others. Example: you may have several paths from your home to your workplace, but some of them are shorter.

Let start by defining a CSP. To model your problem by a CSP, you need to define three things:

- V, the set of variables of your CSP.
- D, the domain of your CSP, that is to say, the set of values your variable can take.
- C, the set of constraint symbols of your CSP.

Let's take a simple example:

- V = {x, y, z}. The variables of our CSP would be *x*, *y* and *z*.
- D = {0, 1, 2}. Our variable x, y and z can take a value from D, ie, be either equals to *0*, *1* or *2*. We can have for instance x = 1, y = 1 and z = 0.
- C = { =, $\neq$, $<$ }. We have three types of constraint symbols here: *equal*, *different* and *less than*.

Ok, now what? Well, to describe our problem, we have to build a formula from our CSP. This is a bit like playing to Lego: you combine blocks to build bigger blocks, then you combien your bigger blocks to create an object.

Here, your blocks are your variables. You can combine them with a constraint symbol to build a bigger block, ie, a constraint. For instance, we can build the constraint (x=z), or the constraint (z$\neq$y), etc.

Then, we can build a formula by combining constraints. Combining means here we have a conjuction, ie a "and"-operator linking two constraints. A formula with the CSP describe above could be for instance

```
(z=y) AND (y&ne;x) AND (x<z)
```

## A first example of a CSP formula

Concider the following CSP:

- V = {a, b, c, d}.
- D = {0, 1, 2}.
- C = { =, $\neq$, $<$ }.

and suppose our problem is modeled by the formula

```
(a=b) AND (b&ne;d) AND (d<c) AND (b<c)
```

A solution of our problem is a good evaluation of each variable to a value of the domain D. In other words, if we find a way to give a value from D to each variable of V such that all constraints are true (we also say, are satisfyed), then we have a solution to our problem.

For instance, the evaluation a=1, b=1, c=2 and d=1 is not a solution of the formula, because the second constraint (b$\neq$d) is not satisfyed (indeed (1$\neq$1) is false).

However, the evaluation a=1, b=1, c=2 and d=0 satisfies all constraints of the formula, and is then a solution to our problem.

### A concrete problem modeled by a CSP formula

Ok, now how to model a problem through a CSP formula? This is not always trivial and require some experience. Let see how to model two famous graph problems with the CSP formalism.

I assume the reader know what a graph is; otherwise here is the main idea: it is a set of vertices where some of them are linked by an edge. See the picture below, an example of graph with four vertices (named A, B, C and D). Graphs are simple mathematical objects but expresive enough to model complex problems, like finding the shortest path between two cities (your GPS use graphs!).

Let consider the 3-COLOR problem: Given a graph, is it possible to colorize each vertex with one of the three available colors (say, red, blue and green) such that there is no couple of vertices linked by an edge sharing the same color?

Before continuing, think about:

- How could you define a CSP modelling the 3-COLOR problem?
- If I give you the graph above, how could you built a CSP formula to solve the 3-COLOR problem for this graph?

## How to use GHOST?

TODO

## How to define and solve my own CSP problem with GHOST?

TODO

# Chapter 2

# Namespace Index

## 2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

# Chapter 3

# Hierarchical Index

## 3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

# Chapter 4

# Class Index

## 4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 5

# File Index

## 5.1 File List

Here is a list of all files with brief descriptions:

# Chapter 6

# Namespace Documentation

## 6.1 ghost Namespace Reference

### Classes

- class Constraint

  *Constraint is the class encoding constraints of your CSP/COP.*
- class WallinConstraint
- class Overlap
- class Buildable
- class NoGaps
- class StartingTargetTiles
- class Domain

  *Domain is the class encoding the domain of your CSP/COP.*
- class WallinDomain
- class Random

  *Random is the class coding pseudo-random generators used in GHOST.*
- class Objective

  *Objective is the class encoding objective functions of your CSP/COP.*
- class NullObjective

  *NullObjective is used when no objective functions have been given to the solver (ie, for pure satisfaction runs).*
- class WallinObjective
- class NoneObj
- class GapObj
- class BuildingObj
- class TechTreeObj
- class Solver

  *Solver is the class coding the solver itself.*
- class Building
- class Variable

  *Variable is the class encoding the variables of your CSP/COP.*

### Typedefs

- using mapFail = map< pair< int, int >, string >

### Enumerations

- enum Race { Terran, Protoss, Zerg, Unknown }

### Functions

- std::vector< std::shared_ptr < Building > > makeProtossBuildings ()
- vector::set< Constraint * > makeProtossConstraints (const std::vector< std::shared_ptr< Building > > &vec, const WallinDomain &domain)
- Building factoryTerranBuilding (const string &name, int pos=-1)
- vector< Building > makeTerranBuildings ()
- vector< shared_ptr < WallinConstraint > > makeTerranConstraints (const vector< Building > *vec, const WallinDomain *domain)
- ostream & operator<< (ostream &os, const WallinDomain &g)
- ostream & operator<< (ostream &os, const Building &b)

### Variables

- std::shared_ptr< Building > c
- std::shared_ptr< Building > f
- std::shared_ptr< Building > g1
- std::shared_ptr< Building > g2
- std::shared_ptr< Building > p1
- std::shared_ptr< Building > p2
- std::shared_ptr< Building > y1
- std::shared_ptr< Building > y2
- std::shared_ptr< Building > y3
- std::shared_ptr< Building > y4
- std::shared_ptr< Building > s
- shared_ptr< Constraint > overlap
- shared_ptr< Constraint > buildable
- shared_ptr< Constraint > noGaps
- shared_ptr< Constraint > specialTiles
- shared_ptr< Constraint > pylons

## 6.1.1 Typedef Documentation

#### 6.1.1.1 using ghost::mapFail = typedef map<pair<int, int>, string>

## 6.1.2 Enumeration Type Documentation

#### 6.1.2.1 enum ghost::Race

The enumeration type containing all StarCraft races, ie, Terran, Protoss, Zerg and Unknown.

**Enumerator**

***Terran***

***Protoss***

***Zerg***

***Unknown***

### 6.1.3 Function Documentation

6.1.3.1 **Building** ghost::factoryTerranBuilding ( const string & *name,* int *pos* = −1 )

6.1.3.2 std::vector<std::shared_ptr<**Building**> > ghost::makeProtossBuildings ( )

6.1.3.3 vector::set< **Constraint**∗ > ghost::makeProtossConstraints ( const std::vector< std::shared_ptr< Building > > & *vec,* const WallinDomain & *domain* )

6.1.3.4 vector< **Building** > ghost::makeTerranBuildings ( )

6.1.3.5 vector< shared_ptr<**WallinConstraint**> > ghost::makeTerranConstraints ( const vector< Building > ∗ *vec,* const WallinDomain ∗ *domain* )

6.1.3.6 ostream& ghost::operator<< ( ostream & *os,* const Building & *b* )

6.1.3.7 ostream& ghost::operator<< ( ostream & *os,* const WallinDomain & *g* )

### 6.1.4 Variable Documentation

6.1.4.1 shared_ptr<**Constraint**> ghost::buildable

6.1.4.2 std::shared_ptr<**Building**> ghost::c

6.1.4.3 std::shared_ptr<**Building**> ghost::f

6.1.4.4 std::shared_ptr<**Building**> ghost::g1

6.1.4.5 std::shared_ptr<**Building**> ghost::g2

6.1.4.6 shared_ptr<**Constraint**> ghost::noGaps

6.1.4.7 shared_ptr<**Constraint**> ghost::overlap

6.1.4.8 std::shared_ptr<**Building**> ghost::p1

6.1.4.9 std::shared_ptr<**Building**> ghost::p2

6.1.4.10 shared_ptr<**Constraint**> ghost::pylons

6.1.4.11 std::shared_ptr<**Building**> ghost::s

6.1.4.12 shared_ptr<**Constraint**> ghost::specialTiles

6.1.4.13 std::shared_ptr<**Building**> ghost::y1

6.1.4.14 std::shared_ptr<**Building**> ghost::y2

6.1.4.15 std::shared_ptr<**Building**> ghost::y3

6.1.4.16 std::shared_ptr<**Building**> ghost::y4

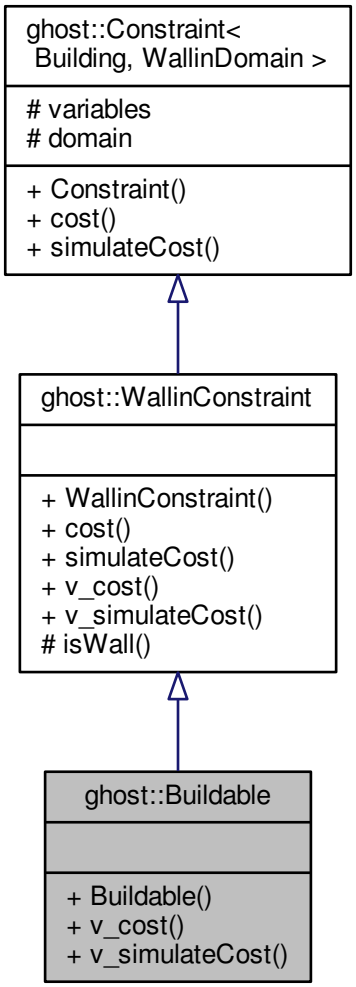
# Chapter 7

# Class Documentation

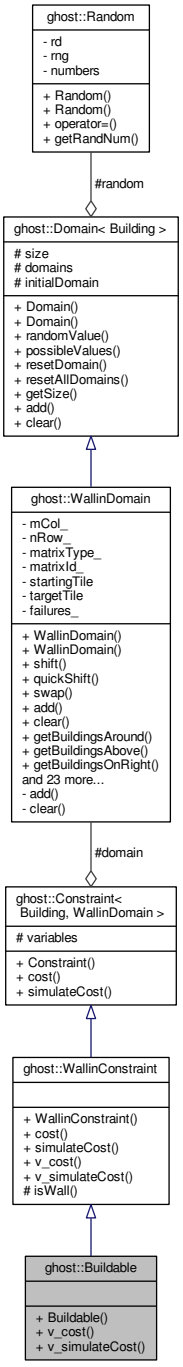
## 7.1 ghost::Buildable Class Reference

`#include <wallinConstraint.hpp>`

Inheritance diagram for ghost::Buildable:



Collaboration diagram for ghost::Buildable:




## Public Member Functions

- Buildable (const vector< Building > *, const WallinDomain *)
- double v_cost (vector< double > &) const
- vector< double > v_simulateCost (Building &, const vector< int > &, vector< vector< double > > &)

### Additional Inherited Members

### 7.1.1 Constructor & Destructor Documentation

#### 7.1.1.1 ghost::Buildable::Buildable ( const vector< Building > * *variables,* const WallinDomain * *domain* )

### 7.1.2 Member Function Documentation

#### 7.1.2.1 double ghost::Buildable::v_cost ( vector< double > & *varCost* ) const [virtual]

Implements ghost::WallinConstraint.

#### 7.1.2.2 vector< double > ghost::Buildable::v_simulateCost ( Building & *oldBuilding,* const vector< int > & *newPosition,* vector< vector< double > > & *vecVarSimCosts* ) [virtual]

Reimplemented from ghost::WallinConstraint.

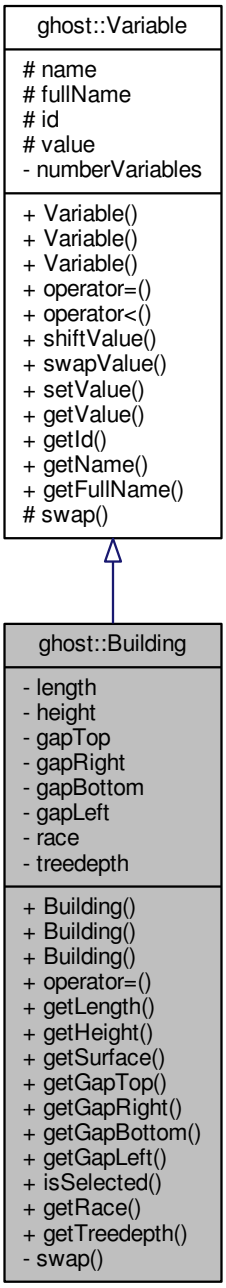
The documentation for this class was generated from the following files:

- include/constraints/wallinConstraint.hpp
- src/constraints/wallinConstraint.cpp

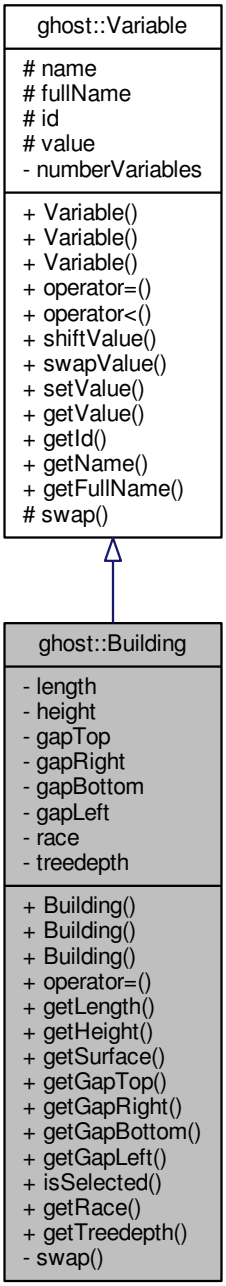
## 7.2 ghost::Building Class Reference

`#include <building.hpp>`

Inheritance diagram for ghost::Building:



Collaboration diagram for ghost::Building:




## Public Member Functions

- Building ()
- Building (int, int, int, int, int, int, Race, int, string, string, int=-1)
- Building (const Building &)
- Building & operator= (Building)
- int getLength () const

- int getHeight () const
- int getSurface () const
- int getGapTop () const
- int getGapRight () const
- int getGapBottom () const
- int getGapLeft () const
- bool isSelected () const
- string getRace () const
- int getTreedepth () const

### Private Member Functions

- void swap (Building &)

### Private Attributes

- int length
- int height
- int gapTop
- int gapRight
- int gapBottom
- int gapLeft
- Race race
- int treedepth

### Friends

- ostream & operator<< (ostream &, const Building &)

### Additional Inherited Members

## 7.2.1 Constructor & Destructor Documentation

#### 7.2.1.1 ghost::Building::Building ( )

#### 7.2.1.2 ghost::Building::Building ( int *x,* int *y,* int *top,* int *right,* int *bottom,* int *left,* Race *race,* int *treedepth,* string *name,* string *fullName,* int *position* = `-1` )

#### 7.2.1.3 ghost::Building::Building ( const Building & *other* )

## 7.2.2 Member Function Documentation

#### 7.2.2.1 int ghost::Building::getGapBottom ( ) const `[inline]`

#### 7.2.2.2 int ghost::Building::getGapLeft ( ) const `[inline]`

#### 7.2.2.3 int ghost::Building::getGapRight ( ) const `[inline]`

#### 7.2.2.4 int ghost::Building::getGapTop ( ) const `[inline]`

#### 7.2.2.5 int ghost::Building::getHeight ( ) const `[inline]`

#### 7.2.2.6 int ghost::Building::getLength ( ) const [inline]

#### 7.2.2.7 string ghost::Building::getRace ( ) const [inline]

#### 7.2.2.8 int ghost::Building::getSurface ( ) const [inline]

#### 7.2.2.9 int ghost::Building::getTreedepth ( ) const [inline]

#### 7.2.2.10 bool ghost::Building::isSelected ( ) const [inline]

#### 7.2.2.11 Building & ghost::Building::operator= ( Building *other* )

#### 7.2.2.12 void ghost::Building::swap ( Building & *other* ) [private]

### 7.2.3 Friends And Related Function Documentation

#### 7.2.3.1 ostream& operator<< ( ostream & *os,* const Building & *b* ) [friend]

### 7.2.4 Member Data Documentation

#### 7.2.4.1 int ghost::Building::gapBottom [private]

#### 7.2.4.2 int ghost::Building::gapLeft [private]

#### 7.2.4.3 int ghost::Building::gapRight [private]

#### 7.2.4.4 int ghost::Building::gapTop [private]

#### 7.2.4.5 int ghost::Building::height [private]

#### 7.2.4.6 int ghost::Building::length [private]

#### 7.2.4.7 Race ghost::Building::race [private]

#### 7.2.4.8 int ghost::Building::treedepth [private]

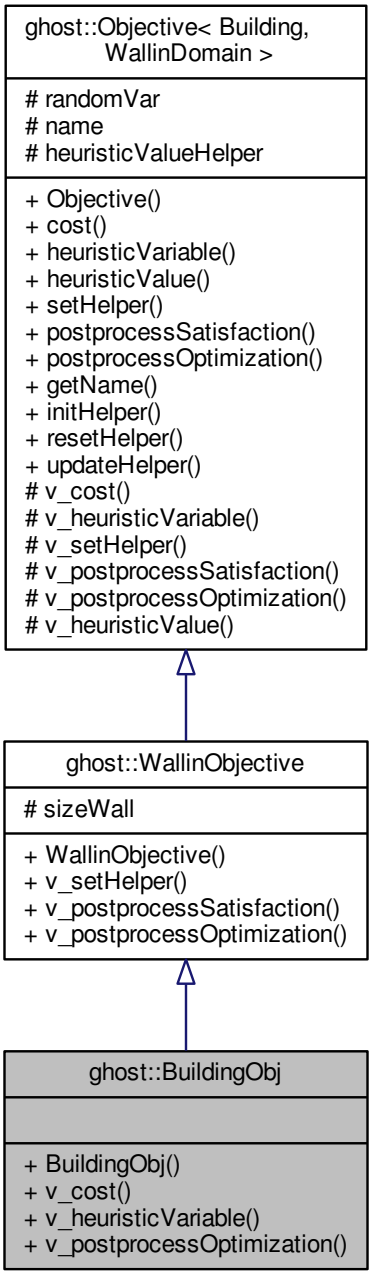
The documentation for this class was generated from the following files:

- include/variables/building.hpp
- src/variables/building.cpp

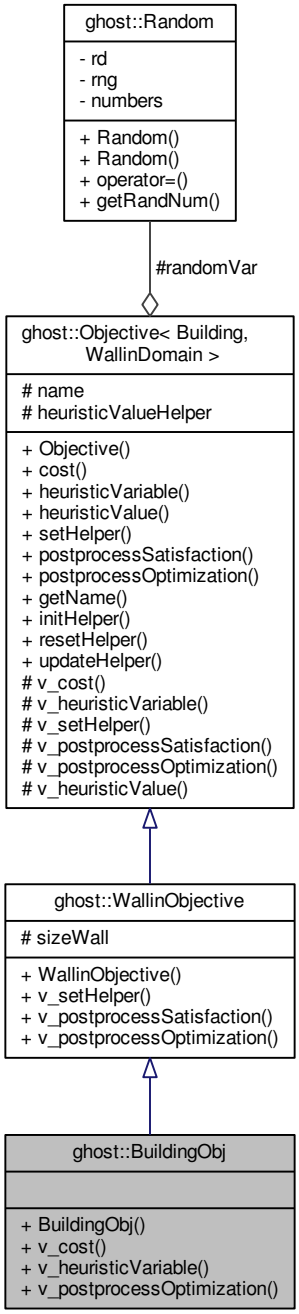
## 7.3 ghost::BuildingObj Class Reference

```
#include <wallinObjective.hpp>
```

Inheritance diagram for ghost::BuildingObj:



Collaboration diagram for ghost::BuildingObj:




## Public Member Functions

- BuildingObj ()
- double v_cost (const vector< Building > *vecVariables, const WallinDomain *domain) const

  *Pure virtual function to compute the value of the objective function on the current configuration.*
- int v_heuristicVariable (const vector< int > &vecId, const vector< Building > *vecVariables, WallinDomain *domain)

*Pure virtual function to apply the variable heuristic used by the solver.*

- double v_postprocessOptimization (vector< Building > *vecVariables, WallinDomain *domain, double &bestCost)

  *Virtual function to perform optimization post-processing.*

## Additional Inherited Members

### 7.3.1 Constructor & Destructor Documentation

#### 7.3.1.1 ghost::BuildingObj::BuildingObj ( )

### 7.3.2 Member Function Documentation

#### 7.3.2.1 double ghost::BuildingObj::v_cost ( const vector< Building > * *vecVariables*, const WallinDomain * *domain* ) const [virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implements ghost::Objective< Building, WallinDomain >.

