

# **Design Patterns**

## **Elements of Reusable Object-Oriented Software**

*Angelo Afeltra*

# Che cosa sono i Design Pattern?

*I Design Pattern descrivono problemi che si verifica più è più volte nello sviluppo software. Per tali problemi essi descrivono la relativa soluzione in modo da poterla utilizzare milioni di volte, senza farlo allo stesso modo due volte*

# Come sono composti i Design Pattern?

Design  
Pattern

Pattern Name

Identificativo del design pattern

Problema

Spiega il problema e il suo contesto, includendo a volte un elenco di condizioni che devono essere soddisfatte in modo tale che abbia senso applicare il pattern

Soluzione

Describe gli elementi che compongono il design pattern le loro relazioni, responsabilità e collaborazioni. La soluzione non descrive un implementazione concreta, ma fornisce una descrizione astratta ad un problema di progettazione

Conseguenza

Risultati e compromessi dell'applicazione del pattern. Esse devono essere prese in considerazione in modo da valutare alternative di progettazione.

# Come sono organizzati i Design Pattern?

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

# Come sono organizzati i Design Pattern?

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method (107)	Adapter (class) (139)	Interpreter (243)
Object	Abstract Factory (101)	I pattern che operano sulla classe si occupano delle relazioni tra le classi e le loro sottoclassi. Queste relazioni sono stabilite attraverso l'ereditarietà, quindi sono statiche, fisse al momento della compilazione.		
	Builder (114)	Singleton (127)	Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Template Method (325) Composite (163) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

# Come sono organizzati i Design Pattern?

		Purpose			
		Creational	Structural	Behavioral	
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243)	
Object	Builder	I pattern che operano sugli oggetti si occupano di relazioni con gli oggetti, che possono essere modificate in fase di esecuzione e sono più dinamici.	Prototype (117) Singleton (127)	Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Template Method (325) Abstract Factory (67) Builder (99) Bridge (151) Command (183) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

# Come sono organizzati i Design Pattern?

Scope	Class	Purpose		
		Creational	Structural	Behavioral
Object	Factory Method (107)	Adapter (class) (139)	Interpreter (243)	Template Method (325)
Object	Abstract Factory (87)	Adapter (object) (139)	Chain of Responsibility (223)	Builder (97)
Object	Singleton (127)	Decorator (175)	Mediator (273)	Facade (185)
Object		Flyweight (195)	Observer (293)	Proxy (207)
Object		Proxy (207)	State (305)	
Object			Strategy (315)	
Object			Visitor (331)	

I pattern creazionali riguardano il processo di creazione degli oggetti.

Quando agiscono sulle classi sfruttano l'ereditarietà per variare la classe che è istanziata, mentre se agiscono sugli oggetti delegano l'istanza a un altro oggetto.

# Come sono organizzati i Design Pattern?

Scope	Class	Purpose		
		Creational	Structural	Behavioral
Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243)	Template Method (325)
Object	Abstract Factory (87) Builder (127) Bridge (127) Composite (127) Command (231) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Adapter (object) (139) Composite (127) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (231) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)	

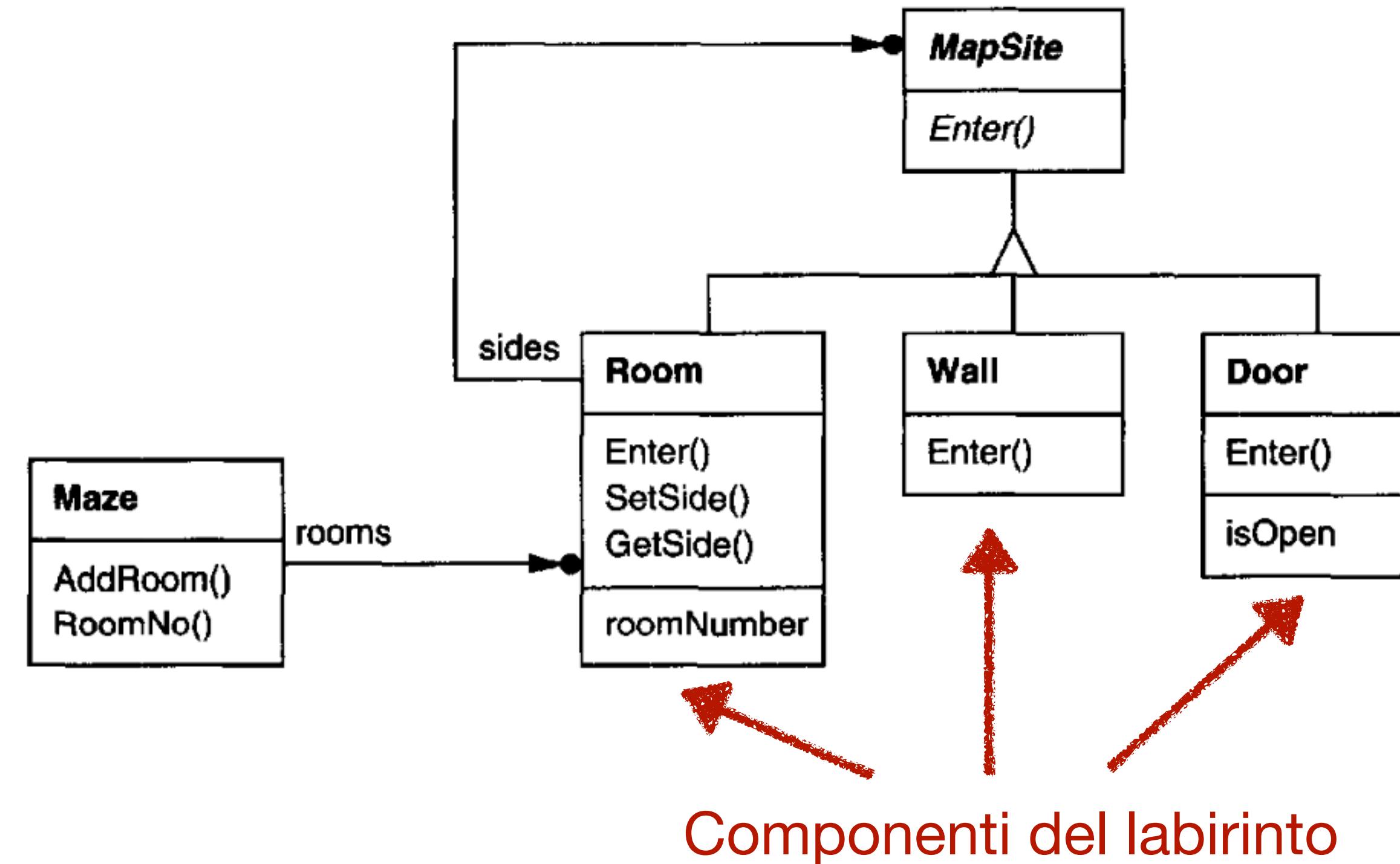
I pattern strutturali si occupano della composizione di classi e oggetti. Quando agiscono sulle classi sfruttano l'ereditarietà per comporre le classi, al contrario quando agiscono sugli oggetti descrivono i modi per assemblare gli oggetti.