#### 7.3.2.2 int ghost::BuildingObj::v_heuristicVariable ( const vector< int > & *vecVarId*, const vector< Building > * *vecVariables*, WallinDomain * *domain* ) [virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implements ghost::Objective< Building, WallinDomain >.

#### 7.3.2.3 double ghost::BuildingObj::v_postprocessOptimization ( vector< Building > * *vecVariables*, WallinDomain * *domain*, double & *bestCost* ) [virtual]

Virtual function to perform optimization post-processing.

This function is called by the solver after all optimization runs to apply human-knowledge optimization, allowing to improve the optimization cost.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best optimization cost found by the solver so far. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessOptimization

Reimplemented from ghost::WallinObjective.

The documentation for this class was generated from the following files:

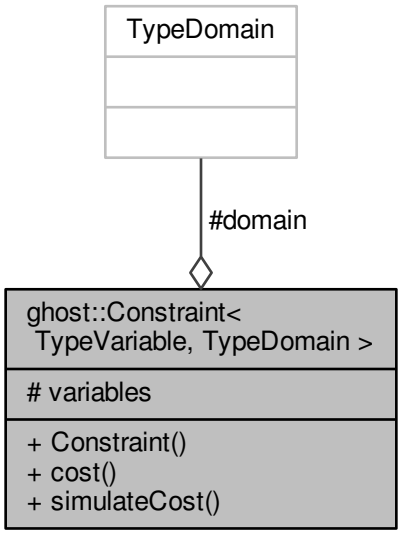
- include/objectives/wallinObjective.hpp
- src/objectives/wallinObjective.cpp

## 7.4 ghost::Constraint< TypeVariable, TypeDomain > Class Template Reference

Constraint is the class encoding constraints of your CSP/COP.

`#include <constraint.hpp>`

Collaboration diagram for ghost::Constraint< TypeVariable, TypeDomain >:




## Public Member Functions

- Constraint (const vector< TypeVariable > *variables, const TypeDomain *domain)

  *The unique Constraint constructor.*
- virtual double cost (vector< double > &varCost) const =0

  *Pure virtual function to compute the current cost of the constraint.*
- virtual vector< double > simulateCost (TypeVariable &currentVar, const vector< int > &possibleValues, vector< vector< double > > &vecVarSimCosts)=0

  *Pure virtual function to simulate the cost of the constraint on all possible values of the given variable.*

## Protected Attributes

- vector< TypeVariable > * variables

  *A pointer to the vector of variable objects of the CSP/COP.*
- TypeDomain * domain

  *A pointer to the domain object of the CSP/COP.*

## Friends

- ostream & operator<< (ostream &os, const Constraint< TypeVariable, TypeDomain > &c)

  *friend override of operator<<*

### 7.4.1 Detailed Description

**template<typename TypeVariable, typename TypeDomain>class ghost::Constraint< TypeVariable, TypeDomain >**

Constraint is the class encoding constraints of your CSP/COP.

In GHOST, many different constraint objects can be instanciate.

The Constraint class is a template class, waiting for both the type of variable and the type of domain. Thus, you must instanciate a constraint by specifying the class of your variable objects and the class of your domain object, like for instance Constraint<Variable, Domain> or Constraint<MyCustomVariable, MyCustomDomain>, if MyCustomVariable inherits from the ghost::Variable class and MyCustomDomain inherits from the ghost::Domain class.

You cannot directly use this class Constraint to encode your CSP/COP constraints, since this is an abstract class (see the list of pure virtual functions below). Thus, you must write your own constraint class inheriting from ghost::Constraint.

Pure virtual Constraint functions:

- cost
- simulateCost

**See Also**

Variable, Domain

### 7.4.2 Constructor & Destructor Documentation

#### 7.4.2.1 template<typename TypeVariable, typename TypeDomain> ghost::Constraint< TypeVariable, TypeDomain >::Constraint ( const vector< TypeVariable > * *variables*, const TypeDomain * *domain* ) [inline]

The unique Constraint constructor.

**Parameters**

| | |
|---|---|
| *variables* | A constant pointer toward the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer toward the domain object of the CSP/COP. |

### 7.4.3 Member Function Documentation

#### 7.4.3.1 template<typename TypeVariable, typename TypeDomain> virtual double ghost::Constraint< TypeVariable, TypeDomain >::cost ( vector< double > & *varCost* ) const [pure virtual]

Pure virtual function to compute the current cost of the constraint.

In cost, the parameter varCost is not given to be used by the function, but to store into varCost the projected cost of each variable. This must be computed INSIDE the cost function.

**Parameters**

| | |
|---|---|
| *varCost* | A reference to a vector of double in order to store the projected cost of each variable. |

**Returns**

A double representing the cost of the constraint on the current configuration.

**See Also**

simulateCost

Implemented in ghost::WallinConstraint.

#### 7.4.3.2 template<typename TypeVariable, typename TypeDomain> virtual vector<double> ghost::Constraint< TypeVariable, TypeDomain >::simulateCost ( TypeVariable & *currentVar,* const vector< int > & *possibleValues,* vector< vector< double > > & *vecVarSimCosts* ) [pure virtual]

Pure virtual function to simulate the cost of the constraint on all possible values of the given variable.

In cost, the parameter vecVarSimCosts is not given to be used by the function, but to store into vecVarSimCosts the projected cost of currentVar on all possible values. This must be computed INSIDE the simulateCost function.

**Parameters**

| | |
|---|---|
| *currentVar* | A reference to the variable we want to change the current value. |
| *possibleValues* | A reference to a constant vector of the possible values for currentVar. |
| *vecVarSimCosts* | A reference to the vector of vector of double in order to store the projected cost of currentVar on all possible values. |

**Returns**

The vector of the cost of the constraint for each possible value of currentVar.

**See Also**

cost

Implemented in ghost::WallinConstraint.

### 7.4.4 Friends And Related Function Documentation

#### 7.4.4.1 template<typename TypeVariable, typename TypeDomain> ostream& operator<< ( ostream & *os,* const Constraint< TypeVariable, TypeDomain > & *c* ) [friend]

friend override of operator<<

### 7.4.5 Member Data Documentation

#### 7.4.5.1 template<typename TypeVariable, typename TypeDomain> TypeDomain* ghost::Constraint< TypeVariable, TypeDomain >::domain [protected]

A pointer to the domain object of the CSP/COP.

#### 7.4.5.2 template<typename TypeVariable, typename TypeDomain> vector< TypeVariable >* ghost::Constraint< TypeVariable, TypeDomain >::variables [protected]

A pointer to the vector of variable objects of the CSP/COP.

The documentation for this class was generated from the following file:

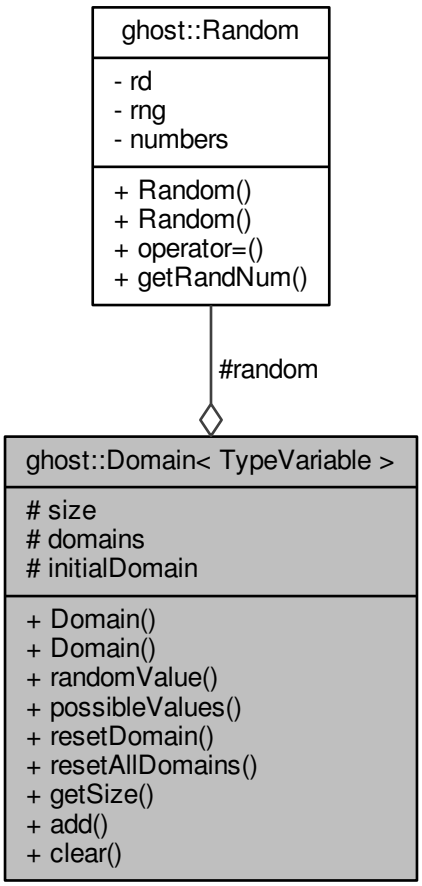
- include/constraints/constraint.hpp

## 7.5 ghost::Domain< TypeVariable > Class Template Reference

Domain is the class encoding the domain of your CSP/COP.

`#include <domain.hpp>`

Collaboration diagram for ghost::Domain< TypeVariable >:




## Public Member Functions

- Domain (int size, int numberVariables, int start=0)

  *First Domain constructor.*
- Domain (int size, int numberVariables, const vector< int > &initialDomain)

  *Second and last Domain constructor.*
- int randomValue (const TypeVariable &variable)

  *Inline function to get a random value among the possible values of a given variable.*
- vector< int > possibleValues (const TypeVariable &variable) const

  *Inline function to get the vector of the possible values of a given variable.*
- void resetDomain (const TypeVariable &variable)
- void resetAllDomains ()
- int getSize () const

  *Inline accessor to get the size of the domain.*
- void add (const TypeVariable &variable)

  *Inline function to add something into the domain.*
- void clear (const TypeVariable &variable)

  *Inline function to clear (or remove) something into the domain.*

### Protected Attributes

- int size

  *An integer to specify the size of the domain.*

- vector< vector< int > > domains

  *The vector of vector of integers, containing the domain of each variables. Thus, domains[i] is the domain of the variable i.*

- vector< int > initialDomain

  *The initial domain, created or given according to the constructor which has been called.*

- Random random

  *The random generator used by the function randomValue.*

### Friends

- ostream & operator<< (ostream &os, const Domain< TypeVariable > &domain)

  *friend override of operator<<*

## 7.5.1 Detailed Description

**template<typename TypeVariable>class ghost::Domain< TypeVariable >**

Domain is the class encoding the domain of your CSP/COP.

In GHOST, only one domain object should be instanciate. At least, the solver is only taking one domain object in parameter.

The Domain class is a template class, waiting for the type of variable. Thus, you must instanciate a domain by specifying the class of your variable objects, like for instance Domain<Variable> or Domain<MyCustomVariable>, if MyCustomVariable inherits from the ghost::Variable class.

Since in GHOST, variables can only take integer values, a domain object would contain the possible integer values for each variable of the CSP/COP.

To encode your CSP/COP domain, you can either directly use this class Domain (there are no pure virtual functions here), or inherit from it to make your own domain class.

**See Also**

Variable

## 7.5.2 Constructor & Destructor Documentation

### 7.5.2.1 template<typename TypeVariable> ghost::Domain< TypeVariable >::Domain ( int *size*, int *numberVariables*, int *start* = 0 ) [inline]

First Domain constructor.

In this constructor, the domain of each variable is built to be equals to the range [start, start + size[

**Parameters**

| | |
|---|---|
| *size* | An integer to specify the size of the domain. |
| *numberVariables* | An integer to specify the number of variables in the CSP/COP. |
| *start* | The starting value of the domain. If not given, the default value is 0. |

#### 7.5.2.2 template<typename TypeVariable> ghost::Domain< TypeVariable >::Domain ( int *size*, int *numberVariables*, const vector< int > & *initialDomain* ) [inline]

Second and last Domain constructor.

In this constructor, the domain of each variable is given as a parameter.

**Parameters**

| | |
|---|---|
| *size* | An integer to specify the size of the domain. |
| *numberVariables* | An integer to specify the number of variables in the CSP/COP. |
| *initialDomain* | A constant reference to an vector of integer, representing the inital domain for each variable. |

### 7.5.3 Member Function Documentation

#### 7.5.3.1 template<typename TypeVariable> void ghost::Domain< TypeVariable >::add ( const TypeVariable & *variable* ) [inline]

Inline function to add something into the domain.

The implementation by default does nothing. This function has been declared because it could be useful for some custom domain classes to add a value from the given variable into a custom data structure. This function is called into the solver three times:

- during a move (Solver::move), ie, when the solver assigns a new value to a given variable.
- during a reset (Solver::reset).
- just between the end of the optimization run and the beginning of the optimization post-processing, in Solver::solve.

**Parameters**

| | |
|---|---|
| *variable* | A constant reference to a variable. |

#### 7.5.3.2 template<typename TypeVariable> void ghost::Domain< TypeVariable >::clear ( const TypeVariable & *variable* ) [inline]

Inline function to clear (or remove) something into the domain.

The implementation by default does nothing. This function has been declared because it could be useful for some custom domain classes to clear/remove a value from the given variable into a custom data structure. This function is called into the solver three times:

- during a move (Solver::move), ie, when the solver assigns a new value to a given variable.
- during a reset (Solver::reset).
- just between the end of the optimization run and the beginning of the optimization post-processing, in Solver::solve.

**Parameters**

| | |
|---|---|
| *variable* | A constant reference to a variable. |

#### 7.5.3.3 template<typename TypeVariable> int ghost::Domain< TypeVariable >::getSize ( ) const [inline]

Inline accessor to get the size of the domain.

#### 7.5.3.4 template<typename TypeVariable> vector<int> ghost::Domain< TypeVariable >::possibleValues ( const TypeVariable & *variable* ) const [inline]

Inline function to get the vector of the possible values of a given variable.

**Parameters**

| | |
|---|---|
| *variable* | A constant reference to a variable. |

**Returns**

The vector of integers of all possible values of variable.

#### 7.5.3.5 template<typename TypeVariable> int ghost::Domain< TypeVariable >::randomValue ( const TypeVariable & *variable* ) [inline]

Inline function to get a random value among the possible values of a given variable.

**Parameters**

| | |
|---|---|
| *variable* | A constant reference to a variable. |

**Returns**

A random value among the possible values of variable.

**See Also**

Random

#### 7.5.3.6 template<typename TypeVariable> void ghost::Domain< TypeVariable >::resetAllDomains ( ) [inline]

Inline function to reset all variable domains to the initial domain.

All variable domains will be reset to the initial domain created or given while the domain object has been instanciated.

#### 7.5.3.7 template<typename TypeVariable> void ghost::Domain< TypeVariable >::resetDomain ( const TypeVariable & *variable* ) [inline]

Inline function to reset the domain of a given variable to the initial domain.

The domain of the given variable will be reset to the initial domain created or given while the domain object has been instanciated.

**Parameters**

| | |
|---|---|
| *variable* | A constant reference to a variable. |

### 7.5.4 Friends And Related Function Documentation

#### 7.5.4.1 template<typename TypeVariable> ostream& operator<< ( ostream & *os,* const Domain< TypeVariable > & *domain* ) [friend]

friend override of operator<<

### 7.5.5 Member Data Documentation

#### 7.5.5.1 template<typename TypeVariable> vector< vector< int > > ghost::Domain< TypeVariable >::domains [protected]

The vector of vector of integers, containing the domain of each variables. Thus, domains[i] is the domain of the variable i.

#### 7.5.5.2 template<typename TypeVariable> vector< int > ghost::Domain< TypeVariable >::initialDomain [protected]

The initial domain, created or given according to the constructor which has been called.

#### 7.5.5.3 template<typename TypeVariable> Random ghost::Domain< TypeVariable >::random [protected]

The random generator used by the function randomValue.

#### 7.5.5.4 template<typename TypeVariable> int ghost::Domain< TypeVariable >::size [protected]

An integer to specify the size of the domain.

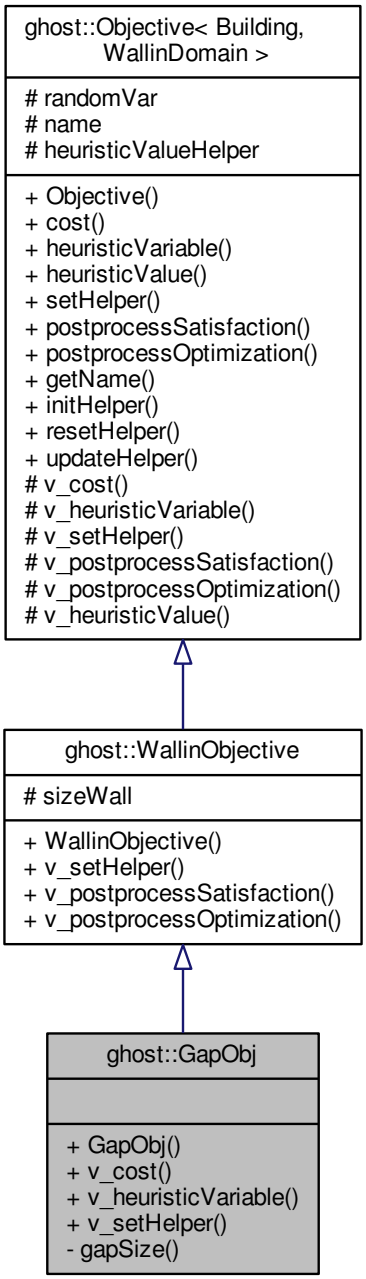
The documentation for this class was generated from the following file:

- include/domains/domain.hpp

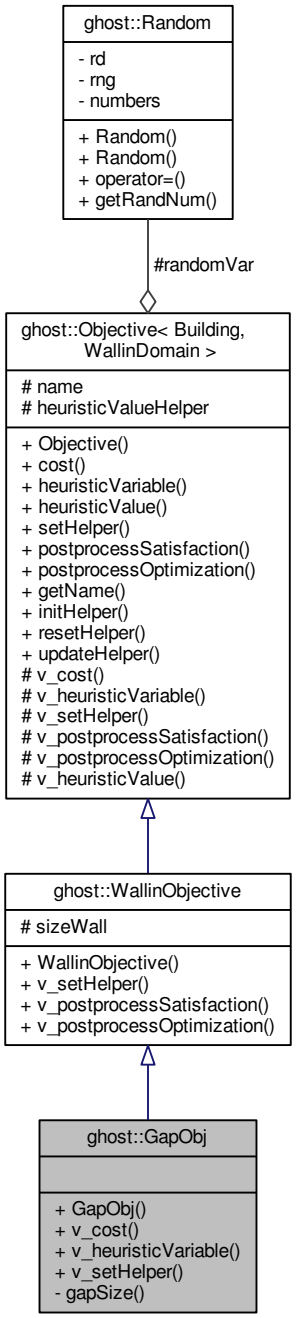
## 7.6 ghost::GapObj Class Reference

`#include <wallinObjective.hpp>`

Inheritance diagram for ghost::GapObj:



Collaboration diagram for ghost::GapObj:




## Public Member Functions

- GapObj ()
- double v_cost (const vector< Building > *vecVariables, const WallinDomain *domain) const

  *Pure virtual function to compute the value of the objective function on the current configuration.*
- int v_heuristicVariable (const vector< int > &vecId, const vector< Building > *vecVariables, WallinDomain *domain)

*Pure virtual function to apply the variable heuristic used by the solver.*

- void v_setHelper (const Building &b, const vector< Building > *vecVariables, const WallinDomain *domain)

  *Pure virtual function to set heuristicValueHelper[currentVar.getValue()].*

### Private Member Functions

- int gapSize (const Building &b, const vector< Building > *vecVariables, const WallinDomain *domain) const

### Additional Inherited Members

## 7.6.1 Constructor & Destructor Documentation

#### 7.6.1.1 ghost::GapObj::GapObj ( )

## 7.6.2 Member Function Documentation

#### 7.6.2.1 int ghost::GapObj::gapSize ( const Building & *b*, const vector< Building > * *vecVariables*, const WallinDomain * *domain* ) const [private]

#### 7.6.2.2 double ghost::GapObj::v_cost ( const vector< Building > * *vecVariables*, const WallinDomain * *domain* ) const [virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implements ghost::Objective< Building, WallinDomain >.

#### 7.6.2.3 int ghost::GapObj::v_heuristicVariable ( const vector< int > & *vecVarId*, const vector< Building > * *vecVariables*, WallinDomain * *domain* ) [virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implements ghost::Objective< Building, WallinDomain >.

#### 7.6.2.4 void ghost::GapObj::v_setHelper ( const Building & *currentVar,* const vector< Building > * *vecVariables,* const WallinDomain * *domain* ) [virtual]

Pure virtual function to set heuristicValueHelper[currentVar.getValue()].

**Parameters**

| | |
|---|---|
| *currentVar* | A constant reference to a variable object. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**See Also**

setHelper, heuristicValueHelper

Reimplemented from ghost::WallinObjective.

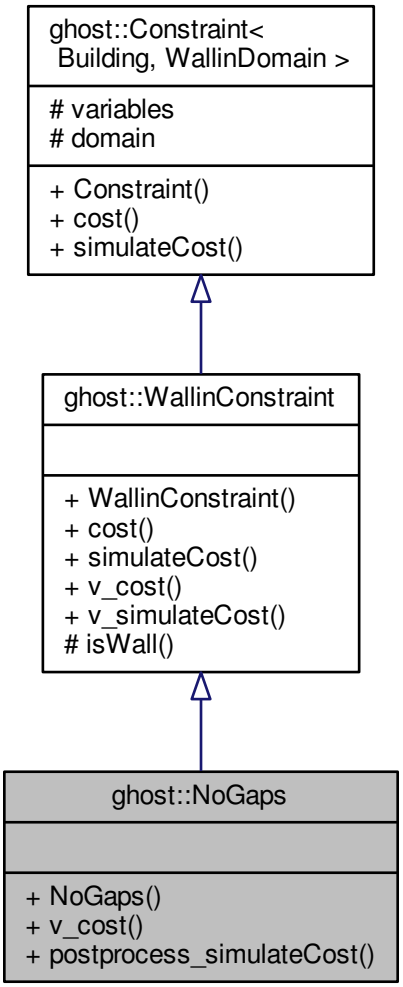
The documentation for this class was generated from the following files:

- include/objectives/wallinObjective.hpp
- src/objectives/wallinObjective.cpp

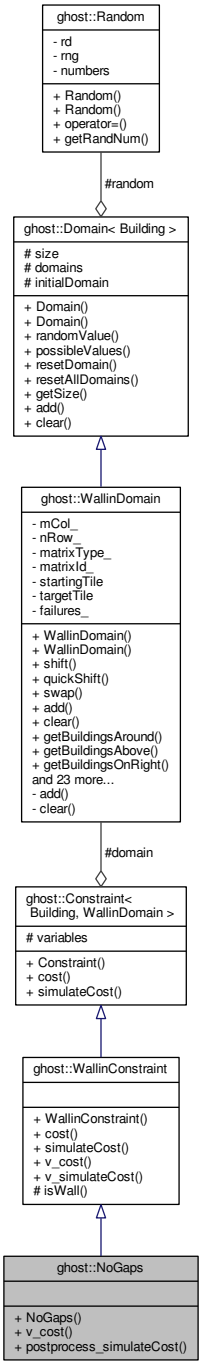
## 7.7 ghost::NoGaps Class Reference

```
#include <wallinConstraint.hpp>
```

Inheritance diagram for ghost::NoGaps:



Collaboration diagram for ghost::NoGaps:




## Public Member Functions

- NoGaps (const vector< Building > *, const WallinDomain *)
- double v_cost (vector< double > &) const
- double postprocess_simulateCost (Building &, const int, vector< double > &)

### Additional Inherited Members

### 7.7.1 Constructor & Destructor Documentation

#### 7.7.1.1 ghost::NoGaps::NoGaps ( const vector< Building > * *variables,* const WallinDomain * *domain* )

### 7.7.2 Member Function Documentation

#### 7.7.2.1 double ghost::NoGaps::postprocess_simulateCost ( Building & *oldBuilding,* const int *newPosition,* vector< double > & *varSimCost* )

#### 7.7.2.2 double ghost::NoGaps::v_cost ( vector< double > & *varCost* ) const [virtual]

Implements ghost::WallinConstraint.

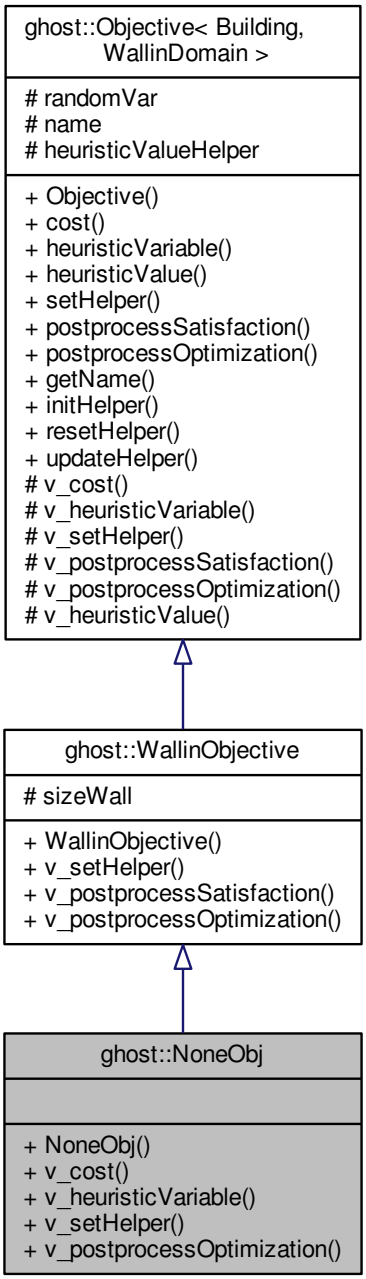
The documentation for this class was generated from the following files:

- include/constraints/wallinConstraint.hpp
- src/constraints/wallinConstraint.cpp

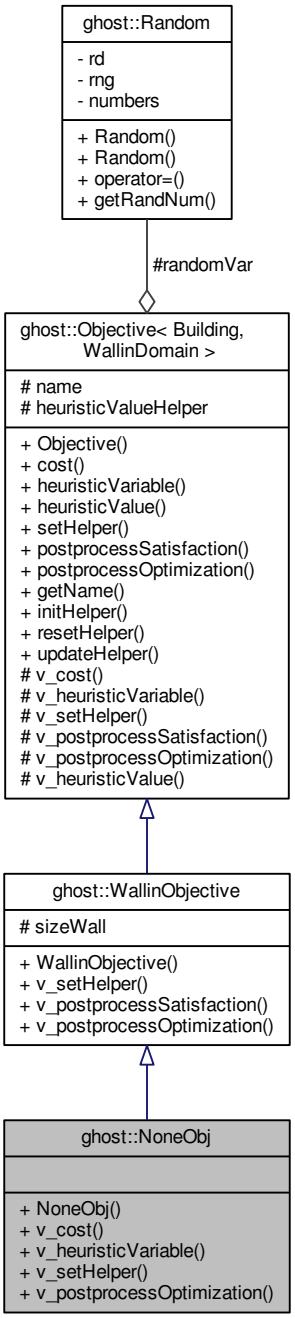
## 7.8 ghost::NoneObj Class Reference

`#include <wallinObjective.hpp>`

Inheritance diagram for ghost::NoneObj:



Collaboration diagram for ghost::NoneObj:




## Public Member Functions

- NoneObj ()
- double v_cost (const vector< Building > *vecVariables, const WallinDomain *domain) const

  *Pure virtual function to compute the value of the objective function on the current configuration.*
- int v_heuristicVariable (const vector< int > &vecId, const vector< Building > *vecVariables, WallinDomain *domain)

*Pure virtual function to apply the variable heuristic used by the solver.*

- void v_setHelper (const Building &b, const vector< Building > *vecVariables, const WallinDomain *domain)

  *Pure virtual function to set heuristicValueHelper[currentVar.getValue()].*

- double v_postprocessOptimization (vector< Building > *vecVariables, WallinDomain *domain, double &bestCost)

  *Virtual function to perform optimization post-processing.*

## Additional Inherited Members

## 7.8.1 Constructor & Destructor Documentation

### 7.8.1.1 ghost::NoneObj::NoneObj ( )

## 7.8.2 Member Function Documentation

### 7.8.2.1 double ghost::NoneObj::v_cost ( const vector< Building > * *vecVariables*, const WallinDomain * *domain* ) const [virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implements ghost::Objective< Building, WallinDomain >.

### 7.8.2.2 int ghost::NoneObj::v_heuristicVariable ( const vector< int > & *vecVarId*, const vector< Building > * *vecVariables*, WallinDomain * *domain* ) [virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implements ghost::Objective< Building, WallinDomain >.

#### 7.8.2.3 double ghost::NoneObj::v_postprocessOptimization ( vector< Building > * *vecVariables,* WallinDomain * *domain,* double & *bestCost* ) [virtual]

Virtual function to perform optimization post-processing.

This function is called by the solver after all optimization runs to apply human-knowledge optimization, allowing to improve the optimization cost.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best optimization cost found by the solver so far. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessOptimization

Reimplemented from ghost::WallinObjective.

#### 7.8.2.4 void ghost::NoneObj::v_setHelper ( const Building & *currentVar,* const vector< Building > * *vecVariables,* const WallinDomain * *domain* ) [virtual]

Pure virtual function to set heuristicValueHelper[currentVar.getValue()].

**Parameters**

| | |
|---|---|
| *currentVar* | A constant reference to a variable object. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**See Also**

setHelper, heuristicValueHelper

Reimplemented from ghost::WallinObjective.

The documentation for this class was generated from the following files:

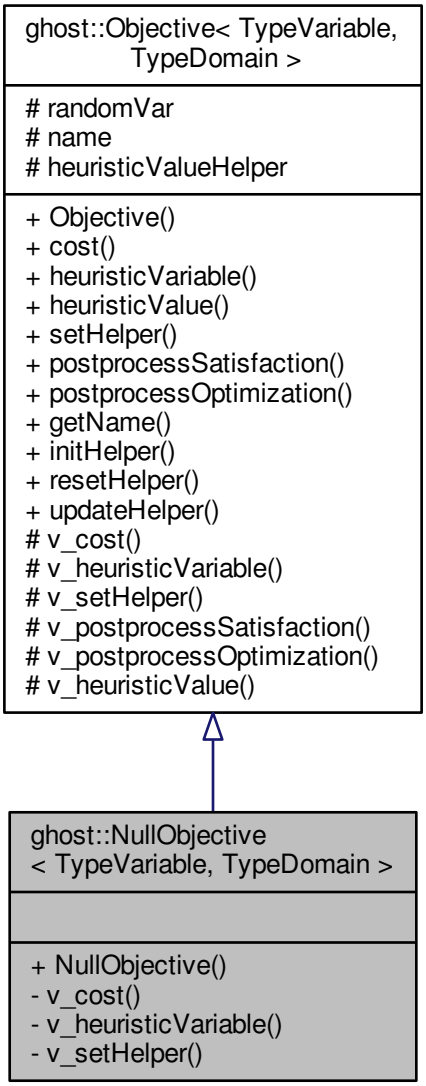
- include/objectives/wallinObjective.hpp
- src/objectives/wallinObjective.cpp

## 7.9 ghost::NullObjective< TypeVariable, TypeDomain > Class Template Reference

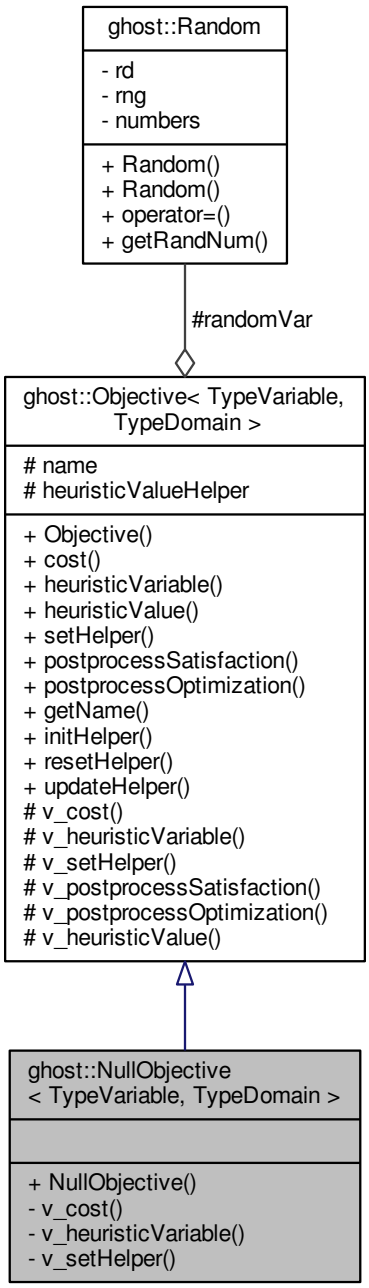
NullObjective is used when no objective functions have been given to the solver (ie, for pure satisfaction runs).

`#include <objective.hpp>`

Inheritance diagram for ghost::NullObjective< TypeVariable, TypeDomain >:



Collaboration diagram for ghost::NullObjective< TypeVariable, TypeDomain >:




## Public Member Functions

- NullObjective ()

## Private Member Functions

- virtual double v_cost (const vector< TypeVariable > *vecVariables, const TypeDomain *domain) const

*Pure virtual function to compute the value of the objective function on the current configuration.*

- virtual int v_heuristicVariable (const vector< int > &vecId, const vector< TypeVariable > *vecVariables, TypeDomain *domain)

  *Pure virtual function to apply the variable heuristic used by the solver.*

- virtual void v_setHelper (const TypeVariable &b, const vector< TypeVariable > *vecVariables, const TypeDomain *domain)

  *Pure virtual function to set heuristicValueHelper[currentVar.getValue()].*

## Additional Inherited Members

### 7.9.1 Detailed Description

**template<typename TypeVariable, typename TypeDomain>class ghost::NullObjective< TypeVariable, TypeDomain >**

NullObjective is used when no objective functions have been given to the solver (ie, for pure satisfaction runs).

### 7.9.2 Constructor & Destructor Documentation

#### 7.9.2.1 template<typename TypeVariable , typename TypeDomain > ghost::NullObjective< TypeVariable, TypeDomain >::NullObjective ( ) [inline]

### 7.9.3 Member Function Documentation

#### 7.9.3.1 template<typename TypeVariable , typename TypeDomain > virtual double ghost::NullObjective< TypeVariable, TypeDomain >::v_cost ( const vector< TypeVariable > * *vecVariables,* const TypeDomain * *domain* ) const [inline], [private], [virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implements ghost::Objective< TypeVariable, TypeDomain >.

#### 7.9.3.2 template<typename TypeVariable , typename TypeDomain > virtual int ghost::NullObjective< TypeVariable, TypeDomain >::v_heuristicVariable ( const vector< int > & *vecVarId,* const vector< TypeVariable > * *vecVariables,* TypeDomain * *domain* ) [inline], [private], [virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implements ghost::Objective< TypeVariable, TypeDomain >.

#### 7.9.3.3 template<typename TypeVariable , typename TypeDomain > virtual void ghost::NullObjective< TypeVariable, TypeDomain >::v_setHelper ( const TypeVariable & *currentVar,* const vector< TypeVariable > * *vecVariables,* const TypeDomain * *domain* ) [inline], [private], [virtual]

Pure virtual function to set heuristicValueHelper[currentVar.getValue()].

**Parameters**

| | |
|---|---|
| *currentVar* | A constant reference to a variable object. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**See Also**

setHelper, heuristicValueHelper

Implements ghost::Objective< TypeVariable, TypeDomain >.

The documentation for this class was generated from the following file:

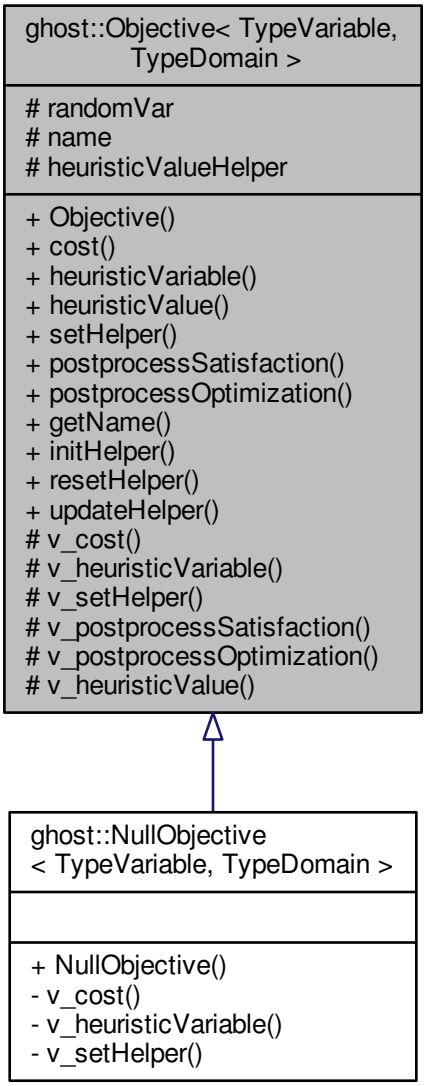
- include/objectives/objective.hpp

## 7.10 ghost::Objective< TypeVariable, TypeDomain > Class Template Reference

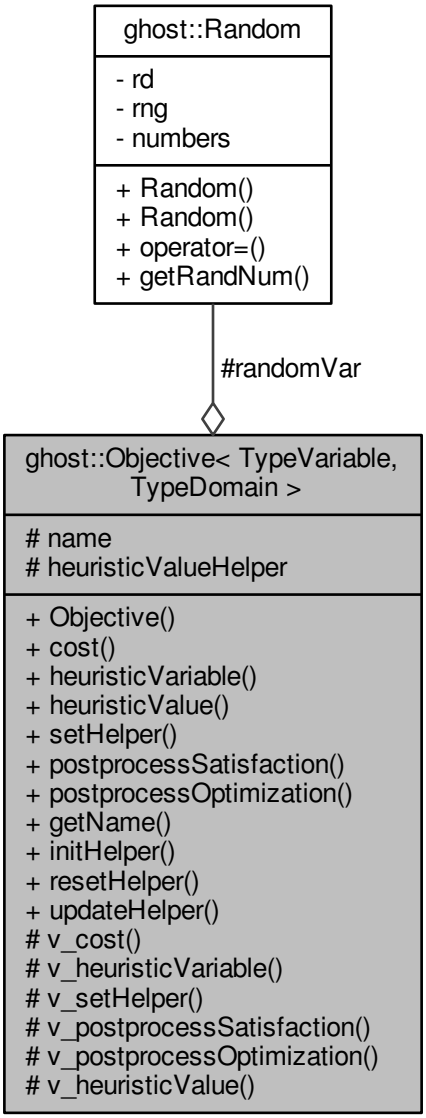
Objective is the class encoding objective functions of your CSP/COP.