# Come sono organizzati i Design Pattern?

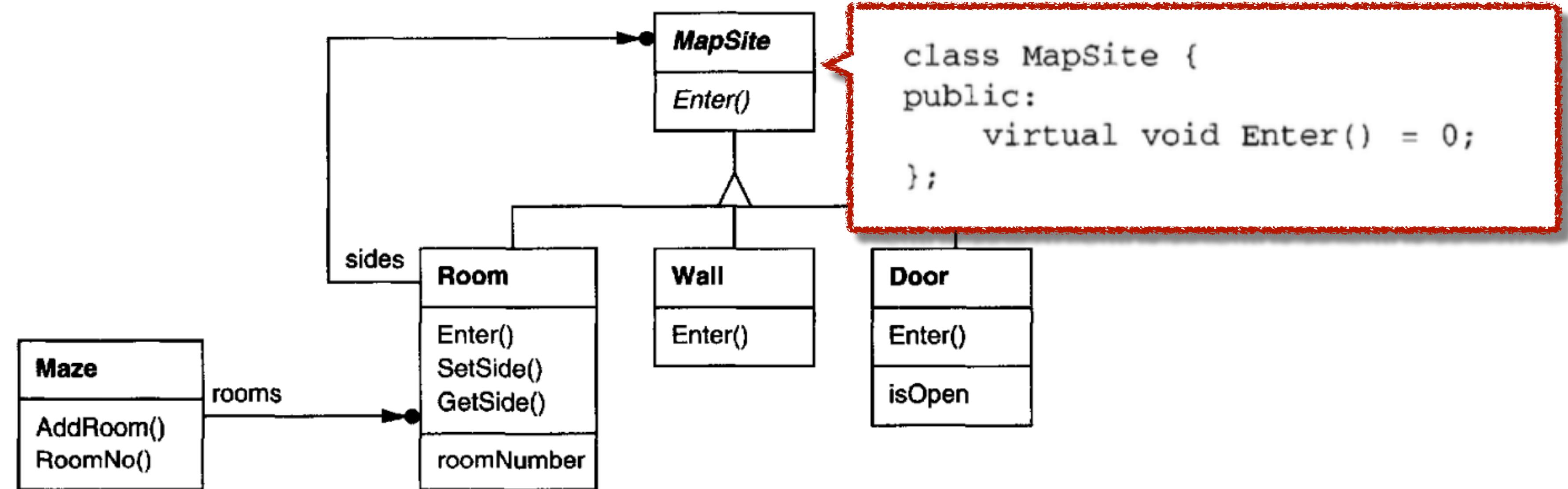
Purpose				
	Creational	Structural	Behavioral	
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (323)
Object	I pattern comportamentali caratterizzano il modo in cui le classi e gli oggetti interagiscono e distribuiscono la responsabilità.  Quando agiscono sulle classi utilizzando l'ereditarietà per descrivere algoritmi e flussi di controllo. Mentre se agiscono sugli oggetti essi descrivono come gruppi di oggetti collaborano per eseguire un'attività che nessun singolo oggetto può eseguire da solo.	Abstract Factory (87) Builder (97) Composite (114) Decorator (128) Facade (169) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (237) Observer (293) State (305)	Strategy (315) Visitor (331)

# Pattern Creazionali - Esempio Applicazione (Labirinto)

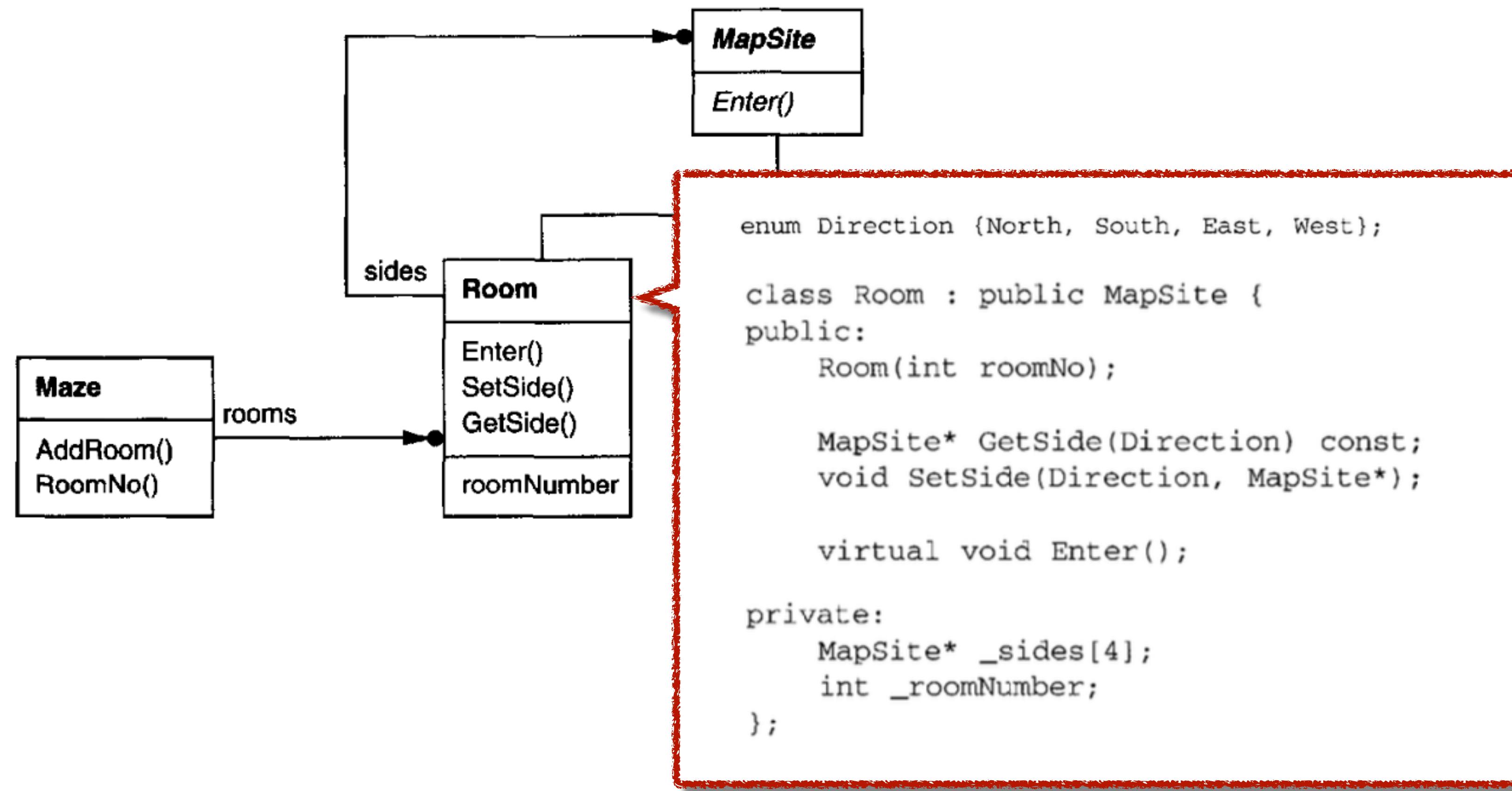
Un labirinto è composto da un insieme di stanze, dove ogni stanza conosce il suo vicino che può essere un'altra stanza, un muro o una porta.



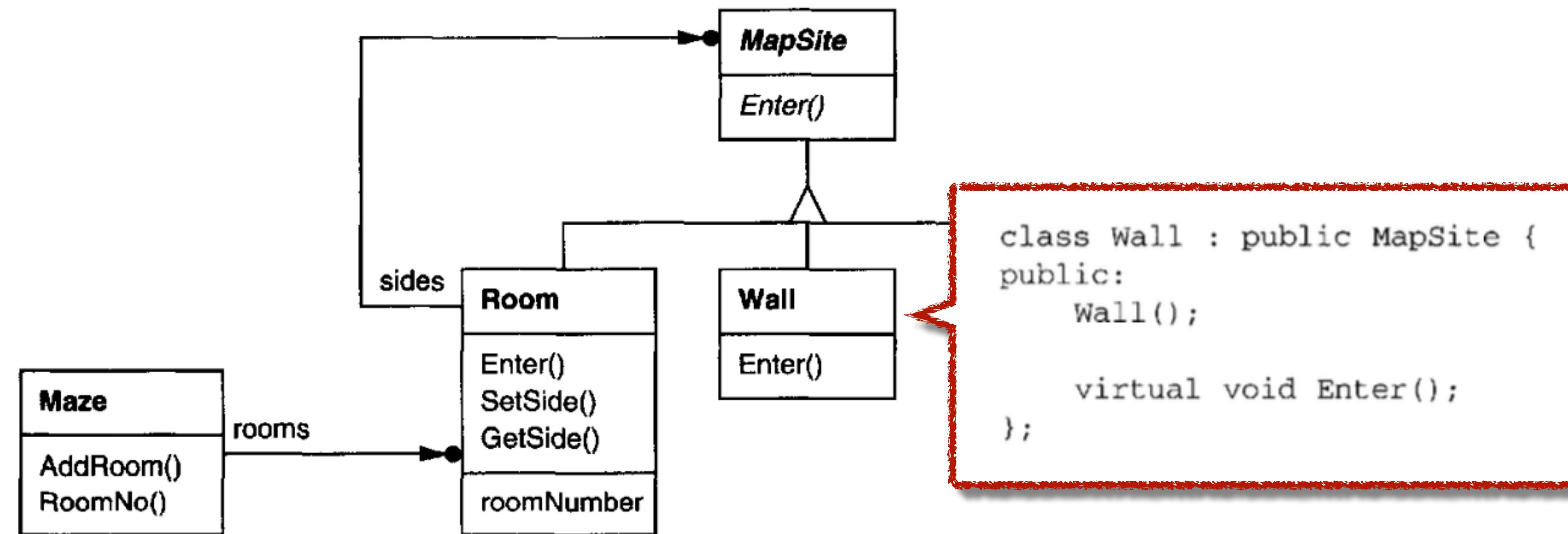
# Pattern Creazionali - Esempio Applicazione (Labirinto)