`#include <objective.hpp>`

Inheritance diagram for ghost::Objective< TypeVariable, TypeDomain >:



Collaboration diagram for ghost::Objective< TypeVariable, TypeDomain >:




## Public Member Functions

- Objective (const string &name)

  *The unique Objective constructor.*
- double cost (const vector< TypeVariable > *vecVariables, const TypeDomain *domain) const
- int heuristicVariable (const vector< int > &vecVarId, const vector< TypeVariable > *vecVariables, TypeDomain *domain)
- int heuristicValue (const std::vector< double > &vecGlobalCosts, double &bestEstimatedCost, int &bestValue) const
- void setHelper (const TypeVariable &variable, const vector< TypeVariable > *vecVariables, const TypeDomain *domain)

- double postprocessSatisfaction (vector< TypeVariable > *vecVariables, TypeDomain *domain, double &bestCost, vector< int > &bestSolution)
- double postprocessOptimization (vector< TypeVariable > *vecVariables, TypeDomain *domain, double &bestCost)
- string getName ()

  *Inline accessor to get the name of the objective object.*
- void initHelper (int size)
- void resetHelper ()
- void updateHelper (TypeVariable &currentVar, const vector< int > &possibleValues, const vector< TypeVariable > *variables, TypeDomain *domain)

  *updateHelper is used to update heuristicValueHelper.*

### Protected Member Functions

- virtual double v_cost (const vector< TypeVariable > *vecVariables, const TypeDomain *domain) const =0

  *Pure virtual function to compute the value of the objective function on the current configuration.*
- virtual int v_heuristicVariable (const vector< int > &vecVarId, const vector< TypeVariable > *vecVariables, TypeDomain *domain)=0

  *Pure virtual function to apply the variable heuristic used by the solver.*
- virtual void v_setHelper (const TypeVariable &currentVar, const vector< TypeVariable > *vecVariables, const TypeDomain *domain)=0

  *Pure virtual function to set heuristicValueHelper[currentVar.getValue()].*
- virtual double v_postprocessSatisfaction (vector< TypeVariable > *vecVariables, TypeDomain *domain, double &bestCost, vector< int > &solution) const

  *Virtual function to perform satisfaction post-processing.*
- virtual double v_postprocessOptimization (vector< TypeVariable > *vecVariables, TypeDomain *domain, double &bestCost)

  *Virtual function to perform optimization post-processing.*
- virtual int v_heuristicValue (const std::vector< double > &vecGlobalCosts, double &bestEstimatedCost, int &bestValue) const

  *Virtual function to apply the value heuristic used by the solver.*

### Protected Attributes

- Random randomVar

  *The random generator used by the function heuristicValue.*
- string name

  *A string for the name of the objective object.*
- vector< double > heuristicValueHelper

  *The vector of double values implementing the value heuristic for each possible value of a given variable.*

## 7.10.1 Detailed Description

**template<typename TypeVariable, typename TypeDomain>class ghost::Objective< TypeVariable, TypeDomain >**

Objective is the class encoding objective functions of your CSP/COP.

In GHOST, many different objective objects can be instanciate.

The Objective class is a template class, waiting for both the type of variable and the type of domain. Thus, you must instanciate a constraint by specifying the class of your variable objects and the class of your domain object, like for instance Objective<Variable, Domain> or Objective<MyCustomVariable, MyCustomDomain>, if MyCustomVariable inherits from the ghost::Variable class and MyCustomDomain inherits from the ghost::Domain class.

You cannot directly use this class Objective to encode your objective functions, since this is an abstract class (see the list of pure virtual functions below). Thus, you must write your own objective class inheriting from ghost::Objective.

In this class, each virtual function follows the Non-Virtual Interface Idiom (see http://www.gotw.-ca/publications/mill18.htm). The list of all Objective pure virtual functions is below:

- v_cost
- v_heuristicVariable
- v_setHelper

**See Also**

Variable, Domain

### 7.10.2 Constructor & Destructor Documentation

#### 7.10.2.1 template<typename TypeVariable, typename TypeDomain> ghost::Objective< TypeVariable, TypeDomain >::Objective ( const string & *name* ) [inline]

The unique Objective constructor.

**Parameters**

| | |
|---|---|
| *name* | A string to give the Objective object a specific name. |

### 7.10.3 Member Function Documentation

#### 7.10.3.1 template<typename TypeVariable, typename TypeDomain> double ghost::Objective< TypeVariable, TypeDomain >::cost ( const vector< TypeVariable > * *vecVariables,* const TypeDomain * *domain* ) const [inline]

Inline function following the NVI idiom. Calling v_cost.

**See Also**

v_cost

#### 7.10.3.2 template<typename TypeVariable, typename TypeDomain> string ghost::Objective< TypeVariable, TypeDomain >::getName ( ) [inline]

Inline accessor to get the name of the objective object.

#### 7.10.3.3 template<typename TypeVariable, typename TypeDomain> int ghost::Objective< TypeVariable, TypeDomain >::heuristicValue ( const std::vector< double > & *vecGlobalCosts,* double & *bestEstimatedCost,* int & *bestValue* ) const [inline]

Inline function following the NVI idiom. Calling v_heuristicValue.

**See Also**

v_heuristicValue

#### 7.10.3.4 template<typename TypeVariable, typename TypeDomain> int ghost::Objective< TypeVariable, TypeDomain >::heuristicVariable ( const vector< int > & *vecVarId,* const vector< TypeVariable > * *vecVariables,* TypeDomain * *domain* ) [inline]

Inline function following the NVI idiom. Calling v_heuristicVariable.

**See Also**

v_heuristicVariable

#### 7.10.3.5 template<typename TypeVariable, typename TypeDomain> void ghost::Objective< TypeVariable, TypeDomain >::initHelper ( int *size* ) [inline]

Inline function to initialize heuristicValueHelper to a vector of MAX_INT values.

**See Also**

heuristicValueHelper

#### 7.10.3.6 template<typename TypeVariable, typename TypeDomain> double ghost::Objective< TypeVariable, TypeDomain >::postprocessOptimization ( vector< TypeVariable > * *vecVariables,* TypeDomain * *domain,* double & *bestCost* ) [inline]

Inline function following the NVI idiom. Calling v_postprocessOptimization.

**See Also**

v_postprocessOptimization

#### 7.10.3.7 template<typename TypeVariable, typename TypeDomain> double ghost::Objective< TypeVariable, TypeDomain >::postprocessSatisfaction ( vector< TypeVariable > * *vecVariables,* TypeDomain * *domain,* double & *bestCost,* vector< int > & *bestSolution* ) [inline]

Inline function following the NVI idiom. Calling v_postprocessSatisfaction.

**See Also**

v_postprocessSatisfaction

#### 7.10.3.8 template<typename TypeVariable, typename TypeDomain> void ghost::Objective< TypeVariable, TypeDomain >::resetHelper ( ) [inline]

Inline function to reset heuristicValueHelper with MAX_INT values.

**See Also**

heuristicValueHelper

#### 7.10.3.9 template<typename TypeVariable, typename TypeDomain> void ghost::Objective< TypeVariable, TypeDomain >::setHelper ( const TypeVariable & *variable,* const vector< TypeVariable > * *vecVariables,* const TypeDomain * *domain* ) [inline]

Inline function following the NVI idiom. Calling v_setHelper.

**See Also**

v_setHelper

#### 7.10.3.10 template<typename TypeVariable, typename TypeDomain> void ghost::Objective< TypeVariable, TypeDomain >::updateHelper ( TypeVariable & *currentVar,* const vector< int > & *possibleValues,* const vector< TypeVariable > * *variables,* TypeDomain * *domain* ) [inline]

updateHelper is used to update heuristicValueHelper.

The function updateHelper is called by Solver::solve before each call of heuristicValue.

**Parameters**

| | |
|---|---|
| *currentVar* | A reference to a variable object. |
| *possibleValues* | A constant reference to the vector of all possible values of currentVar. |
| *variables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A pointer to the domain object of the CSP/COP. |

**See Also**

heuristicValueHelper

#### 7.10.3.11 template<typename TypeVariable, typename TypeDomain> virtual double ghost::Objective< TypeVariable, TypeDomain >::v_cost ( const vector< TypeVariable > * *vecVariables,* const TypeDomain * *domain* ) const [protected], [pure virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implemented in ghost::NullObjective< TypeVariable, TypeDomain >, ghost::TechTreeObj, ghost::BuildingObj, ghost::GapObj, and ghost::NoneObj.

#### 7.10.3.12 template<typename TypeVariable, typename TypeDomain> virtual int ghost::Objective< TypeVariable, TypeDomain >::v_heuristicValue ( const std::vector< double > & *vecGlobalCosts,* double & *bestEstimatedCost,* int & *bestValue* ) const [inline], [protected], [virtual]

Virtual function to apply the value heuristic used by the solver.

This default implementation outputs the value leading to the lowest global cost. It uses heuristicValueHelper as a tiebreaker, if two or more values lead to configurations with the same lowest global cost. If two or more values cannot be tiebreak by heuristicValueHelper, one of them is randomly selected.

**Parameters**

| | |
|---|---|
| *\param* | |
| *\return* | The selected value according to the heuristic. |

**See Also**

heuristicValue, heuristicValueHelper, Random

### 7.10.3.13 template<typename TypeVariable, typename TypeDomain> virtual int ghost::Objective< TypeVariable, TypeDomain >::v_heuristicVariable ( const vector< int > & *vecVarId*, const vector< TypeVariable > * *vecVariables*, TypeDomain * *domain* ) [protected], [pure virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implemented in ghost::NullObjective< TypeVariable, TypeDomain >, ghost::TechTreeObj, ghost::BuildingObj, ghost::GapObj, and ghost::NoneObj.

### 7.10.3.14 template<typename TypeVariable, typename TypeDomain> virtual double ghost::Objective< TypeVariable, TypeDomain >::v_postprocessOptimization ( vector< TypeVariable > * *vecVariables*, TypeDomain * *domain*, double & *bestCost* ) [inline], [protected], [virtual]

Virtual function to perform optimization post-processing.

This function is called by the solver after all optimization runs to apply human-knowledge optimization, allowing to improve the optimization cost.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best optimization cost found by the solver so far. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessOptimization

Reimplemented in ghost::BuildingObj, ghost::NoneObj, and ghost::WallinObjective.

#### 7.10.3.15 template<typename TypeVariable, typename TypeDomain> virtual double ghost::Objective< TypeVariable, TypeDomain >::v_postprocessSatisfaction ( vector< TypeVariable > * *vecVariables*, TypeDomain * *domain*, double & *bestCost*, vector< int > & *solution* ) const [inline], [protected], [virtual]

Virtual function to perform satisfaction post-processing.

This function is called by the solver after a satisfaction run, if the solver was able to find a solution, to apply human-knowledge in order to "clean-up" the proposed solution.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best global cost found by the solver so far. |
| *solution* | A reference to the vector of variable values of the solution found by the solver. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessSatisfaction

Reimplemented in ghost::WallinObjective.

#### 7.10.3.16 template<typename TypeVariable, typename TypeDomain> virtual void ghost::Objective< TypeVariable, TypeDomain >::v_setHelper ( const TypeVariable & *currentVar*, const vector< TypeVariable > * *vecVariables*, const TypeDomain * *domain* ) [protected], [pure virtual]

Pure virtual function to set heuristicValueHelper[currentVar.getValue()].

**Parameters**

| | |
|---|---|
| *currentVar* | A constant reference to a variable object. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**See Also**

setHelper, heuristicValueHelper

Implemented in ghost::NullObjective< TypeVariable, TypeDomain >, ghost::GapObj, ghost::NoneObj, and ghost::WallinObjective.

### 7.10.4 Member Data Documentation

#### 7.10.4.1 template<typename TypeVariable, typename TypeDomain> vector<double> ghost::Objective< TypeVariable, TypeDomain >::heuristicValueHelper [protected]

The vector of double values implementing the value heuristic for each possible value of a given variable.

#### 7.10.4.2 template<typename TypeVariable, typename TypeDomain> string ghost::Objective< TypeVariable, TypeDomain >::name [protected]

A string for the name of the objective object.

#### 7.10.4.3 template<typename TypeVariable, typename TypeDomain> Random ghost::Objective< TypeVariable, TypeDomain >::randomVar [protected]

The random generator used by the function heuristicValue.

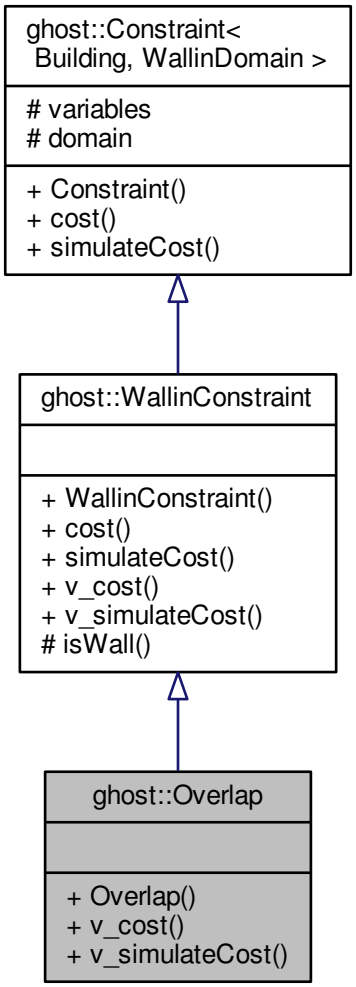
The documentation for this class was generated from the following file:

- include/objectives/objective.hpp

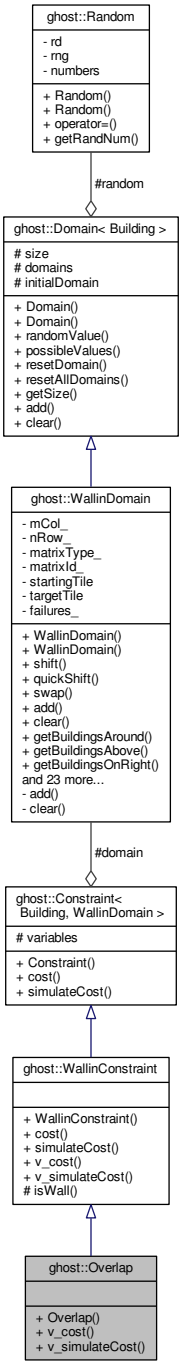
## 7.11 ghost::Overlap Class Reference

`#include <wallinConstraint.hpp>`

Inheritance diagram for ghost::Overlap:



Collaboration diagram for ghost::Overlap:




## Public Member Functions

- Overlap (const vector< Building > *, const WallinDomain *)
- double v_cost (vector< double > &) const
- vector< double > v_simulateCost (Building &, const vector< int > &, vector< vector< double > > &)

### Additional Inherited Members

### 7.11.1 Constructor & Destructor Documentation

#### 7.11.1.1 ghost::Overlap::Overlap ( const vector< Building > * *variables*, const WallinDomain * *domain* )

### 7.11.2 Member Function Documentation

#### 7.11.2.1 double ghost::Overlap::v_cost ( vector< double > & *varCost* ) const [virtual]

Implements ghost::WallinConstraint.

#### 7.11.2.2 vector< double > ghost::Overlap::v_simulateCost ( Building & *oldBuilding*, const vector< int > & *newPosition*, vector< vector< double > > & *vecVarSimCosts* ) [virtual]

Reimplemented from ghost::WallinConstraint.

The documentation for this class was generated from the following files:

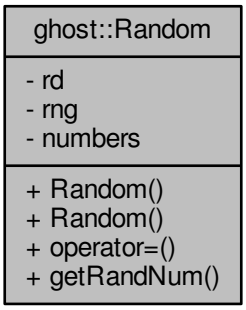
- include/constraints/wallinConstraint.hpp
- src/constraints/wallinConstraint.cpp

## 7.12 ghost::Random Class Reference

Random is the class coding pseudo-random generators used in GHOST.

`#include <random.hpp>`

Collaboration diagram for ghost::Random:




### Public Member Functions

- Random ()
- Random (const Random &other)
- Random operator= (const Random &other)
- int getRandNum (int limit)

  *Inline function to return a random value in [0, limit[.*

### Private Attributes

- std::random_device rd
- std::mt19937 rng
- std::uniform_int_distribution < int > numbers

### 7.12.1 Detailed Description

Random is the class coding pseudo-random generators used in GHOST.

Random use the C++11 Mersenne Twister (mt19937) as a pseudo-random generator.

Seeds are generated by C++11 std::random_device.