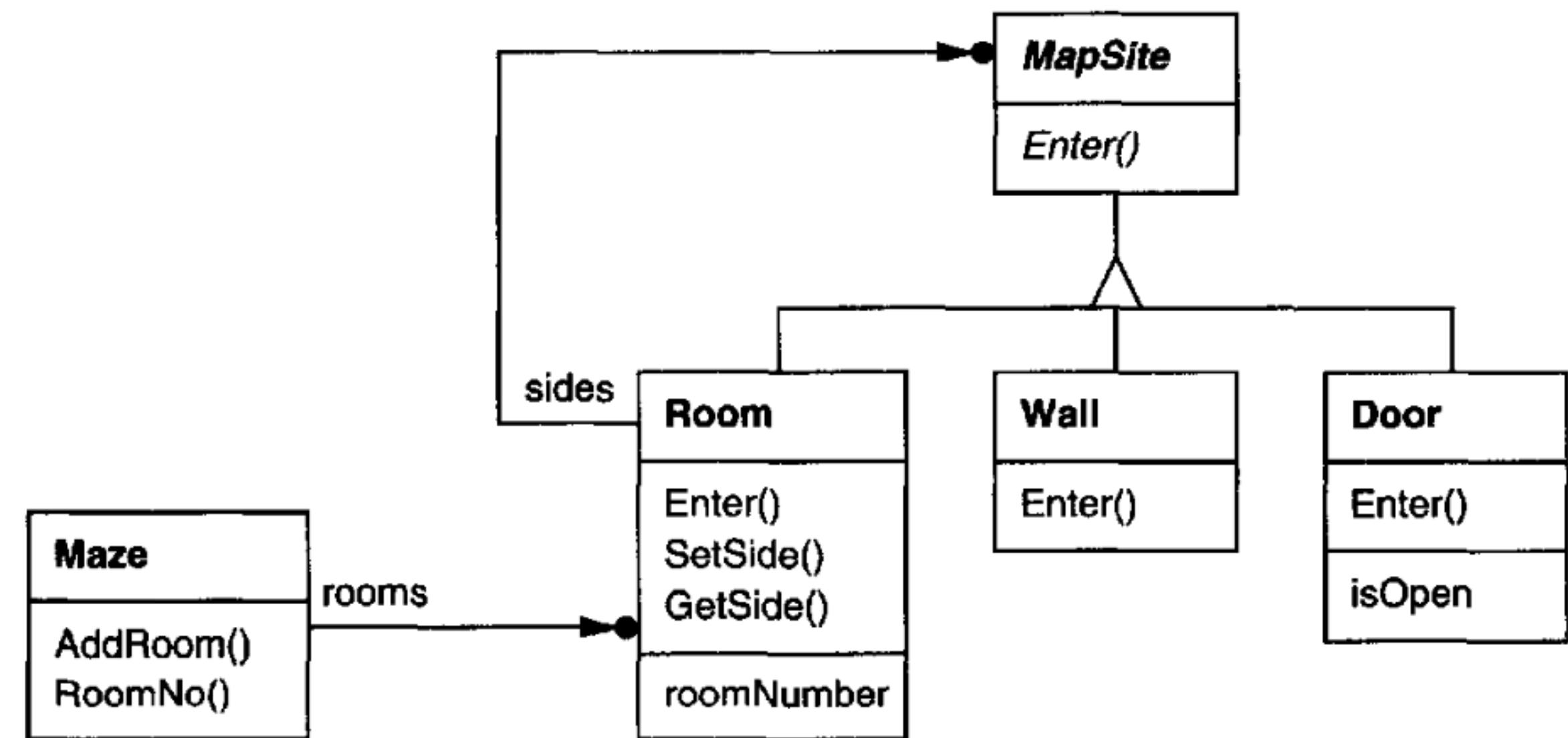
# Pattern Creazionali - Esempio Applicazione (Labirinto)



# Pattern Creazionali - Esempio Applicazione (Labirinto)



# Pattern Creazionali - Esempio Applicazione (Labirinto)



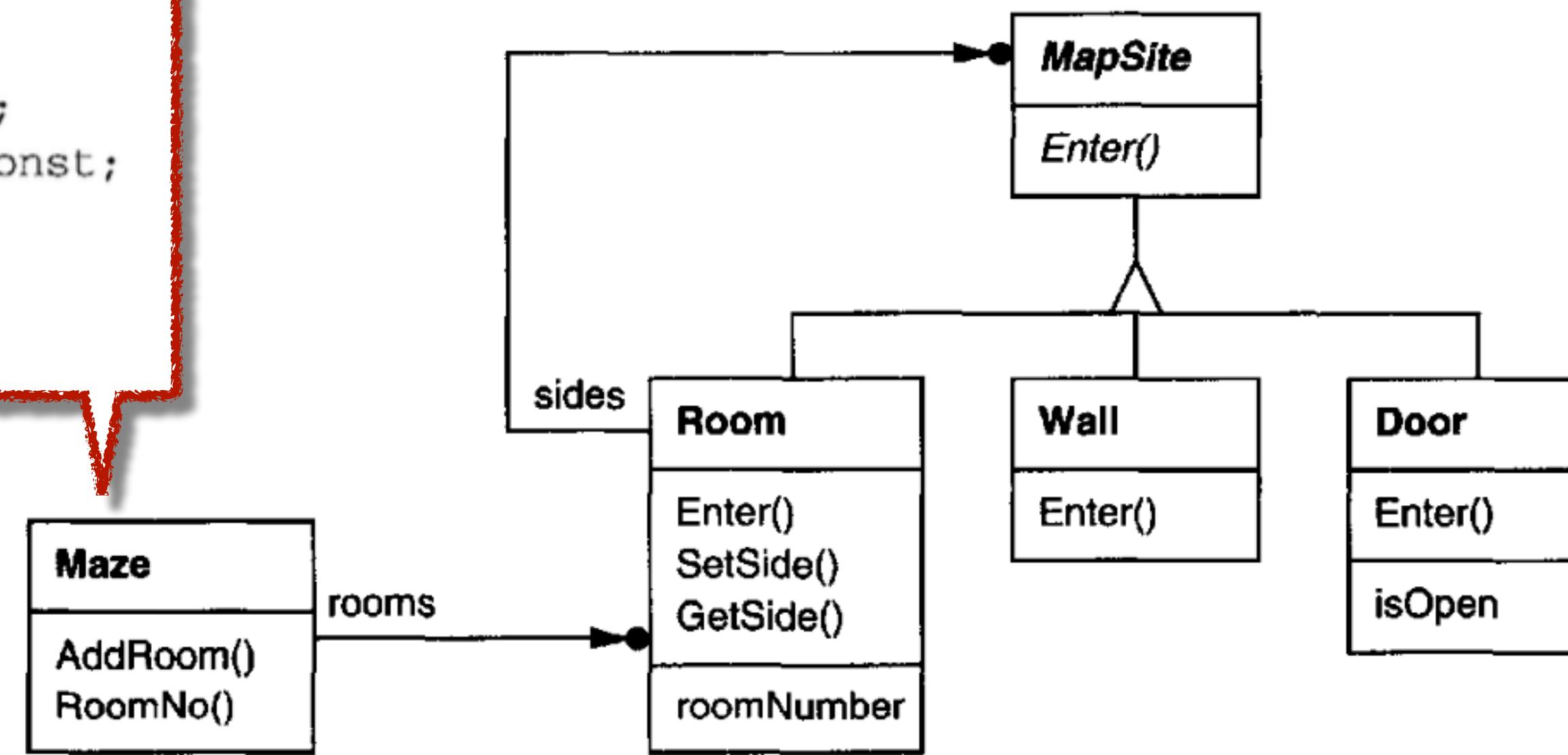
```
class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

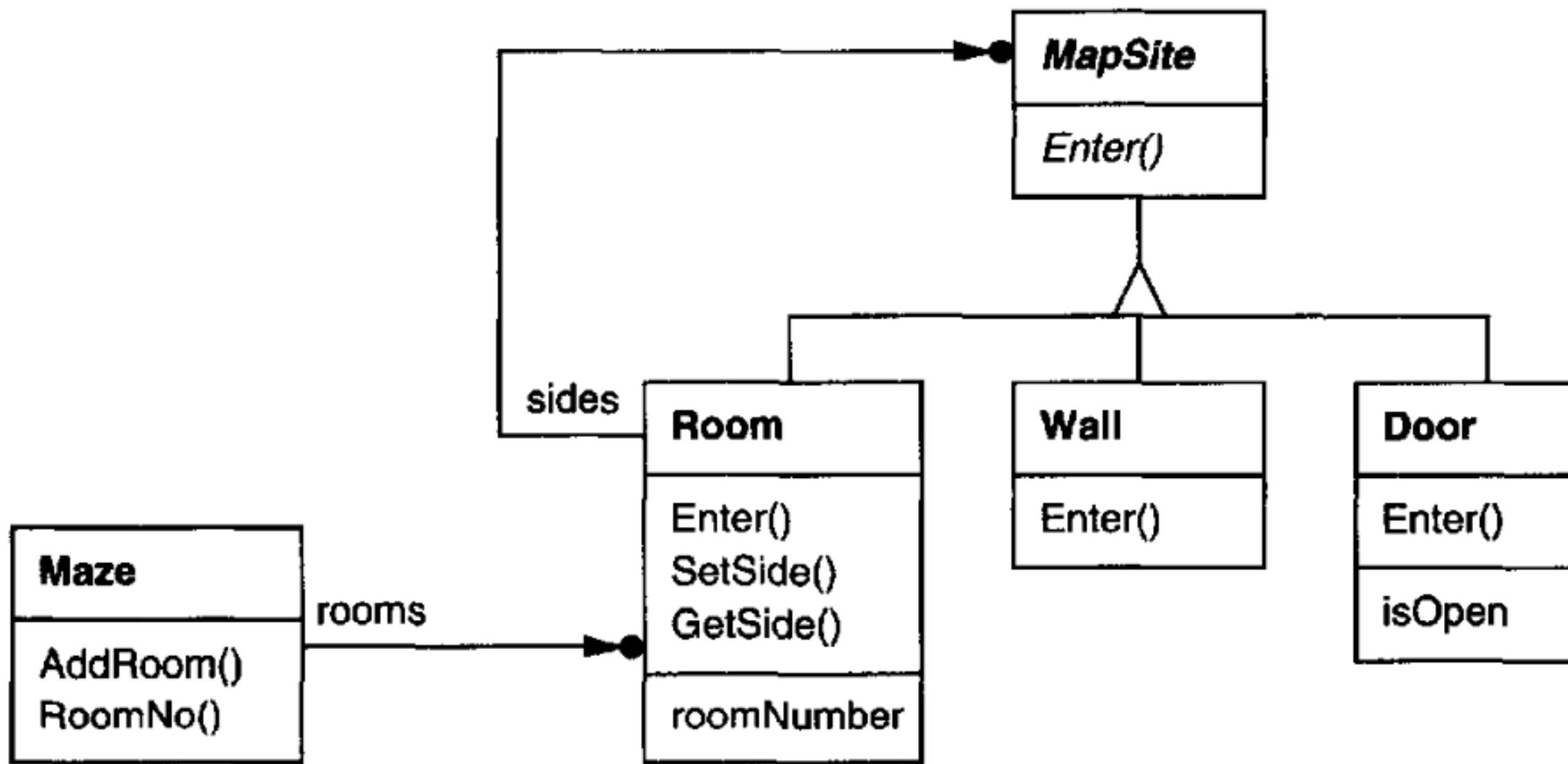
private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

# Pattern Creazionali - Esempio Applicazione (Labirinto)