### 7.12.2 Constructor & Destructor Documentation

#### 7.12.2.1 ghost::Random::Random ( ) [inline]

#### 7.12.2.2 ghost::Random::Random ( const Random & *other* ) [inline]

### 7.12.3 Member Function Documentation

#### 7.12.3.1 int ghost::Random::getRandNum ( int *limit* ) [inline]

Inline function to return a random value in [0, limit[.

getRandNum uses a std::uniform_int_distribution<int> to compute and return a pseudo-random value from the range [0, limit[

**Parameters**

| | |
|---|---|
| *limit* | The upper bound of the range [0, limit[ from where a random value is computed. |

**Returns**

A pseudo-random value in the range [0, limit[

#### 7.12.3.2 Random ghost::Random::operator= ( const Random & *other* ) [inline]

### 7.12.4 Member Data Documentation

#### 7.12.4.1 std::uniform_int_distribution<int> ghost::Random::numbers [private]

#### 7.12.4.2 std::random_device ghost::Random::rd [private]

#### 7.12.4.3 std::mt19937 ghost::Random::rng [private]

The documentation for this class was generated from the following file:

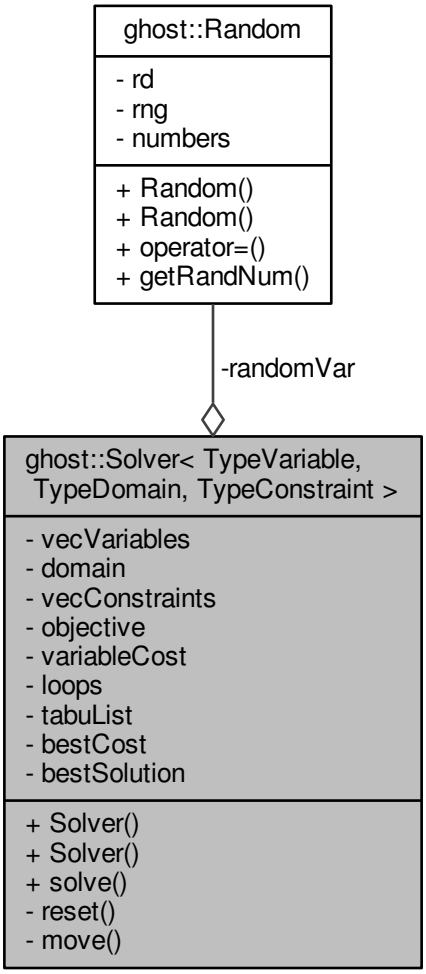
- include/misc/random.hpp

## 7.13 ghost::Solver< TypeVariable, TypeDomain, TypeConstraint > Class Template Reference

Solver is the class coding the solver itself.

`#include <solver.hpp>`

Collaboration diagram for ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >:




## Public Member Functions

- Solver (vector< TypeVariable > *vecVariables, TypeDomain *domain, const vector< shared_ptr< TypeConstraint > > &vecConstraints, const shared_ptr< Objective< TypeVariable, TypeDomain > > &obj=nullptr)

  *Solver's regular constructor.*

- Solver (vector< TypeVariable > *vecVariables, TypeDomain *domain, const vector< shared_ptr< TypeConstraint > > &vecConstraints, const shared_ptr< Objective< TypeVariable, TypeDomain > > &obj, const int loops)

  *Solver's constructor mostly used for tests.*

- double solve (double timeout)

  *Solver's main function, to solve the given CSP/COP.*

### Private Member Functions

- void reset ()
- void move (TypeVariable ∗building, int newPosition)

### Private Attributes

- vector< TypeVariable > ∗ vecVariables

  *A pointer to the vector of variable objects of the CSP/COP.*
- TypeDomain ∗ domain

  *A pointer to the domain object of the CSP/COP.*
- vector< shared_ptr < TypeConstraint > > vecConstraints

  *The vector of (shared pointers of) constraints of the CSP/COP.*
- shared_ptr< Objective < TypeVariable, TypeDomain > > objective

  *The shared pointer of the objective function.*
- vector< double > variableCost

  *The vector of projected costs on each varaible.*
- int loops

  *The number of times we reiterate the satisfaction loop inside Solver::solve.*
- vector< int > tabuList

  *The tabu list, frozing each used variable for TABU iterations.*
- Random randomVar

  *The random generator used by the solver.*
- double bestCost

  *The (satisfaction or optimization) cost of the best solution.*
- vector< int > bestSolution

  *The best solution found by the solver.*

## 7.13.1 Detailed Description

**template<typename TypeVariable, typename TypeDomain, typename TypeConstraint>class ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >**

Solver is the class coding the solver itself.

You just need to instanciate one Solver object.

The Solver class is a template class, waiting for both the type of variable, the type of domain and the type of constraint. Thus, you must instanciate a solver by specifying the class of your variable objects, the class of your domain object and the class of your constraint objects, like for instance Solver<Variable, Domain, Constraint> or Solver<MyCustomVariable, MyCustomDomain, MyCustomConstraint>, if MyCustomVariable inherits from the ghost::Variable class, MyCustomDomain inherits from the ghost::Domain class and MyCustomConstraint inherits from the ghost::Constraint class.

Solver's constructor also need a shared pointer of an Objective object (nullptr by default). The reason why Objective is not a template parameter of Solver but a pointer is to allow a dynamic modification of the objective function.

**See Also**

Variable, Domain, Constraint, Objective

### 7.13.2 Constructor & Destructor Documentation

#### 7.13.2.1 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::Solver ( vector< TypeVariable > * *vecVariables,* TypeDomain * *domain,* const vector< shared_ptr< TypeConstraint > > & *vecConstraints,* const shared_ptr< Objective< TypeVariable, TypeDomain > > & *obj* = `nullptr` ) [inline]

Solver's regular constructor.

The solver is calling Solver(vecVariables, domain, vecConstraints, obj, 0)

**Parameters**

| | |
|---|---|
| *vecVariables* | A pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A pointer to the domain object of the CSP/COP. |
| *vecConstraints* | A constant reference to the vector of shared pointers of Constraint |
| *obj* | A reference to the shared pointer of an Objective object. Default value is nullptr. |

#### 7.13.2.2 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::Solver ( vector< TypeVariable > * *vecVariables,* TypeDomain * *domain,* const vector< shared_ptr< TypeConstraint > > & *vecConstraints,* const shared_ptr< Objective< TypeVariable, TypeDomain > > & *obj,* const int *loops* ) [inline]

Solver's constructor mostly used for tests.

Like the regular constructor, but take also a loops parameter to repeat loops times to satisfaction loop inside Solver::solve. This is mostly used for tests and runtime performance measures.

**Parameters**

| | |
|---|---|
| *vecVariables* | A pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A pointer to the domain object of the CSP/COP. |
| *vecConstraints* | A constant reference to the vector of shared pointers of Constraint |
| *obj* | A reference to the shared pointer of an Objective object. Default value is nullptr. |
| *loops* | The number of times we want to repeat the satisaction loop inside Solver::solve. |

### 7.13.3 Member Function Documentation

#### 7.13.3.1 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > void ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::move ( TypeVariable * *building,* int *newPosition* ) [inline], [private]

Solver's function to make a local move, ie, to assign a given value to a given variable

#### 7.13.3.2 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > void ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::reset ( ) [inline], [private]

Solver's function to perform a reset, ie, to restart the search process from a fresh and randomly generated configuration.

#### 7.13.3.3 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > double ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::solve ( double *timeout* ) [inline]

Solver's main function, to solve the given CSP/COP.

**Parameters**

| | |
|---|---|
| *timeout* | The satisfaction run timeout in milliseconds |

**Returns**

The satisfaction or optimization cost of the best solution, respectively is the Solver object has been instanciate with a null Objective (pure satisfaction run) or an non-null Objective (optimization run).

### 7.13.4 Member Data Documentation

#### 7.13.4.1 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > double ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::bestCost [private]

The (satisfaction or optimization) cost of the best solution.

#### 7.13.4.2 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > vector<int> ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::bestSolution [private]

The best solution found by the solver.

#### 7.13.4.3 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > TypeDomain* ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::domain [private]

A pointer to the domain object of the CSP/COP.

#### 7.13.4.4 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > int ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::loops [private]

The number of times we reiterate the satisfaction loop inside Solver::solve.

#### 7.13.4.5 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > shared_ptr< Objective<TypeVariable, TypeDomain> > ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::objective [private]

The shared pointer of the objective function.

#### 7.13.4.6 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > Random ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::randomVar [private]

The random generator used by the solver.

#### 7.13.4.7 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > vector<int> ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::tabuList [private]

The tabu list, frozing each used variable for TABU iterations.

#### 7.13.4.8 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > vector<double> ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::variableCost [private]

The vector of projected costs on each varaible.

#### 7.13.4.9 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > vector< shared_ptr<TypeConstraint> > ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::vecConstraints [private]

The vector of (shared pointers of) constraints of the CSP/COP.

#### 7.13.4.10 template<typename TypeVariable , typename TypeDomain , typename TypeConstraint > vector< TypeVariable >* ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >::vecVariables [private]

A pointer to the vector of variable objects of the CSP/COP.

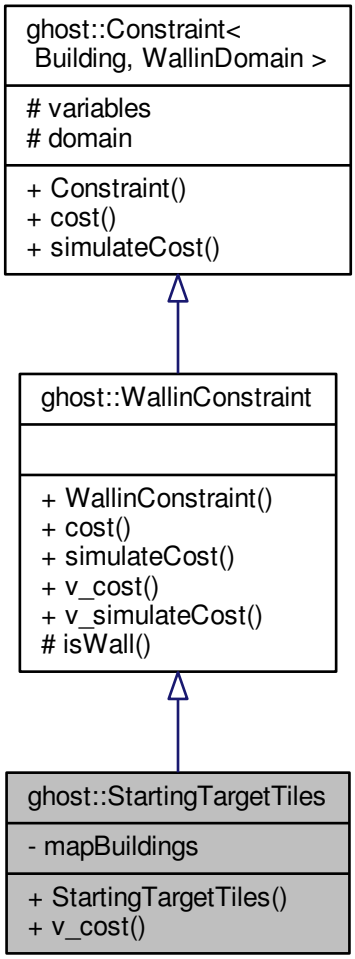
The documentation for this class was generated from the following file:

- include/solver.hpp

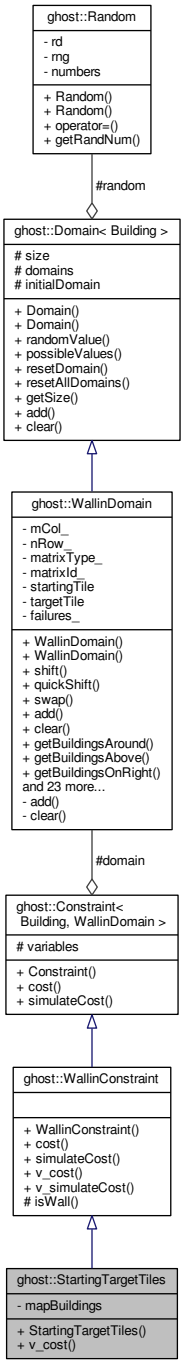
## 7.14 ghost::StartingTargetTiles Class Reference

```
#include <wallinConstraint.hpp>
```

Inheritance diagram for ghost::StartingTargetTiles:



Collaboration diagram for ghost::StartingTargetTiles:




## Public Member Functions

- StartingTargetTiles (const vector< Building > *, const WallinDomain *)
- double v_cost (vector< double > &) const

### Private Attributes

- map< int, Building * > mapBuildings

### Additional Inherited Members

## 7.14.1 Constructor & Destructor Documentation

#### 7.14.1.1 ghost::StartingTargetTiles::StartingTargetTiles ( const vector< Building > * *variables*, const WallinDomain * *domain* )

## 7.14.2 Member Function Documentation

#### 7.14.2.1 double ghost::StartingTargetTiles::v_cost ( vector< double > & *varCost* ) const [virtual]

Implements ghost::WallinConstraint.

## 7.14.3 Member Data Documentation

#### 7.14.3.1 map<int, Building*> ghost::StartingTargetTiles::mapBuildings [private]

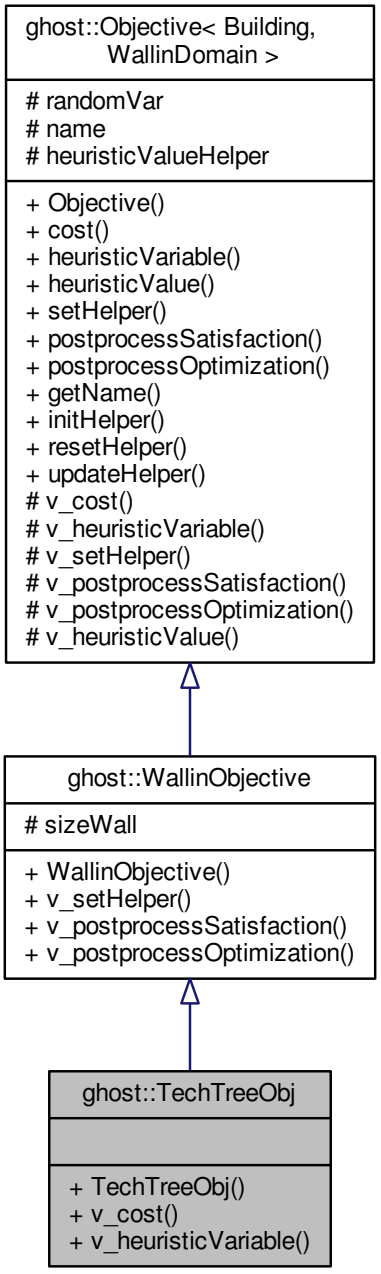
The documentation for this class was generated from the following files:

- include/constraints/wallinConstraint.hpp
- src/constraints/wallinConstraint.cpp

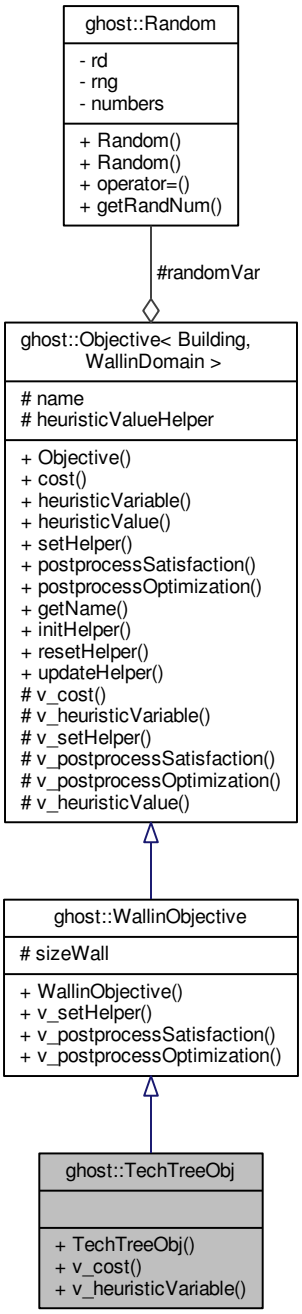
# 7.15 ghost::TechTreeObj Class Reference

```
#include <wallinObjective.hpp>
```

Inheritance diagram for ghost::TechTreeObj:



Collaboration diagram for ghost::TechTreeObj:




## Public Member Functions

- TechTreeObj ()
- double v_cost (const vector< Building > *vecVariables, const WallinDomain *domain) const

  *Pure virtual function to compute the value of the objective function on the current configuration.*
- int v_heuristicVariable (const vector< int > &vecId, const vector< Building > *vecVariables, WallinDomain *domain)

*Pure virtual function to apply the variable heuristic used by the solver.*

## Additional Inherited Members

## 7.15.1 Constructor & Destructor Documentation

### 7.15.1.1 ghost::TechTreeObj::TechTreeObj ( )

## 7.15.2 Member Function Documentation

### 7.15.2.1 double ghost::TechTreeObj::v_cost ( const vector< Building > * *vecVariables*, const WallinDomain * *domain* ) const [virtual]

Pure virtual function to compute the value of the objective function on the current configuration.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The value of the objective function on the current configuration.

**See Also**

cost

Implements ghost::Objective< Building, WallinDomain >.

### 7.15.2.2 int ghost::TechTreeObj::v_heuristicVariable ( const vector< int > & *vecVarId*, const vector< Building > * *vecVariables*, WallinDomain * *domain* ) [virtual]

Pure virtual function to apply the variable heuristic used by the solver.

**Parameters**

| | |
|---|---|
| *vecVarId* | A constant reference to the vector of variable ID objects of the CSP/COP. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**Returns**

The ID of the selected variable according to the heuristic.

**See Also**

heuristicVariable

Implements ghost::Objective< Building, WallinDomain >.

The documentation for this class was generated from the following files:

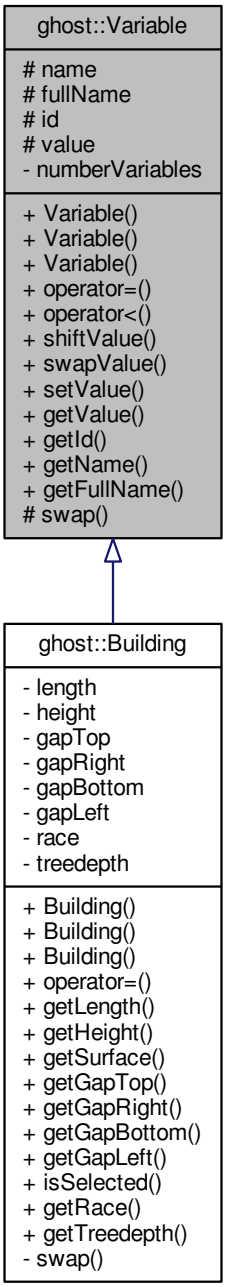
- include/objectives/wallinObjective.hpp
- src/objectives/wallinObjective.cpp

## 7.16 ghost::Variable Class Reference

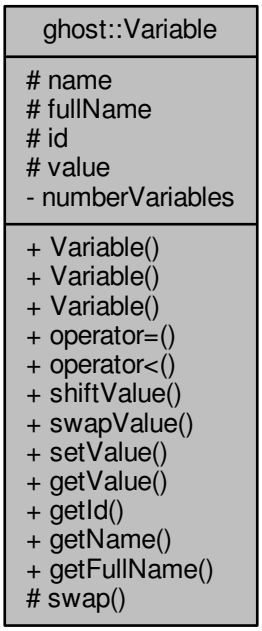
Variable is the class encoding the variables of your CSP/COP.

`#include <variable.hpp>`

Inheritance diagram for ghost::Variable:



Collaboration diagram for ghost::Variable:




## Public Member Functions

- Variable ()

  *Empty Variable constructor by default, doing nothing.*
- Variable (string name, string fullName, int value=-1)

  *The regular Variable constructor.*
- Variable (const Variable &other)

  *Variable's copy constructor, designed to NOT increment numberVariables.*
- Variable & operator= (Variable other)

  *Variable's copy assignment operator, designed to NOT increment numberVariables.*
- bool operator< (const Variable &other) const
- void shiftValue ()
- void swapValue (Variable &other)

  *Inline function to swap the value of two objects.*
- void setValue (int v)

  *Inline mutator to set the object's value.*
- int getValue () const

  *Inline accessor to get the object's value.*
- int getId () const

  *Inline accessor to get the object's id.*
- string getName () const

  *Inline accessor to get the object's name.*
- string getFullName () const

  *Inline accessor to get the object's full name.*

### Protected Member Functions

- void swap (Variable &other)

  *Inline function used for the copy-and-swap idiom.*

### Protected Attributes

- string name

  *A string to give a shorten name to the variable (for instance, "B").*
- string fullName

  *A string to give a full name to the variable (for instance, "Barracks").*
- int id

  *An integer to stamp the object. Its value must be unique among all Variable objects.*
- int value

  *The value of the variable. Must be an integer (it can take negative values).*

### Static Private Attributes

- static int numberVariables = 0

  *A static integer to make sure the object's id is unique. Incremented by calling the regular constructor.*

### Friends

- std::ostream & operator<< (std::ostream &os, const Variable &v)

  *friend override of operator<<*

## 7.16.1 Detailed Description

Variable is the class encoding the variables of your CSP/COP.

In GHOST, all variable objects must be instanciate from the same concrete class. Be careful to model your CSP/COP in order to use one kind of variable only.

To encode your CSP/COP variables, you can either directly use this class Variable (there are no pure virtual functions here), or inherit from it to make your own variable class.

## 7.16.2 Constructor & Destructor Documentation

### 7.16.2.1 ghost::Variable::Variable ( ) [inline]

Empty Variable constructor by default, doing nothing.

### 7.16.2.2 ghost::Variable::Variable ( string *name*, string *fullName*, int *value* = −1 ) [inline]

The regular Variable constructor.

When this constructor is called, the class variable numberVariables is automatically incremented.

**Parameters**

| | |
|---|---|
| *name* | A string to give a shorten name to the variable (for instance, "B"). |
| *fullName* | A string to give a full name to the variable (for instance, "Barracks"). |
| *value* | The initial value of the variable, -1 by default. |

**See Also**

numberVariables

#### 7.16.2.3 ghost::Variable::Variable ( const Variable & *other* ) [inline]

Variable's copy constructor, designed to NOT increment numberVariables.

**Parameters**

| | |
|---|---|
| *other* | A reference to a Variable object. |

**See Also**

numberVariables

### 7.16.3 Member Function Documentation

#### 7.16.3.1 string ghost::Variable::getFullName ( ) const [inline]

Inline accessor to get the object's full name.

#### 7.16.3.2 int ghost::Variable::getId ( ) const [inline]

Inline accessor to get the object's id.

#### 7.16.3.3 string ghost::Variable::getName ( ) const [inline]

Inline accessor to get the object's name.

#### 7.16.3.4 int ghost::Variable::getValue ( ) const [inline]

Inline accessor to get the object's value.

#### 7.16.3.5 bool ghost::Variable::operator< ( const Variable & *other* ) const [inline]

Inline function to compare (less-than operator) two Variable objects. In this class, operator< is implemented to compare two Variable objects regarding their id.

#### 7.16.3.6 Variable& ghost::Variable::operator= ( Variable *other* ) [inline]

Variable's copy assignment operator, designed to NOT increment numberVariables.

The copy-and-swap idiom is applyed here.

**Parameters**

| | |
|---|---|
| *other* | A Variable object. |

**See Also**

numberVariables

#### 7.16.3.7 void ghost::Variable::setValue ( int *v* ) [inline]

Inline mutator to set the object's value.

In this class, setValue is a mere value = v

**Parameters**

| | |
|---|---|
| *v* | An integer representing the new value to set. |

#### 7.16.3.8 void ghost::Variable::shiftValue ( ) [inline]

Inline function to shift the object value. In this class, shiftValue is implemented to increment the value (++value).

#### 7.16.3.9 void ghost::Variable::swap ( Variable & *other* ) [inline], [protected]

Inline function used for the copy-and-swap idiom.

**Parameters**

| | |
|---|---|
| *other* | A reference to a Variable object. |

#### 7.16.3.10 void ghost::Variable::swapValue ( Variable & *other* ) [inline]

Inline function to swap the value of two objects.

In this class, swapValue calls std::swap between this->value and other.value.

**Parameters**

| | |
|---|---|
| *other* | A reference to a Variable object. |

### 7.16.4 Friends And Related Function Documentation

#### 7.16.4.1 std::ostream& operator<< ( std::ostream & *os,* const Variable & *v* ) [friend]

friend override of operator<<

### 7.16.5 Member Data Documentation

#### 7.16.5.1 string ghost::Variable::fullName [protected]

A string to give a full name to the variable (for instance, "Barracks").

#### 7.16.5.2 int ghost::Variable::id [protected]

An integer to stamp the object. Its value must be unique among all Variable objects.

#### 7.16.5.3 string ghost::Variable::name [protected]

A string to give a shorten name to the variable (for instance, "B").

#### 7.16.5.4 int ghost::Variable::numberVariables = 0 [static], [private]

A static integer to make sure the object's id is unique. Incremented by calling the regular constructor.

#### 7.16.5.5 int ghost::Variable::value [protected]

The value of the variable. Must be an integer (it can take negative values).

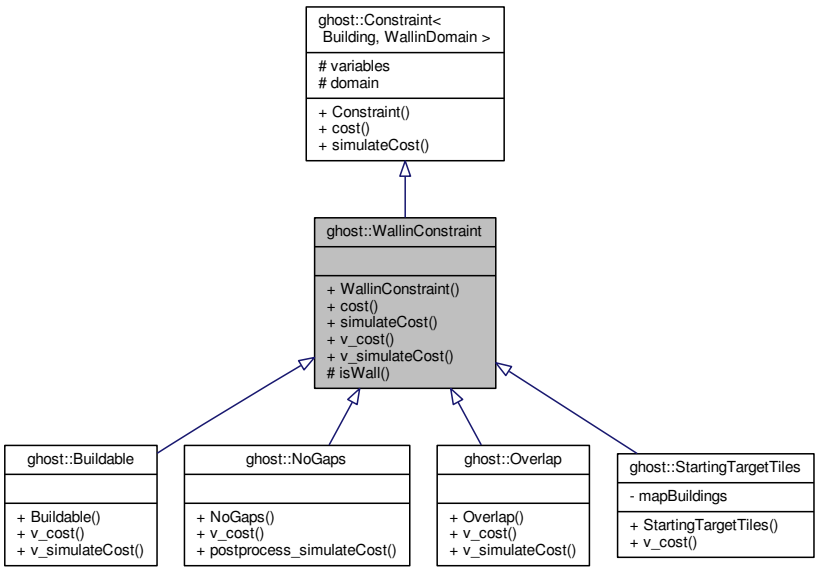
The documentation for this class was generated from the following files:

- include/variables/variable.hpp
- src/variables/variable.cpp

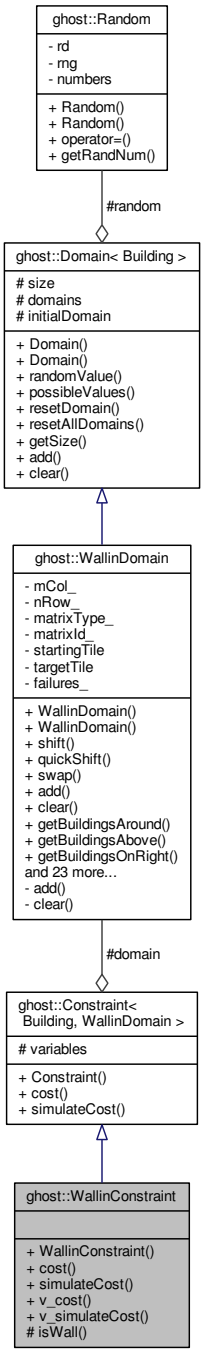
## 7.17 ghost::WallinConstraint Class Reference

`#include <wallinConstraint.hpp>`

Inheritance diagram for ghost::WallinConstraint:



Collaboration diagram for ghost::WallinConstraint:




## Public Member Functions

- WallinConstraint (const vector< Building > *, const WallinDomain *)
- double cost (vector< double > &varCost) const

  *Pure virtual function to compute the current cost of the constraint.*
- vector< double > simulateCost (Building &oldBuilding, const vector< int > &newPosition, vector< vector< double > > &vecVarSimCosts)

*Pure virtual function to simulate the cost of the constraint on all possible values of the given variable.*

- virtual double v_cost (vector< double > &) const =0
- virtual vector< double > v_simulateCost (Building &oldBuilding, const vector< int > &newPosition, vector< vector< double > > &vecVarSimCosts)

## Protected Member Functions

- bool isWall () const

## Additional Inherited Members

### 7.17.1 Constructor & Destructor Documentation

#### 7.17.1.1 ghost::WallinConstraint::WallinConstraint ( const vector< Building > * *variables,* const WallinDomain * *domain* )

### 7.17.2 Member Function Documentation

#### 7.17.2.1 double ghost::WallinConstraint::cost ( vector< double > & *varCost* ) const [inline], [virtual]

Pure virtual function to compute the current cost of the constraint.

In cost, the parameter varCost is not given to be used by the function, but to store into varCost the projected cost of each variable. This must be computed INSIDE the cost function.

**Parameters**

| | |
|---|---|
| *varCost* | A reference to a vector of double in order to store the projected cost of each variable. |

**Returns**

A double representing the cost of the constraint on the current configuration.

**See Also**

simulateCost

Implements ghost::Constraint< Building, WallinDomain >.

#### 7.17.2.2 bool ghost::WallinConstraint::isWall ( ) const [protected]

#### 7.17.2.3 vector<double> ghost::WallinConstraint::simulateCost ( Building & *currentVar,* const vector< int > & *possibleValues,* vector< vector< double > > & *vecVarSimCosts* ) [inline], [virtual]

Pure virtual function to simulate the cost of the constraint on all possible values of the given variable.

In cost, the parameter vecVarSimCosts is not given to be used by the function, but to store into vecVarSimCosts the projected cost of currentVar on all possible values. This must be computed INSIDE the simulateCost function.

**Parameters**

| | |
|---|---|
| *currentVar* | A reference to the variable we want to change the current value. |
| *possibleValues* | A reference to a constant vector of the possible values for currentVar. |

| | |
|---|---|
| *vecVarSimCosts* | A reference to the vector of vector of double in order to store the projected cost of currentVar on all possible values. |

**Returns**

The vector of the cost of the constraint for each possible value of currentVar.

**See Also**

cost

Implements ghost::Constraint< Building, WallinDomain >.

#### 7.17.2.4 virtual double ghost::WallinConstraint::v_cost ( vector< double > & ) const [pure virtual]

Implemented in ghost::StartingTargetTiles, ghost::NoGaps, ghost::Buildable, and ghost::Overlap.

#### 7.17.2.5 virtual vector<double> ghost::WallinConstraint::v_simulateCost ( Building & *oldBuilding,* const vector< int > & *newPosition,* vector< vector< double > > & *vecVarSimCosts* ) [inline], [virtual]

Reimplemented in ghost::Buildable, and ghost::Overlap.

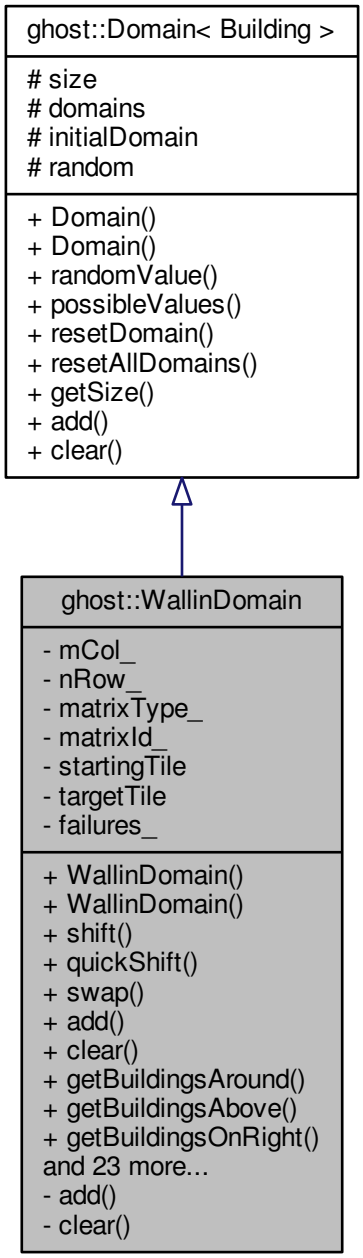
The documentation for this class was generated from the following files:

- include/constraints/wallinConstraint.hpp
- src/constraints/wallinConstraint.cpp

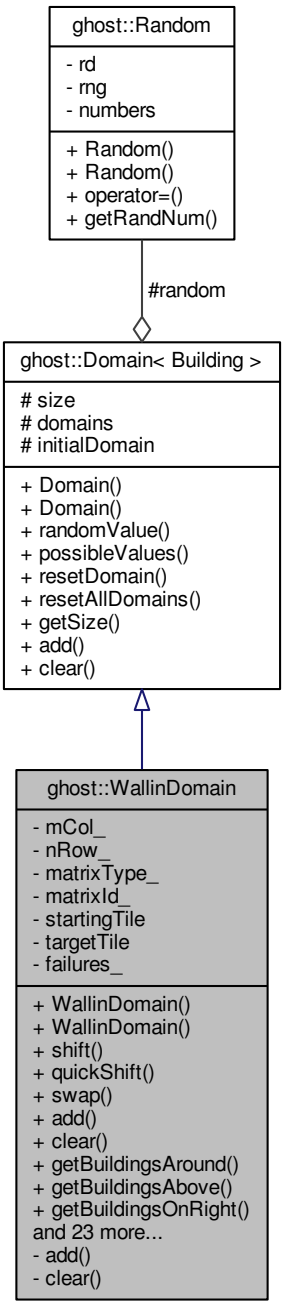
## 7.18 ghost::WallinDomain Class Reference

`#include <wallinDomain.hpp>`

Inheritance diagram for ghost::WallinDomain:



Collaboration diagram for ghost::WallinDomain:




## Public Member Functions

- WallinDomain (int, int, int, int, int, int, int)
- WallinDomain (int, int, const vector< pair< int, int > > &, const vector< Building > *, int, int, int, int)
- pair< int, int > shift (Building &)
- void quickShift (Building &)
- void swap (Building &, Building &)

- void add (const Building &)
- void clear (const Building &)
- set< Building > getBuildingsAround (const Building &, const vector< Building > *) const
- set< Building > getBuildingsAbove (const Building &, const vector< Building > *) const
- set< Building > getBuildingsOnRight (const Building &, const vector< Building > *) const
- set< Building > getBuildingsBelow (const Building &, const vector< Building > *) const
- set< Building > getBuildingsOnLeft (const Building &, const vector< Building > *) const
- int distanceTo (int source, int target) const
- int distanceToTarget (int source) const
- int distanceTo (int, pair< int, int >) const
- void unbuildable (int row, int col)
- void unbuildable (vector< pair< int, int > >)
- set< int > buildingsAt (int row, int col) const
- set< int > buildingsAt (pair< int, int > p) const
- set< int > buildingsAt (int p) const
- pair< int, int > getStartingTile () const
- pair< int, int > getTargetTile () const
- int getNberRows () const
- int getNberCols () const
- bool hasFailure () const
- mapFail failures () const
- pair< int, int > lin2mat (int p) const
- int mat2lin (int row, int col) const
- int mat2lin (pair< int, int > p) const
- bool isStartingOrTargetTile (int) const
- bool isNeightborOfSTTBuildings (const Building &, vector< Building >) const
- int countAround (const Building &, const vector< Building > *) const
- vector< int > possiblePos (const Building &) const

### Private Member Functions

- void add (int, int, string, int)
- void clear (int, int, string, int)

### Private Attributes

- int mCol_
- int nRow_
- vector< vector< string > > matrixType_
- vector< vector< set< int > > > matrixId_
- pair< int, int > startingTile
- pair< int, int > targetTile
- mapFail failures_

### Friends

- ostream & operator<< (ostream &, const WallinDomain &)

### Additional Inherited Members

## 7.18.1 Constructor & Destructor Documentation

7.18.1.1 ghost::WallinDomain::WallinDomain ( int *col,* int *row,* int *nbVar,* int *sRow,* int *sCol,* int *tRow,* int *tCol* )

7.18.1.2 ghost::WallinDomain::WallinDomain ( int *col,* int *row,* const vector< pair< int, int > > & *unbuildables,* const vector< **Building** > * *variables,* int *sRow,* int *sCol,* int *tRow,* int *tCol* )

## 7.18.2 Member Function Documentation

7.18.2.1 void ghost::WallinDomain::add ( const **Building** & *building* )

7.18.2.2 void ghost::WallinDomain::add ( int *row,* int *col,* string *b_short,* int *b_id* ) [private]

7.18.2.3 set<int> ghost::WallinDomain::buildingsAt ( int *row,* int *col* ) const [inline]

7.18.2.4 set<int> ghost::WallinDomain::buildingsAt ( pair< int, int > *p* ) const [inline]

7.18.2.5 set<int> ghost::WallinDomain::buildingsAt ( int *p* ) const [inline]

7.18.2.6 void ghost::WallinDomain::clear ( const **Building** & *building* )

7.18.2.7 void ghost::WallinDomain::clear ( int *row,* int *col,* string *b_short,* int *b_id* ) [private]

7.18.2.8 int ghost::WallinDomain::countAround ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.9 int ghost::WallinDomain::distanceTo ( int *source,* int *target* ) const [inline]

7.18.2.10 int ghost::WallinDomain::distanceTo ( int *source,* pair< int, int > *target* ) const

7.18.2.11 int ghost::WallinDomain::distanceToTarget ( int *source* ) const [inline]

7.18.2.12 **mapFail** ghost::WallinDomain::failures ( ) const [inline]

7.18.2.13 set< **Building** > ghost::WallinDomain::getBuildingsAbove ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.14 set< **Building** > ghost::WallinDomain::getBuildingsAround ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.15 set< **Building** > ghost::WallinDomain::getBuildingsBelow ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.16 set< **Building** > ghost::WallinDomain::getBuildingsOnLeft ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.17 set< **Building** > ghost::WallinDomain::getBuildingsOnRight ( const **Building** & *b,* const vector< **Building** > * *variables* ) const

7.18.2.18 int ghost::WallinDomain::getNberCols ( ) const [inline]

7.18.2.19 int ghost::WallinDomain::getNberRows ( ) const [inline]

7.18.2.20 pair<int, int> ghost::WallinDomain::getStartingTile ( ) const [inline]

7.18.2.21 pair<int, int> ghost::WallinDomain::getTargetTile ( ) const [inline]

7.18.2.22 bool ghost::WallinDomain::hasFailure ( ) const [inline]

7.18.2.23 bool ghost::WallinDomain::isNeightborOfSTTBuildings ( const Building & *building,* vector< Building > *others* ) const

7.18.2.24 bool ghost::WallinDomain::isStartingOrTargetTile ( int *id* ) const

7.18.2.25 pair<int, int> ghost::WallinDomain::lin2mat ( int *p* ) const [inline]

7.18.2.26 int ghost::WallinDomain::mat2lin ( int *row,* int *col* ) const [inline]

7.18.2.27 int ghost::WallinDomain::mat2lin ( pair< int, int > *p* ) const [inline]

7.18.2.28 vector< int > ghost::WallinDomain::possiblePos ( const Building & *b* ) const

7.18.2.29 void ghost::WallinDomain::quickShift ( Building & *building* )

7.18.2.30 pair< int, int > ghost::WallinDomain::shift ( Building & *building* )

7.18.2.31 void ghost::WallinDomain::swap ( Building & *first,* Building & *second* )

7.18.2.32 void ghost::WallinDomain::unbuildable ( int *row,* int *col* ) [inline]

7.18.2.33 void ghost::WallinDomain::unbuildable ( vector< pair< int, int > > *unbuildables* )

### 7.18.3 Friends And Related Function Documentation

7.18.3.1 ostream& operator<< ( ostream & *os,* const WallinDomain & *g* ) [friend]

### 7.18.4 Member Data Documentation

7.18.4.1 mapFail ghost::WallinDomain::failures_ [private]

7.18.4.2 vector< vector< set<int> > > ghost::WallinDomain::matrixId_ [private]