```
class Maze {  
public:  
    Maze();  
  
    void AddRoom(Room*);  
    Room* RoomNo(int) const;  
private:  
    // ...  
};
```



# Pattern Creazionali - Esempio Applicazione (Labirinto)



```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

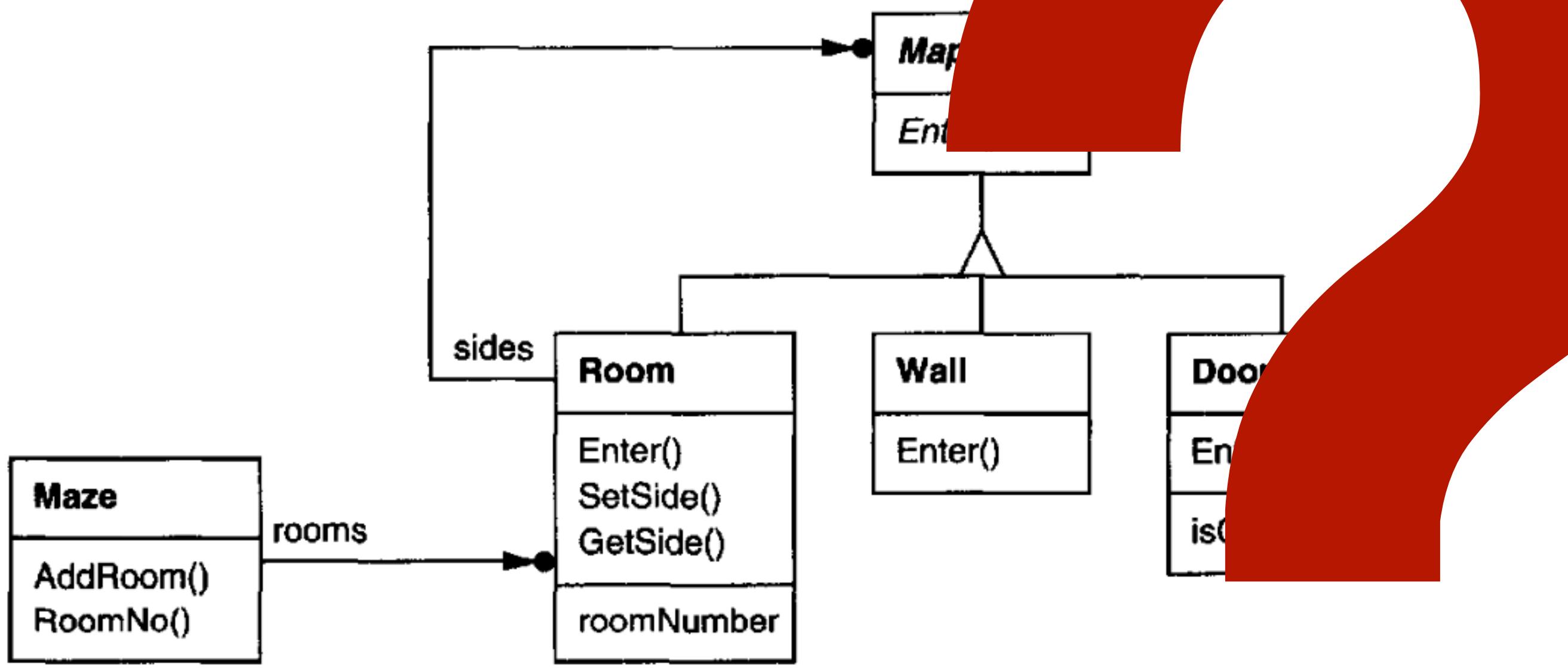
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

# Pattern Creazionali - Esempio Applicazione (Labirinto)



```
maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

# Pattern Creazionali - Esempio Applicazione (Labirinto)

```
Maze* MazeGame::CreateMaze () {  
    Maze* aMaze = new Maze;  
    Room* r1 = new Room(1);  
    Room* r2 = new Room(2);  
    Door* theDoor = new Door(r1, r2);  
  
    aMaze->AddRoom(r1);  
    aMaze->AddRoom(r2);  
  
    r1->SetSide(North, new Wall);  
    r1->SetSide(East, theDoor);  
    r1->SetSide(South, new Wall);  
    r1->SetSide(West, new Wall);  
  
    r2->SetSide(North, new Wall);  
    r2->SetSide(East, new Wall);  
    r2->SetSide(South, new Wall);  
    r2->SetSide(West, theDoor);  
  
    return aMaze;  
}
```

Considerando che creiamo un labirinto con solo due stanze la funzione è piuttosto complicata. Essa codifica duramente il layout del labirinto, in quanto per cambiarlo bisognerebbe riimplementare l'intera funzione. Con un design pattern creazionale si può rendere il design più flessibile.

Ad esempio si potrebbe applicare i pattern:

- Factory Method
- Abstract Factory
- Builder
- Prototype

# Come scelgo il design pattern giusto da utilizzare?

## Identificazione del problema

La prima fase consiste nell'identificare un problema specifico nel tuo progetto software. Questo potrebbe essere un problema di progettazione o una sfida ricorrente che altri sviluppatori hanno già affrontato in passato.

# Come scelgo il design pattern giusto da utilizzare?

Identificazione del problema

Ricerca del pattern

Una volta che hai identificato il problema, inizia a cercare design pattern che potrebbero essere adatti per risolverlo.

Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.		
Adapter (139) Convert the interface of a class into another interface clients expect.	Adapter lets classes use interchangeable interfaces.	Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.
Bridge (151) Decouple an application's interface from its implementation so that the two can vary independently.	Interpreter (243) Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Builder (97) Separate the construction of a complex object from its representation so that the same construction code can build different representations.	Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.	Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Chain of Responsibility (223) Establish a chain of responsibility that allows requests to pass between several objects until an object handles them.	Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.	Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.	Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.	Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composite structures uniformly.	Prototype (117) Specify the kinds of objects to create using a prototypical instance, and then create new objects by copying this prototype.	
Decorator (175) Attach additional responsibilities to objects dynamically. Decorator lets you provide more functionality to objects without inheritance.	Proxy (207) Provide a surrogate or placeholder for another object to control access to it.	
Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that simplifies the interface of the subsystem.	Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.	
Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to implement.	State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	
Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	
Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.		

Purpose			
	Creational	Structural	Behavioral
Scope	Factory Method (107)	Adapter (class) (139)	Interpreter (243)
Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

# Come scelgo il design pattern giusto da utilizzare?

Identificazione del problema

Ricerca del pattern

Analisi dei candidati

Quando hai individuato alcuni design pattern potenziali, analizzali attentamente per determinare se si adattano al tuo caso specifico. Considera i seguenti fattori:

- Confronta i requisiti del tuo problema con le caratteristiche del design pattern.
- Valuta le implicazioni di design, come la flessibilità, l'estendibilità e le prestazioni.
- Controlla se ci sono esempi o casi d'uso simili in cui il design pattern è stato applicato con successo.

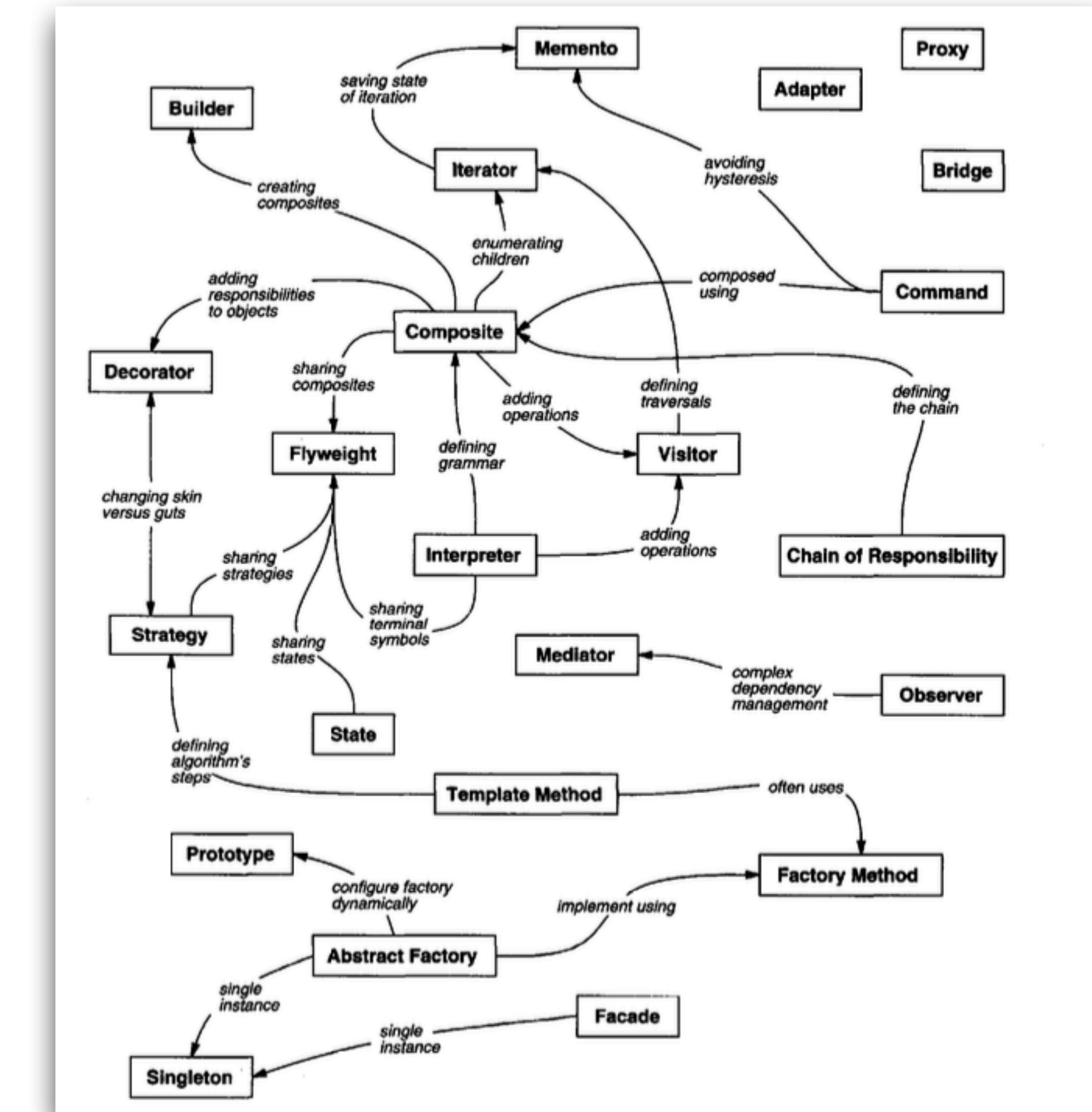
# Come scelgo il design pattern giusto da utilizzare?

Identificazione del problema

Ricerca del pattern

Analisi dei candidati

Analizza le relazioni tra pattern



# Come scelgo il design pattern giusto da utilizzare?

Identificazione del problema

Ricerca del pattern

Analisi dei candidati

Analizza le relazioni tra pattern

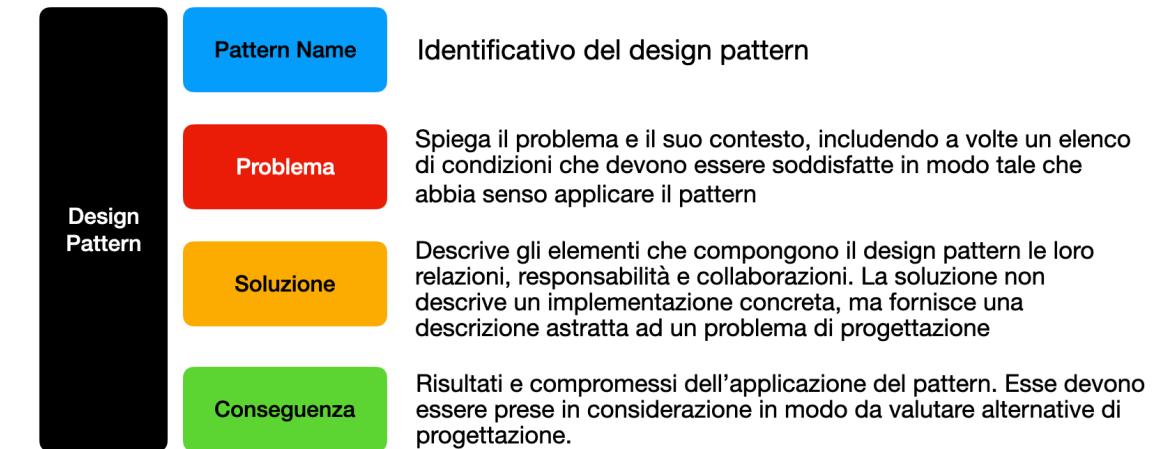
Adattamento e personalizzazione

Spesso, è necessario adattare o personalizzare il design pattern in base alle esigenze specifiche del tuo progetto. Questo potrebbe richiedere modifiche o estensioni al design pattern stesso.

## Che cosa sono i Design Pattern?

I Design Pattern descrivono problemi che si verifica più è più volte nello sviluppo software. Per tali problemi essi descrivono la relativa soluzione in modo da poterla utilizzare milioni di volte, senza farlo allo stesso modo due volte

## Come sono composti i Design Pattern?

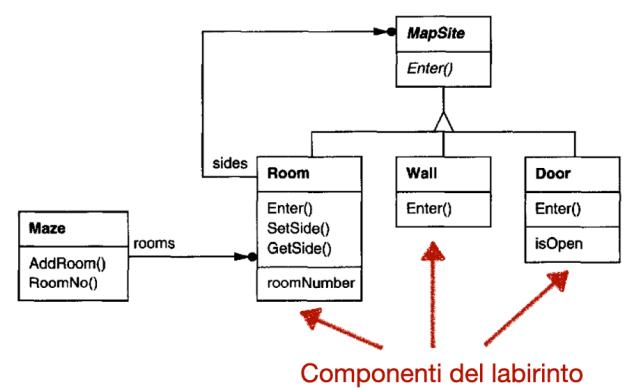


## Come sono organizzati i Design Pattern?

Purpose			
	Creational	Structural	
Scope	Class	Behavioral	
Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Interpreter (243) Template Method (325) Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

## Pattern Creazionali - Esempio Applicazione (Labirinto)

Un labirinto è composto da un insieme di stanze, dove ogni stanza conosce il suo vicino che può essere un'altra stanza, un muro o una porta.



## Pattern Creazionali - Esempio Applicazione (Labirinto)

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Considerando che creiamo un labirinto con solo due stanze la funzione è piuttosto complicata. Essa codifica duramente il layout del labirinto, in quanto per cambiarlo bisognerebbe riimplementare l'intera funzione. Con un design pattern creazionale si può rendere il design più flessibile.

Ad esempio si potrebbe applicare i pattern:

- Factory Method
- Abstract Factory
- Builder
- Prototype

## Come scelgo il design pattern giusto da utilizzare?



Spesso, è necessario adattare o personalizzare il design pattern in base alle esigenze specifiche del tuo progetto. Questo potrebbe richiedere modifiche o estensioni al design pattern stesso.

# Design Patterns Elements of Reusable Object-Oriented Software