7.18.4.3 vector< vector<string> > ghost::WallinDomain::matrixType_ [private]

7.18.4.4 int ghost::WallinDomain::mCol_ [private]

7.18.4.5 int ghost::WallinDomain::nRow_ [private]

7.18.4.6 pair<int, int> ghost::WallinDomain::startingTile [private]

7.18.4.7 pair<int, int> ghost::WallinDomain::targetTile [private]

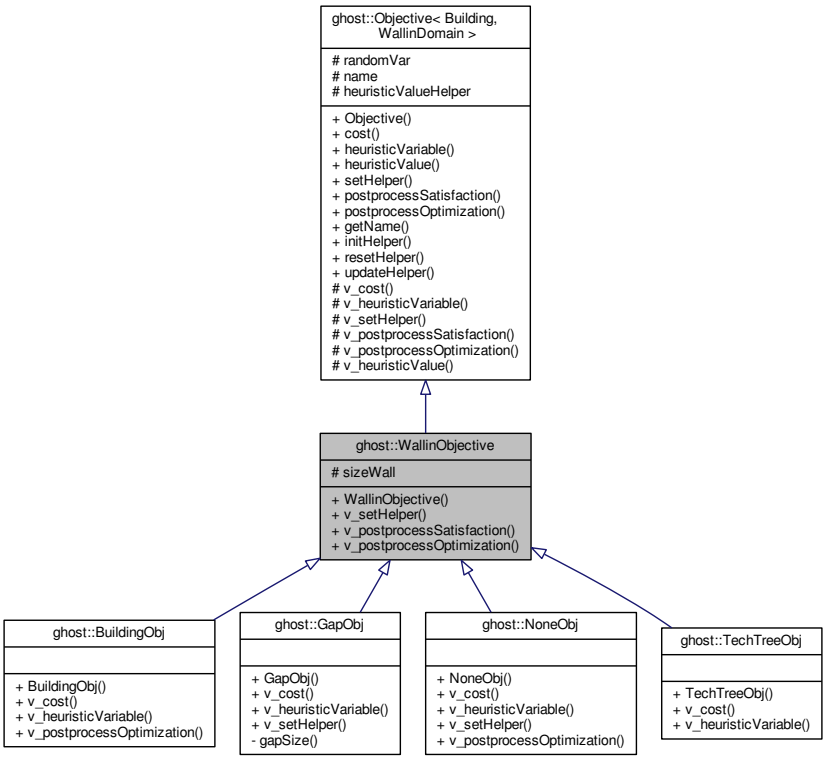
The documentation for this class was generated from the following files:

- include/domains/wallinDomain.hpp
- src/domains/wallinDomain.cpp

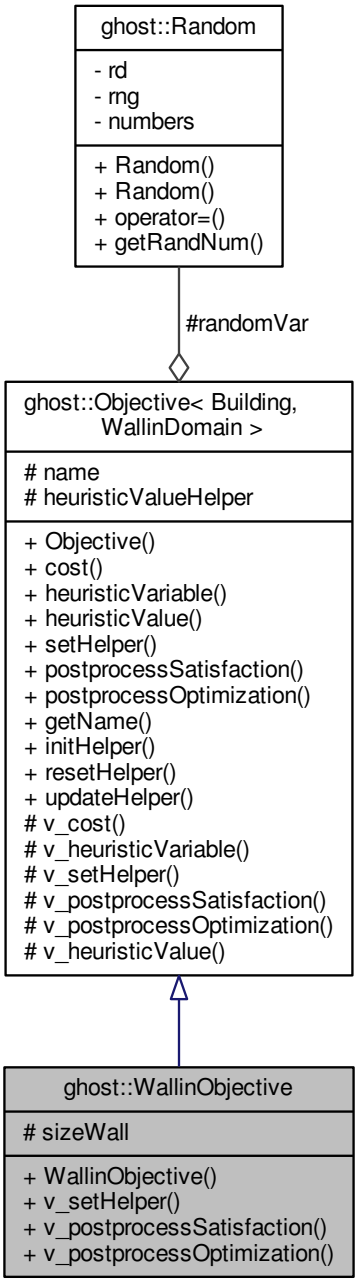
## 7.19 ghost::WallinObjective Class Reference

`#include <wallinObjective.hpp>`

Inheritance diagram for ghost::WallinObjective:



Collaboration diagram for ghost::WallinObjective:




## Public Member Functions

- WallinObjective (const string &)
- virtual void v_setHelper (const Building &b, const vector< Building > *vecVariables, const WallinDomain *domain)

  *Pure virtual function to set heuristicValueHelper[currentVar.getValue()].*
- virtual double v_postprocessSatisfaction (vector< Building > *vecVariables, WallinDomain *domain, double &bestCost, vector< int > &bestSolution) const

*Virtual function to perform satisfaction post-processing.*

- virtual double v_postprocessOptimization (vector< Building > *vecBuildings, WallinDomain *domain, double &bestCost)

  *Virtual function to perform optimization post-processing.*

### Static Protected Attributes

- static int sizeWall = numeric_limits<int>::max()

### Additional Inherited Members

## 7.19.1 Constructor & Destructor Documentation

#### 7.19.1.1 ghost::WallinObjective::WallinObjective ( const string & *name* )

## 7.19.2 Member Function Documentation

#### 7.19.2.1 double ghost::WallinObjective::v_postprocessOptimization ( vector< Building > * *vecVariables,* WallinDomain * *domain,* double & *bestCost* ) [virtual]

Virtual function to perform optimization post-processing.

This function is called by the solver after all optimization runs to apply human-knowledge optimization, allowing to improve the optimization cost.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best optimization cost found by the solver so far. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessOptimization

Reimplemented from ghost::Objective< Building, WallinDomain >.

Reimplemented in ghost::BuildingObj, and ghost::NoneObj.

#### 7.19.2.2 double ghost::WallinObjective::v_postprocessSatisfaction ( vector< Building > * *vecVariables,* WallinDomain * *domain,* double & *bestCost,* vector< int > & *solution* ) const [virtual]

Virtual function to perform satisfaction post-processing.

This function is called by the solver after a satisfaction run, if the solver was able to find a solution, to apply human-knowledge in order to "clean-up" the proposed solution.

This implementation by default does nothing.

**Parameters**

| | |
|---|---|
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |
| *bestCost* | A reference the double representing the best global cost found by the solver so far. |
| *solution* | A reference to the vector of variable values of the solution found by the solver. |

**Returns**

The function runtime in milliseconds.

**See Also**

postprocessSatisfaction

Reimplemented from ghost::Objective< Building, WallinDomain >.

#### 7.19.2.3 void ghost::WallinObjective::v_setHelper ( const Building & *currentVar,* const vector< Building > * *vecVariables,* const WallinDomain * *domain* ) [virtual]

Pure virtual function to set heuristicValueHelper[currentVar.getValue()].

**Parameters**

| | |
|---|---|
| *currentVar* | A constant reference to a variable object. |
| *vecVariables* | A constant pointer to the vector of variable objects of the CSP/COP. |
| *domain* | A constant pointer to the domain object of the CSP/COP. |

**See Also**

setHelper, heuristicValueHelper

Implements ghost::Objective< Building, WallinDomain >.

Reimplemented in ghost::GapObj, and ghost::NoneObj.

### 7.19.3 Member Data Documentation

#### 7.19.3.1 int ghost::WallinObjective::sizeWall = numeric_limits<int>::max() [static], [protected]

The documentation for this class was generated from the following files:

- include/objectives/wallinObjective.hpp
- src/objectives/wallinObjective.cpp

# Chapter 8

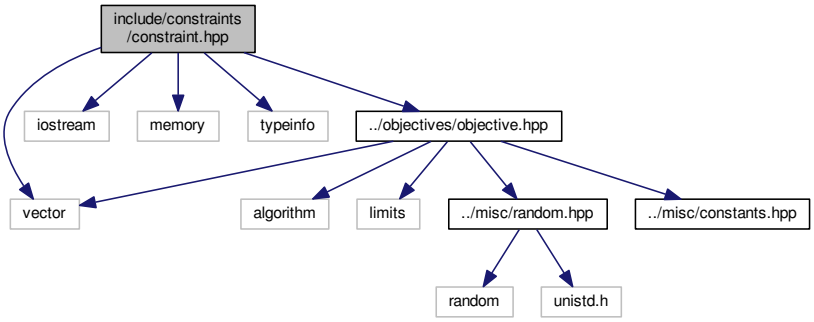
# File Documentation

## 8.1 doc/mainpage.dox File Reference

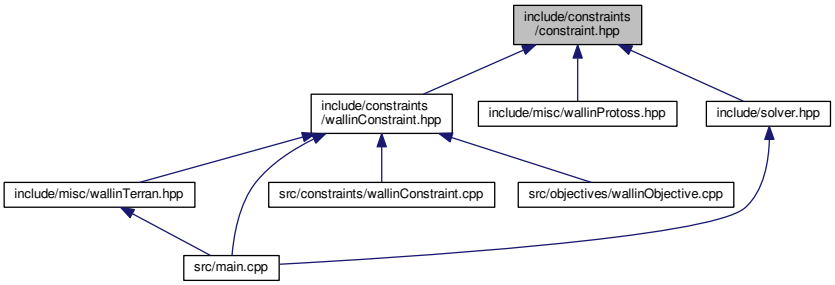
## 8.2 include/constraints/constraint.hpp File Reference

```
#include <vector>
#include <iostream>
#include <memory>
#include <typeinfo>
#include "../objectives/objective.hpp"
```

Include dependency graph for constraint.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Constraint< TypeVariable, TypeDomain >

  *Constraint is the class encoding constraints of your CSP/COP.*

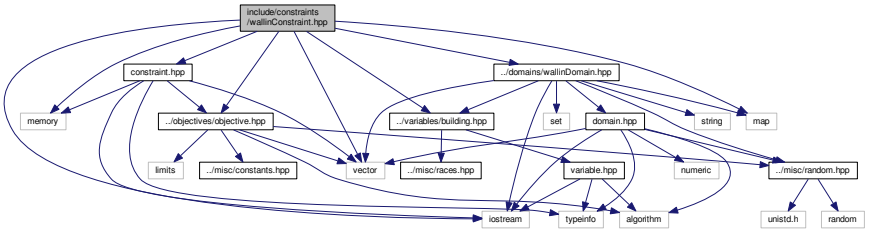
### Namespaces

- ghost

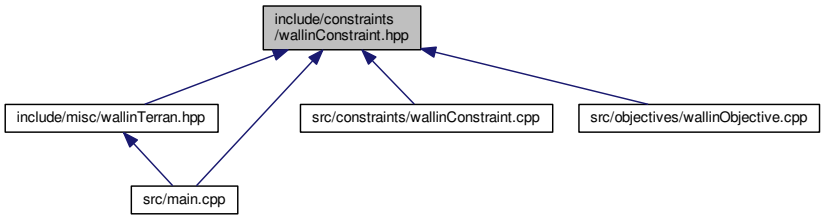
## 8.3 include/constraints/wallinConstraint.hpp File Reference

```
#include <vector>
#include <iostream>
#include <memory>
#include <map>
#include "constraint.hpp"
#include "../variables/building.hpp"
#include "../domains/wallinDomain.hpp"
#include "../objectives/objective.hpp"
```

Include dependency graph for wallinConstraint.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::WallinConstraint
- class ghost::Overlap
- class ghost::Buildable
- class ghost::NoGaps
- class ghost::StartingTargetTiles

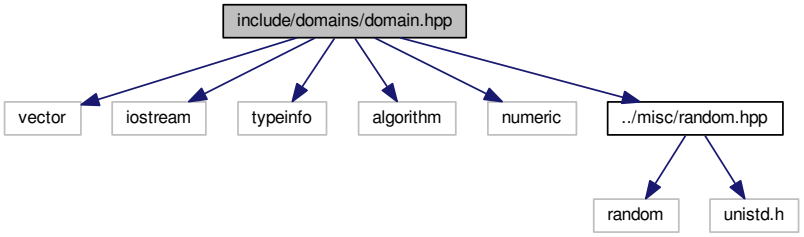
### Namespaces

- ghost

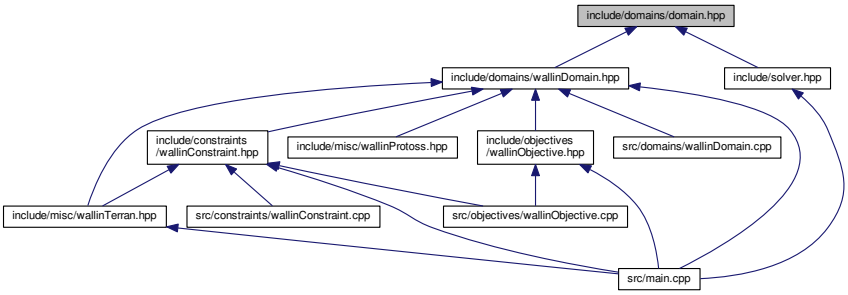
## 8.4 include/domains/domain.hpp File Reference

```
#include <vector>
#include <iostream>
#include <typeinfo>
#include <algorithm>
#include <numeric>
#include "../misc/random.hpp"
```

Include dependency graph for domain.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Domain< TypeVariable >

  *Domain is the class encoding the domain of your CSP/COP.*

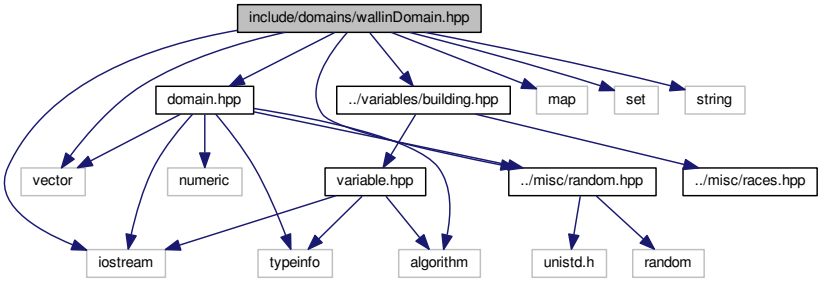
### Namespaces

- ghost

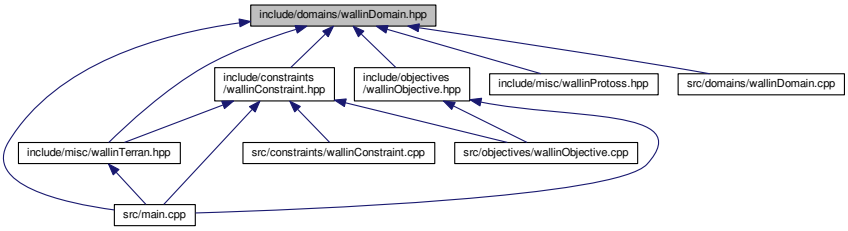
## 8.5 include/domains/wallinDomain.hpp File Reference

```
#include <vector>
#include <map>
#include <set>
#include <string>
#include <iostream>
#include "domain.hpp"
#include "../variables/building.hpp"
#include "../misc/random.hpp"
```

Include dependency graph for wallinDomain.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::WallinDomain

### Namespaces

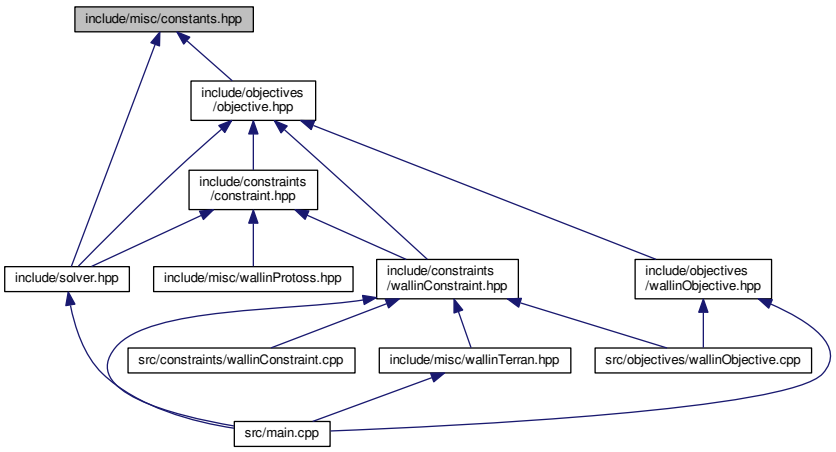
- ghost

### Typedefs

- using ghost::mapFail = map< pair< int, int >, string >

## 8.6 include/misc/constants.hpp File Reference

This graph shows which files directly or indirectly include this file:




### Variables

- constexpr int TABU = 5
- constexpr int OPT_TIME = 150

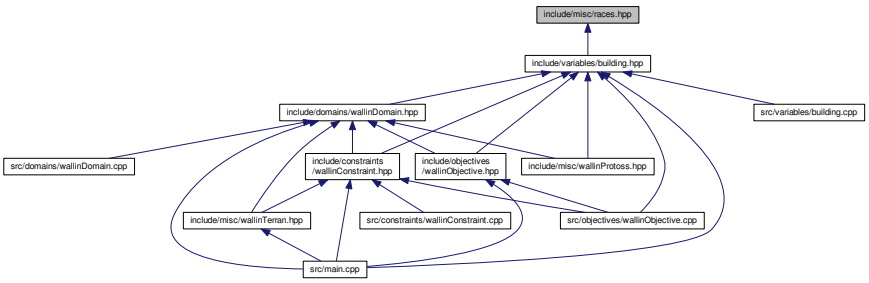
### 8.6.1 Variable Documentation

#### 8.6.1.1 constexpr int OPT_TIME = 150

#### 8.6.1.2 constexpr int TABU = 5

## 8.7 include/misc/races.hpp File Reference

This graph shows which files directly or indirectly include this file:




### Namespaces

- ghost

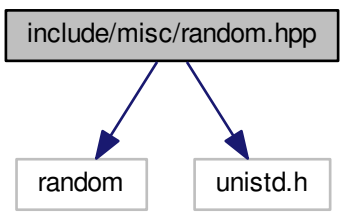
### Enumerations

- enum ghost::Race { ghost::Terran, ghost::Protoss, ghost::Zerg, ghost::Unknown }

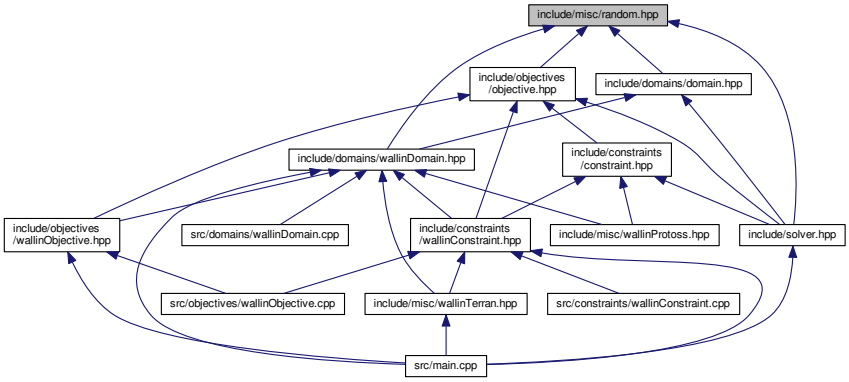
## 8.8 include/misc/random.hpp File Reference

```
#include <random>
#include <unistd.h>
```

Include dependency graph for random.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Random

  *Random is the class coding pseudo-random generators used in GHOST.*

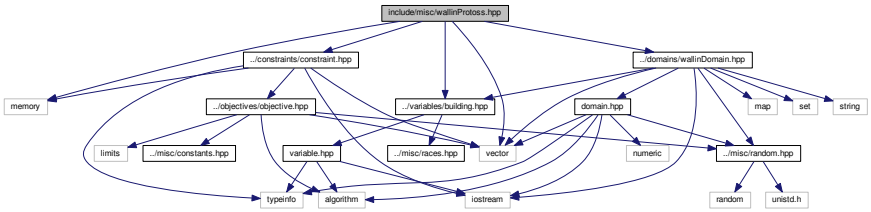
### Namespaces

- ghost

## 8.9 include/misc/wallinProtoss.hpp File Reference

```
#include <vector>
#include <memory>
#include "../variables/building.hpp"
#include "../constraints/constraint.hpp"
#include "../domains/wallinDomain.hpp"
```

Include dependency graph for wallinProtoss.hpp:




### Namespaces

- ghost

### Functions

- std::vector< std::shared_ptr < Building > > ghost::makeProtossBuildings ()
- vector::set< Constraint * > ghost::makeProtossConstraints (const std::vector< std::shared_ptr< Building > > &vec, const WallinDomain &domain)

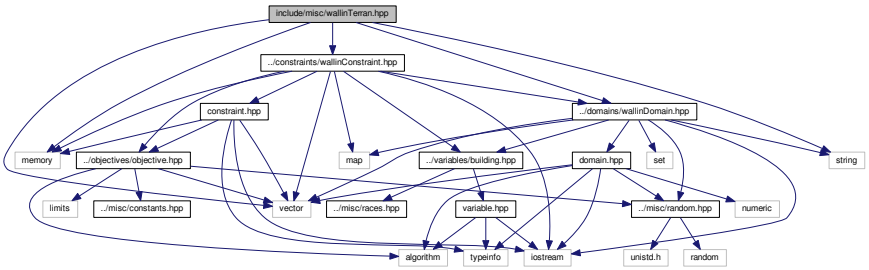
### Variables

- std::shared_ptr< Building > ghost::c
- std::shared_ptr< Building > ghost::f
- std::shared_ptr< Building > ghost::g1
- std::shared_ptr< Building > ghost::g2
- std::shared_ptr< Building > ghost::p1
- std::shared_ptr< Building > ghost::p2
- std::shared_ptr< Building > ghost::y1
- std::shared_ptr< Building > ghost::y2
- std::shared_ptr< Building > ghost::y3
- std::shared_ptr< Building > ghost::y4
- std::shared_ptr< Building > ghost::s
- shared_ptr< Constraint > ghost::overlap
- shared_ptr< Constraint > ghost::buildable
- shared_ptr< Constraint > ghost::noGaps
- shared_ptr< Constraint > ghost::specialTiles
- shared_ptr< Constraint > ghost::pylons

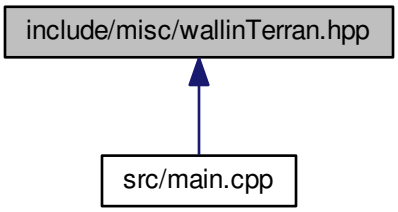
## 8.10 include/misc/wallinTerran.hpp File Reference

```
#include <vector>
#include <memory>
#include <string>
#include "../constraints/wallinConstraint.hpp"
#include "../domains/wallinDomain.hpp"
```

Include dependency graph for wallinTerran.hpp:



This graph shows which files directly or indirectly include this file:




### Namespaces

- ghost

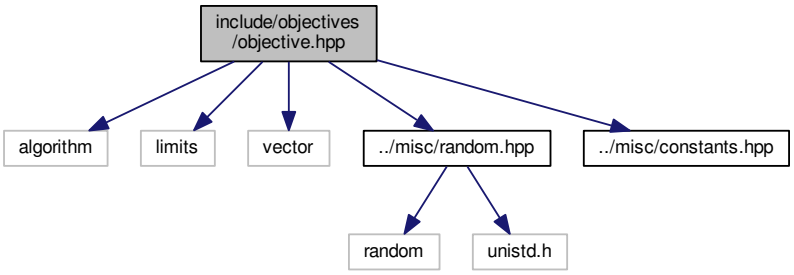
### Functions

- Building ghost::factoryTerranBuilding (const string &name, int pos=-1)
- vector< Building > ghost::makeTerranBuildings ()
- vector< shared_ptr < WallinConstraint > > ghost::makeTerranConstraints (const vector< Building > *vec, const WallinDomain *domain)

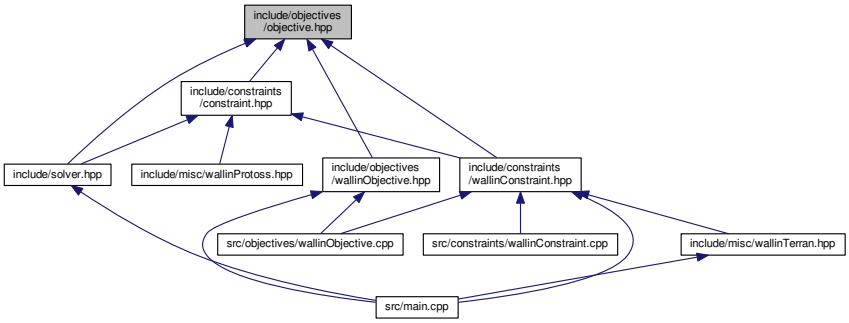
## 8.11 include/objectives/objective.hpp File Reference

```
#include <algorithm>
#include <limits>
#include <vector>
#include "../misc/random.hpp"
#include "../misc/constants.hpp"
```

Include dependency graph for objective.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Objective< TypeVariable, TypeDomain >

  *Objective* *is the class encoding objective functions of your CSP/COP.*

- class ghost::NullObjective< TypeVariable, TypeDomain >

  *NullObjective* *is used when no objective functions have been given to the solver (ie, for pure satisfaction runs).*

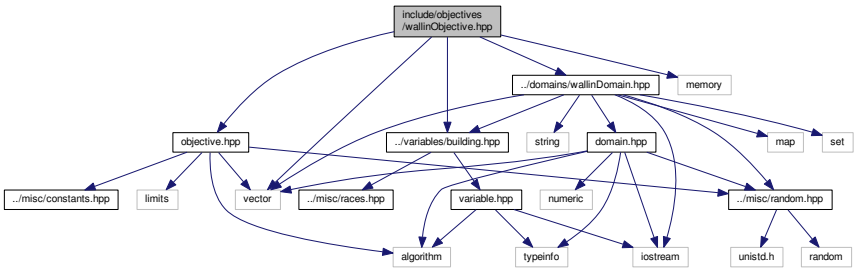
### Namespaces

- ghost

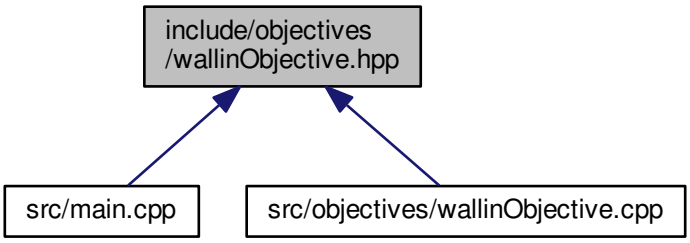
## 8.12 include/objectives/wallinObjective.hpp File Reference

```
#include <vector>
#include <memory>
#include "objective.hpp"
#include "../variables/building.hpp"
#include "../domains/wallinDomain.hpp"
```

Include dependency graph for wallinObjective.hpp:



This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::WallinObjective
- class ghost::NoneObj
- class ghost::GapObj
- class ghost::BuildingObj
- class ghost::TechTreeObj

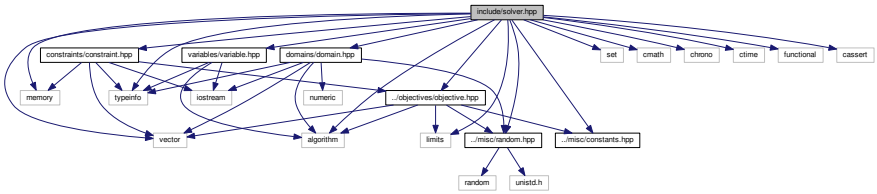
### Namespaces

- ghost

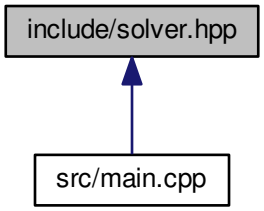
## 8.13 include/solver.hpp File Reference

```
#include <vector>
#include <set>
#include <memory>
#include <cmath>
#include <chrono>
#include <ctime>
#include <limits>
#include <algorithm>
#include <functional>
#include <cassert>
#include <typeinfo>
#include "variables/variable.hpp"
#include "constraints/constraint.hpp"
#include "domains/domain.hpp"
#include "misc/random.hpp"
#include "misc/constants.hpp"
#include "objectives/objective.hpp"
```

Include dependency graph for solver.hpp:




This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Solver< TypeVariable, TypeDomain, TypeConstraint >

  *Solver is the class coding the solver itself.*

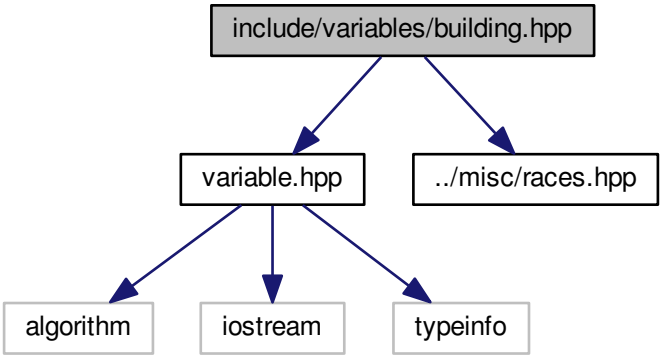
### Namespaces

- ghost

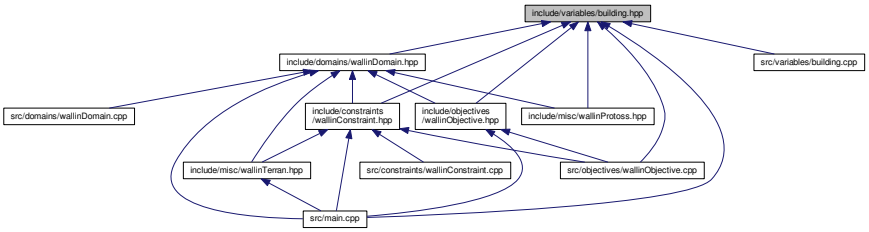
## 8.14 include/variables/building.hpp File Reference

```
#include "variable.hpp"
#include "../misc/races.hpp"
```

Include dependency graph for building.hpp:




This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Building

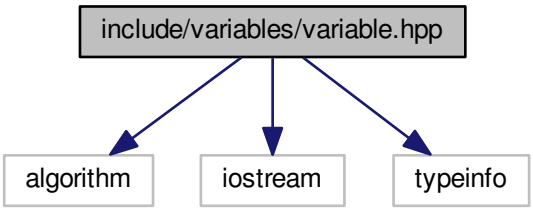
### Namespaces

- ghost

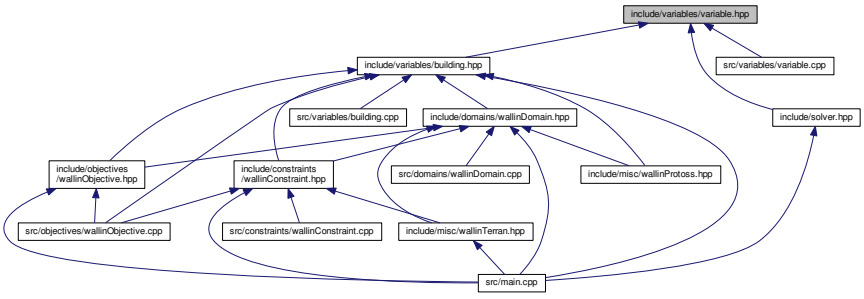
## 8.15 include/variables/variable.hpp File Reference

```
#include <algorithm>
#include <iostream>
#include <typeinfo>
```

Include dependency graph for variable.hpp:




This graph shows which files directly or indirectly include this file:




### Classes

- class ghost::Variable

  *Variable is the class encoding the variables of your CSP/COP.*

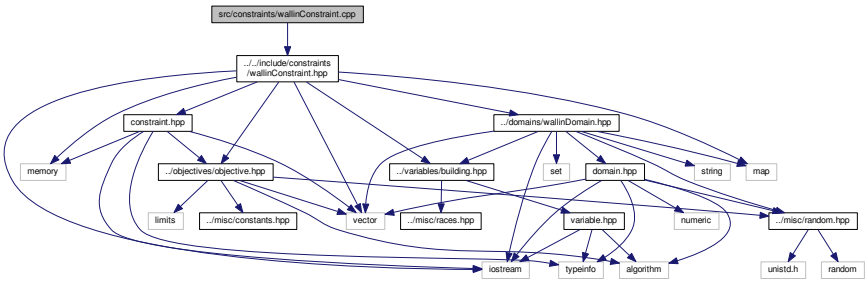
### Namespaces

- ghost

## 8.16 src/constraints/wallinConstraint.cpp File Reference

```
#include "../../include/constraints/wallinConstraint.hpp"
```

Include dependency graph for wallinConstraint.cpp:



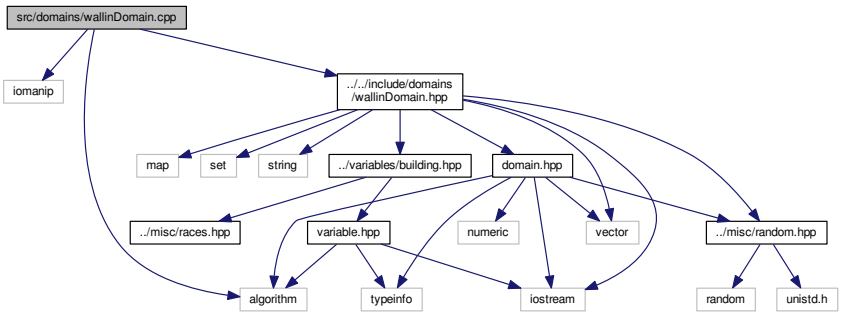

### Namespaces

- ghost

## 8.17 src/domains/wallinDomain.cpp File Reference

```
#include <iomanip>
#include <algorithm>
#include "../../include/domains/wallinDomain.hpp"
```

Include dependency graph for wallinDomain.cpp:




### Namespaces

- ghost

### Functions

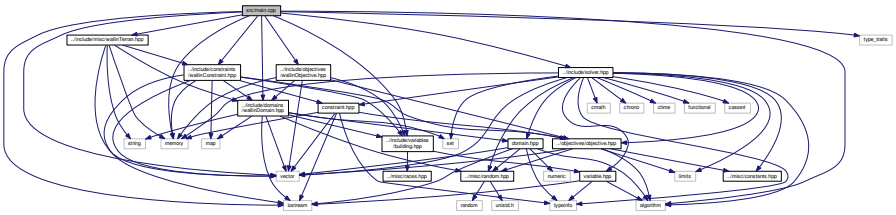
- ostream & ghost::operator<< (ostream &os, const WallinDomain &g)

## 8.18 src/main.cpp File Reference

```
#include <iostream>
```

```
#include <vector>
#include <memory>
#include <algorithm>
#include <type_traits>
#include "../include/variables/building.hpp"
#include "../include/domains/wallinDomain.hpp"
#include "../include/constraints/wallinConstraint.hpp"
#include "../include/objectives/wallinObjective.hpp"
#include "../include/misc/wallinTerran.hpp"
#include "../include/solver.hpp"
```

Include dependency graph for main.cpp:




## Functions

- int main (int argc, char **argv)

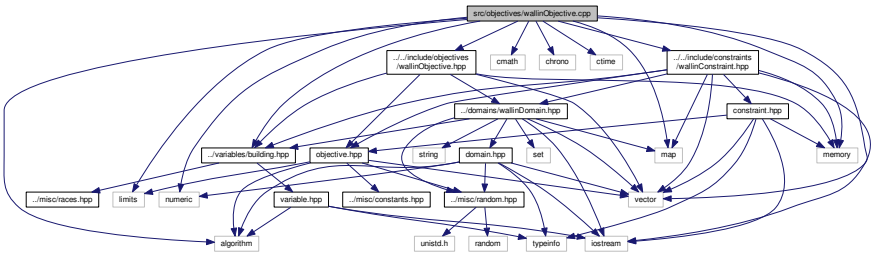
### 8.18.1 Function Documentation

#### 8.18.1.1 int main ( int *argc*, char ** *argv* )

## 8.19 src/objectives/wallinObjective.cpp File Reference

```
#include <vector>
#include <map>
#include <memory>
#include <algorithm>
#include <limits>
#include <cmath>
#include <chrono>
#include <ctime>
#include <numeric>
#include "../../include/objectives/wallinObjective.hpp"
#include "../../include/variables/building.hpp"
#include "../../include/constraints/wallinConstraint.hpp"
```

Include dependency graph for wallinObjective.cpp:



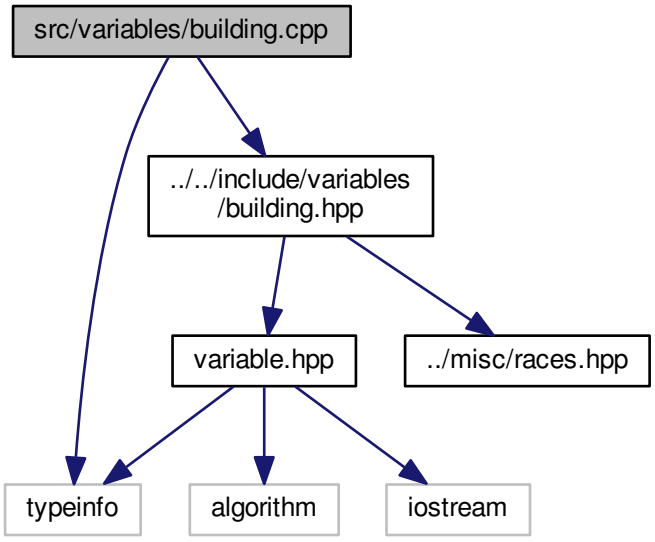

### Namespaces

- ghost

## 8.20 src/variables/building.cpp File Reference

```
#include <typeinfo>
#include "../../include/variables/building.hpp"
```

Include dependency graph for building.cpp:




### Namespaces

- ghost

### Functions

- ostream & ghost::operator<< (ostream &os, const Building &b)

## 8.21 src/variables/variable.cpp File Reference

```
#include "../../include/variables/variable.hpp"
```

Include dependency graph for variable.cpp: