Modellistica Medica

Maria Grazia Pia

INFN Genova

Scuola di Specializzazione in Fisica Sanitaria
Genova
Anno Accademico 2002-2003

Lezione 8

OO modeling

Design Patterns

Introduction

Creational Patterns

Software patterns

- The concept of a pattern as used in software architecture is borrowed from the field of (building) architecture, in particular from the writings of architect Christopher Alexander

- "A ***pattern for software architecture*** describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." *(Buschman)*

Characteristics of patterns

In general, patterns have the following characteristics:

- A pattern describes a **solution to a recurring problem** that arises in specific design situations
- Patterns are not invented; they are distilled from practical experience
- Patterns describe a group of components (e.g., classes or objects), how the components interact, and the responsibilities of each component: that is, they are higher level abstractions than classes or objects
- Patterns **provide a vocabulary** for communication among designers
- Patterns help **document the architectural vision** of a design: if the vision is clearly understood, it will less likely be violated when the system is modified
- Patterns provide a conceptual skeleton for a solution to a design problem and, hence, encourage the construction of software with well-defined properties
- Patterns are building blocks for the construction of more complex designs
- Patterns help designers manage the complexity of the software: when a recurring pattern is identified, the corresponding general solution can be implemented productively to provide a reliable software system

What is a pattern

- A pattern is a nugget of insight that conveys the essence of a proven solution to a recurring problem
- A pattern is the abstraction from a concrete form which keeps recurring in various contexts

Categories of Patterns

- Patterns can be grouped into three categories according to their level of abstraction:
  - Architectural patterns
  - Design patterns
  - Idioms

Architectural patterns

- An architectural pattern expresses a fundamental structural organization schema for software systems
  - It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them
- An architectural pattern is a high-level abstraction
  - The choice of the architectural pattern to be used is a fundamental design decision in the development of a software system
  - It determines the system-wide structure and constrains the design choices available for the various subsystems
  - It is, in general, independent of the implementation language to be used

Design patterns

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them
  - It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context
- A design pattern is a mid-level abstraction
  - The choice of a design pattern does not affect the fundamental structure of the software system, but it does affect the structure of a subsystem
  - Like the architectural pattern, the design pattern tends to be independent of the implementation language to be used

Idioms

- "An idiom is a low-level pattern specific to a programming language
  - An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language
- An idiom is a low-level abstraction
  - It is usually a language-specific pattern that deals with some aspects of both design and implementation

Design Patterns

- Description of communicating objects and classes that are customized to solve a general design in a particular context [GoF]

- Design patterns

  capture the static and dynamic structures of solutions that occur repeatedly

- A design pattern is
  - a template solution to a recurring design problem

- Look before re-inventing the wheel just one more time
  - …reusable design knowledge
  - …an example of modifiable design

Learning to design starts by studying other designs

Purpose of design patterns

- Having found a good solution to a class of problem, the design knowledge is classified and recorded for others to profit of it
  - experience is often the difference between good and poor designers
- Patterns provide:
  - an aid to the reuse of design and analysis
  - a distillation of good practice
  - a means of communication between designers

Modifiable design & software process

A modifiable design enables

- an iterative and incremental development cycle
- concurrent development
- risk management
- flexibility to change

- to minimize the introduction of new problems when fixing old ones
- to deliver more functionality after initial delivery

Design patterns & modifiable designs

- What makes a design easy to evolve?
  - Low coupling and high coherence
  - Clear dependencies
  - Explicit assumptions
- How do design patterns help?
  - They are generalized from existing systems
  - They provide a shared vocabulary to designers
  - They provide examples of modifiable designs
- The way they help:
  - **Abstract classes**
  - **Delegation**

Reference book

GoF *(Gamma, Helm, Johnson, Vlissides)*

Design Patterns: Elements of Reusable Object-Oriented Software, 1995

- Suggestion: a fundamental book to read (and use!)
- but not for beginners…

Structure of a pattern

In general, a pattern has four essential elements:

- the **pattern name**
  - describing the design problem, its solution and consequences in a word or two
- the **problem**
  - describes the problem itself, as well as the context in which the pattern can be applied
- the **solution**
  - describes the abstract elements that make up the design, their relationships, responsibilities and collaborations
- the **consequences**
  - are the results of trade-offs of applying the pattern
  - they are used to evaluate the the costs and benefits of applying the pattern

Design Patterns: categories

- Structural Patterns
  - *Focus:*
    - How objects are composed to form larger structures
  - *Problems solved:*
    - Realize new functionality from old functionality
    - Provide flexibility and extensibility
- Behavioral Patterns
  - *Focus:*
    - Algorithms and the assignment of responsibilities to objects
  - *Problem solved:*
    - Too tight coupling to a particular algorithm
- Creational Patterns
  - *Focus:*
    - Creation of complex objects
  - *Problems solved:*
    - Hide how complex objects are created and put together

Pattern classification categories

- Creational class patterns
  - Defer some part of object creation to subclasses
- Creational object patterns
  - Defer some part of object creation to another object

- Structural class patterns
  - Use inheritance to compose classes
- Structural object patterns
  - Describe ways to assemble objects

- Behavioral class patterns
  - Use inheritance to describe algorithms and flow of control
- Behavioral object patterns
  - Describe how a group of objects cooperate to perform a task that no single object can carry out alone

Pattern classification categories

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Creational Patterns

Patterns used to abstract the process of instantiating objects

- Class-scoped patterns
  - use inheritance to choose the class to be instantiated
    - Factory Method
- Object-scoped patterns
  - use delegation
    - Abstract Factory
    - Builder
    - Prototype
    - Singleton

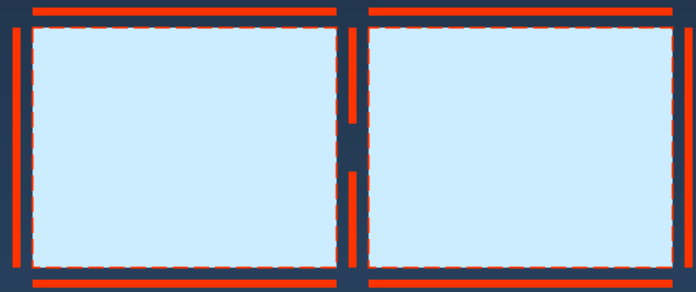
Recurring Themes

- Hide: which concrete classes the system uses
- Hide: how instances are created and associated

- Creational patterns give flexibility in
  - what gets created
  - who creates it
  - how it gets created
  - when it get gets created

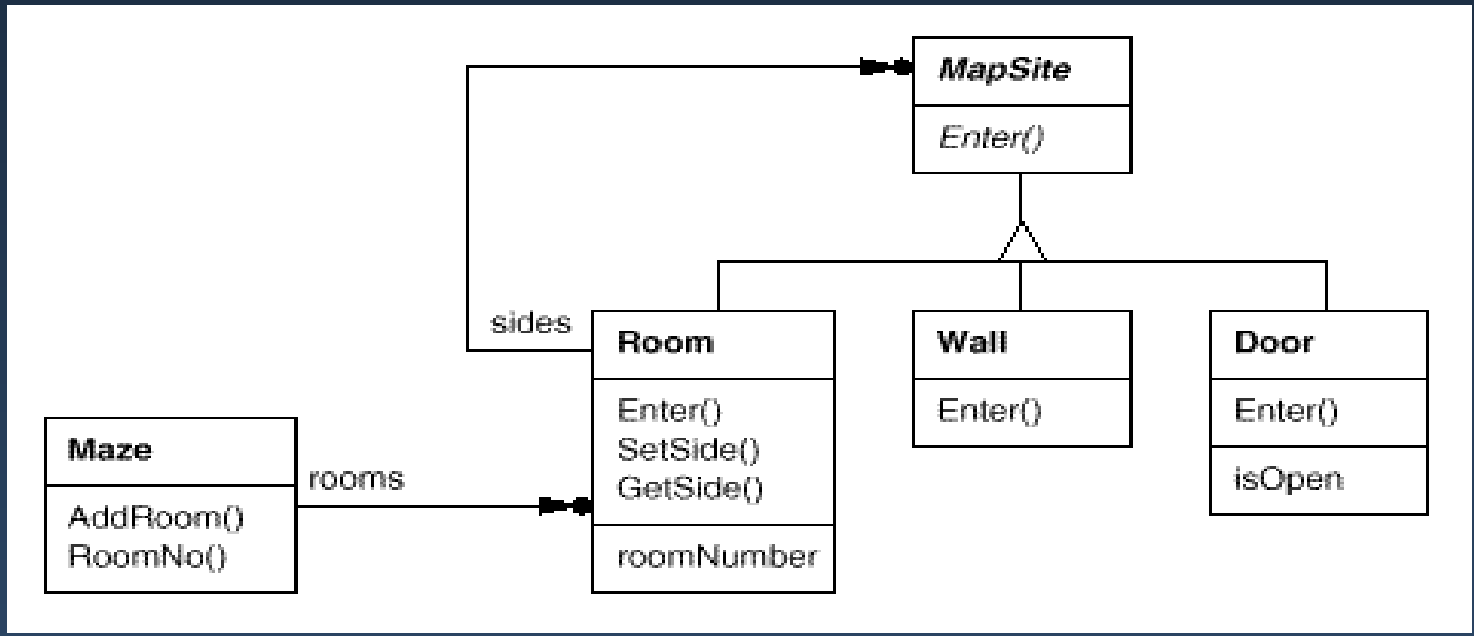
Example from GoF

- Building a maze for a computer game



- A Maze is a set of Rooms
- A Room knows its neighbours
  - another room
  - a wall
  - a door

Maze Example



- MapSite provides a general purpose abstraction for parts of the maze
  - entering a room changes your location
  - entering a door causes you to enter the adjoining room, if it is open
  - entering a wall won't get you very far in a standard maze

Maze Creation

- Fairly complex member just to create a maze with two rooms
- Obvious simplification:
  - Room() could initialize sides with 4 new Wall()
  - That just moves the code elsewhere
- Problem lies elsewhere: *inflexibility*
  - Hard-codes the maze creation
  - Changing the layout can only be done by re-writing, or overriding and re-writing

Creational Patterns Benefits

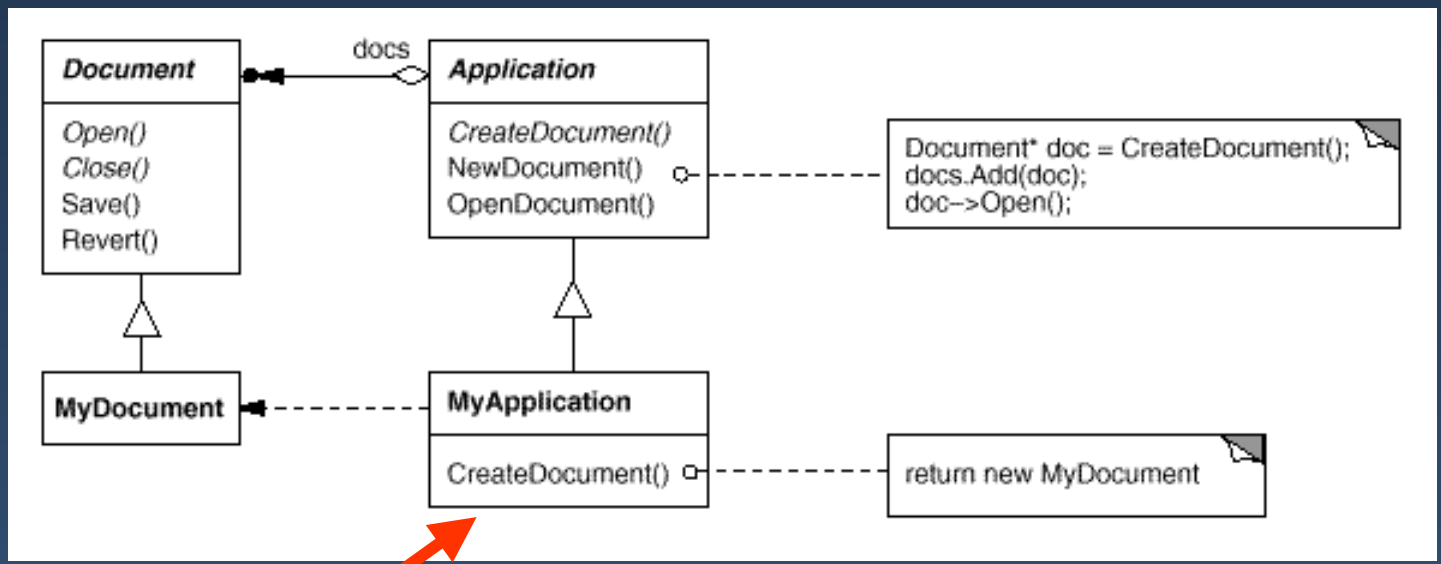
- Will make the maze more flexible
  - easy to change the components of a maze
  - e.g., DoorNeedingSpell, EnchantedRoom
    - How can you change createMaze() so that it creates mazes with these different kind of classes?
    - Biggest obstacle is hard-coding of class names.

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method
- If createMaze() is passed a parameter object to create rooms, walls, …
  - Abstract Factory
- If createMaze() is passed a parameter object to create and connect-up mazes
  - Builder
- If createMaze is parameterized with various prototypical rooms, doors, walls, … which it copies and then adds to the maze
  - Prototype
- Need to ensure there is only one maze per game, and everybody can access it, and can extend or replace the maze without touching other code.
  - Singleton

Factory Method

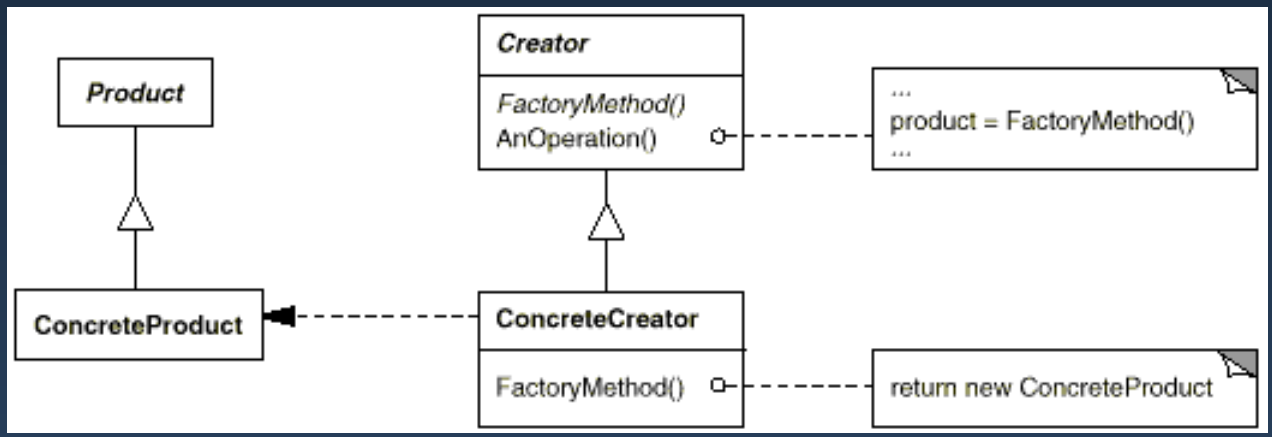
- Define an interface for creating an object, but let subclasses decide which class to instantiate
- a.k.a. Virtual Constructor
- e.g., application framework



Applicability

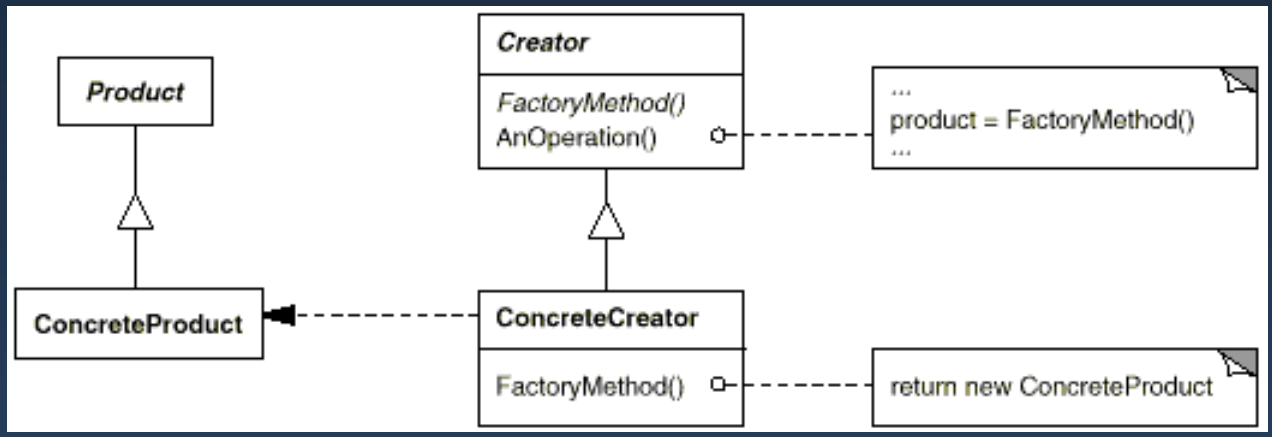
- Use when:
  - A class can't anticipate the kind of objects to create

Structure



- Product
  - defines the interface of objects the factory method creates
- ConcreteProduct
  - implements the Product interface

Structure



- Creator
  - declares the factory method which returns a Product type
  - [define a default implementation]
  - [call the factory method itself]

- ConcreteCreator
  - overrides the factory method to return an instance of a ConcreteProduct

Consequences

Advantage:

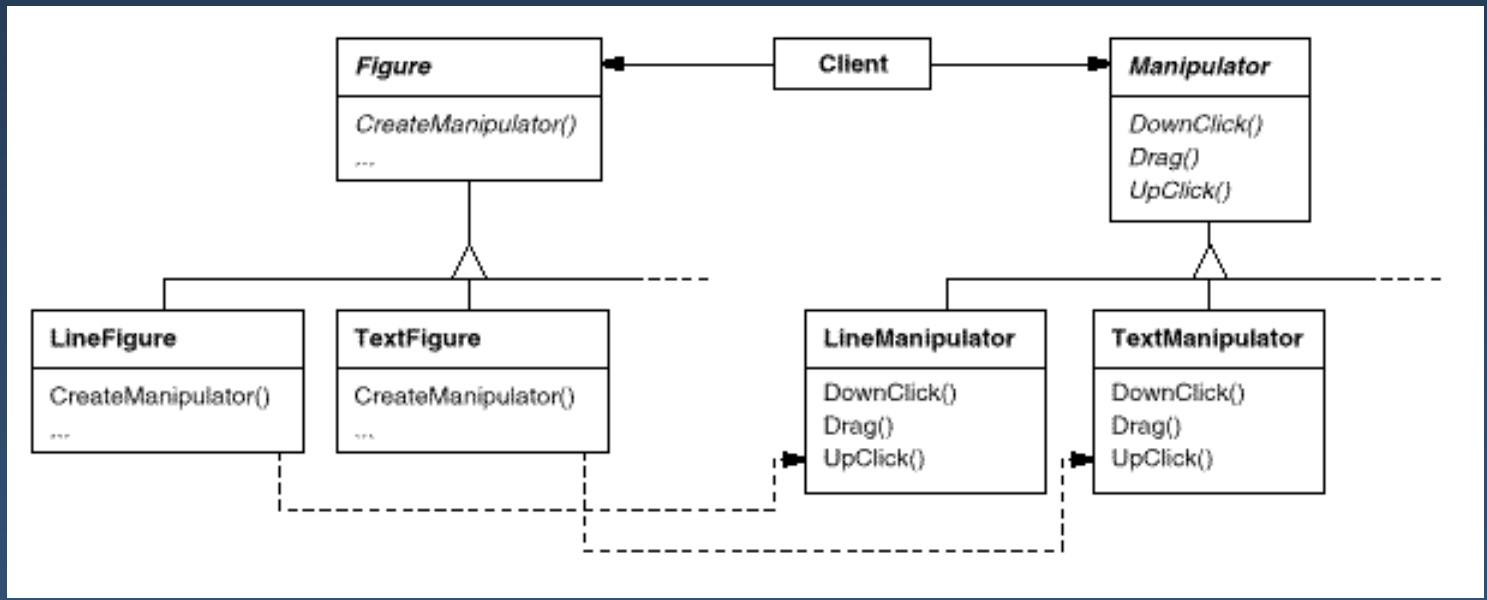
- Eliminates the need to bind to specific implementation classes
  - can work with any user-defined ConcreteProduct classes

Disadvantage:

- Uses inheritance
- Must subclass to define new ConcreteProduct objects
  - consistent interface required

Consequences

- Provides hooks for subclasses
  - always more flexible than direct object creation
- Connects parallel class hierarchies
  - hides which classes belong together



Implementation

- Two major varieties
  - creator class is abstract
    - *requires* subclass to implement
  - creator class is concrete, and provides a default implementation
    - *optionally allows* subclass to re-implement
- Parameterized factory methods
  - takes a class id as a parameter to a generic make() method
- Naming conventions
  - use ‘makeXXX()’ type conventions (e.g., MacApp – DoMakeClass())
- Return class of object to be created
  - or, store as member variable

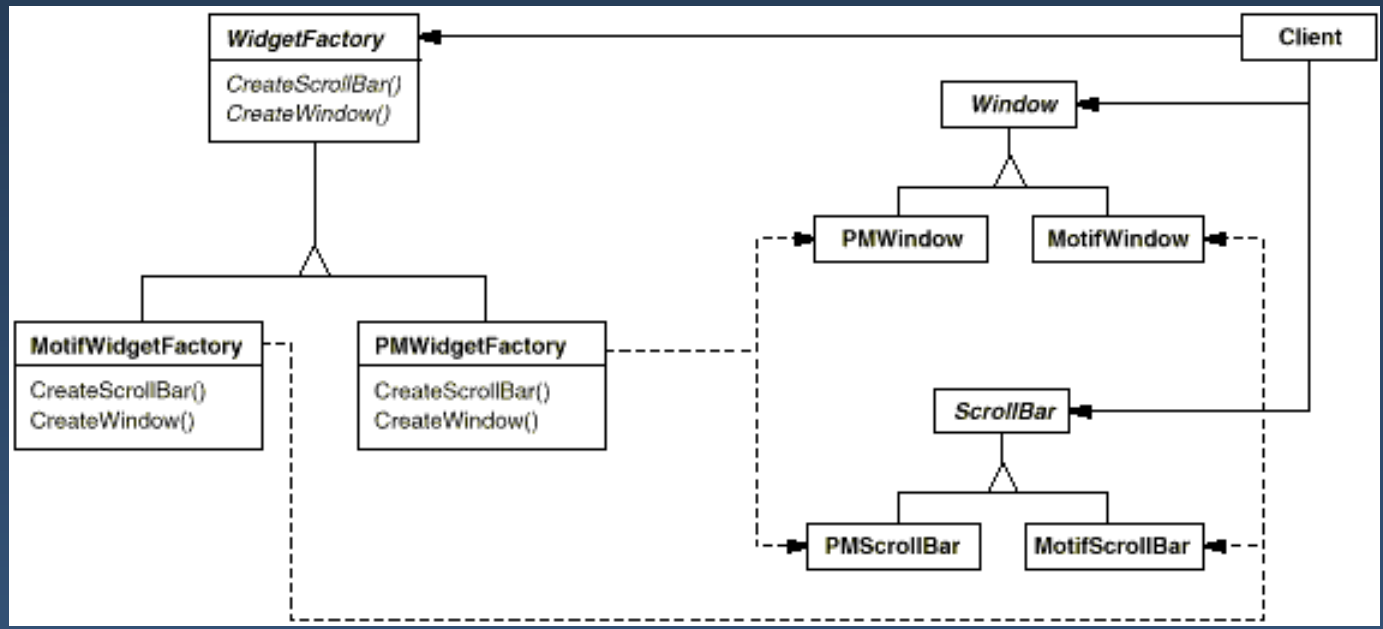
Implementation

- Lazy initialization
  - In C++, subclass vtable pointers aren't installed until after parent class initialization is complete.
    - DON'T CREATE DURING CONSTRUCTION!
    - can use lazy instatiation:

```
Product getProduct() {
    if( product == 0 ) {
        product = makeProduct();
    }
    return product;
}
```

Abstract Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- e.g., look-and-feel portability
  - independence
  - enforced consistency

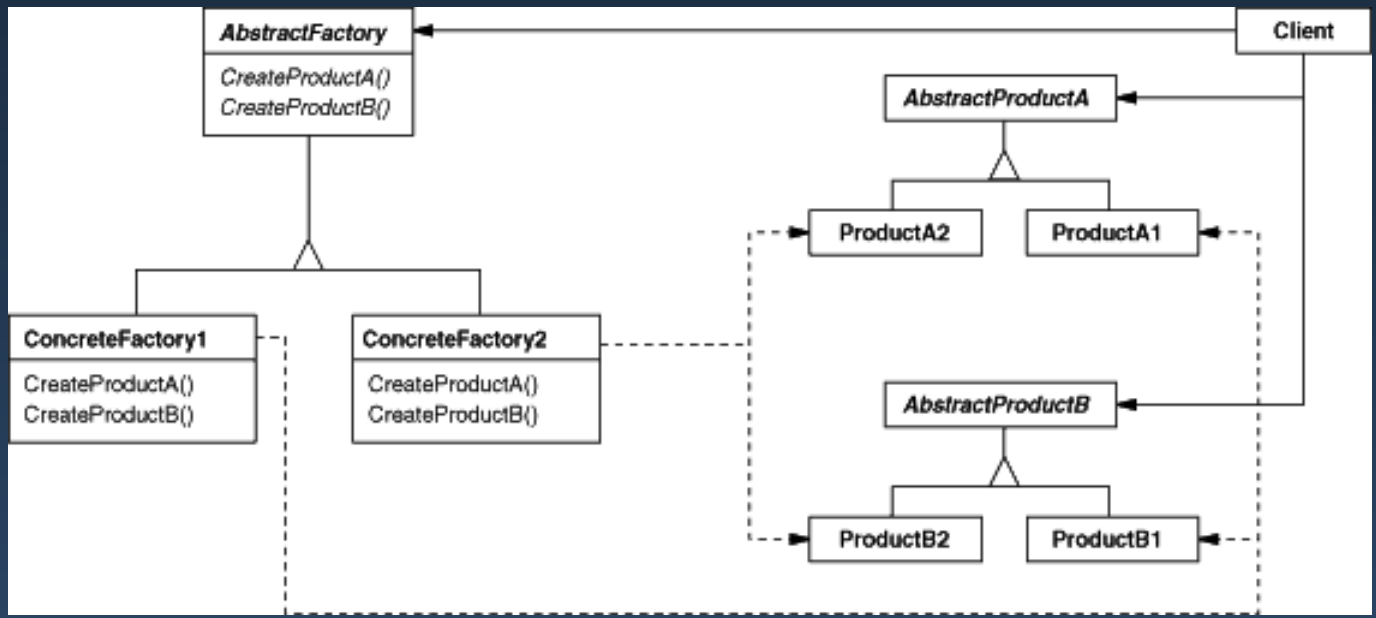


Applicability

Use when:

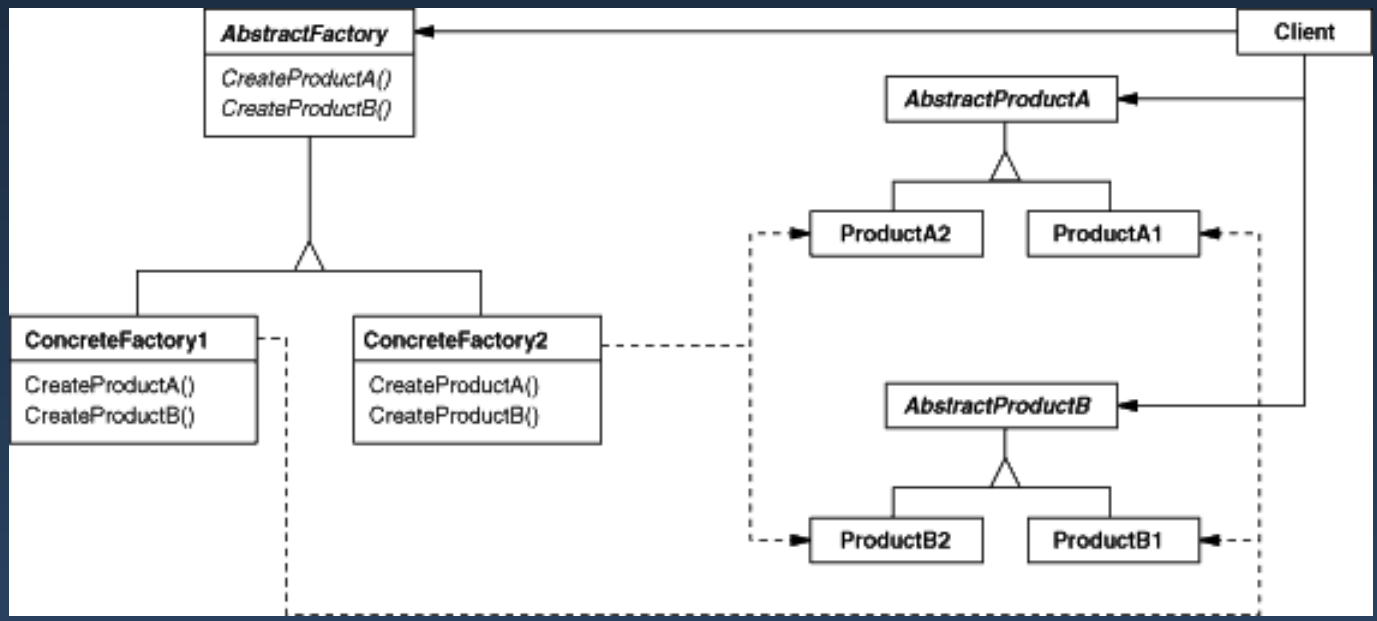
- a system should be independent of how its products are created, composed and represented
- a system should be configured with one of multiple families of products
- a family of related product objects is designed to be used together, and you need to enforce this constraint
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations
- you want to hide and reuse awkward or complex details of construction

Structure



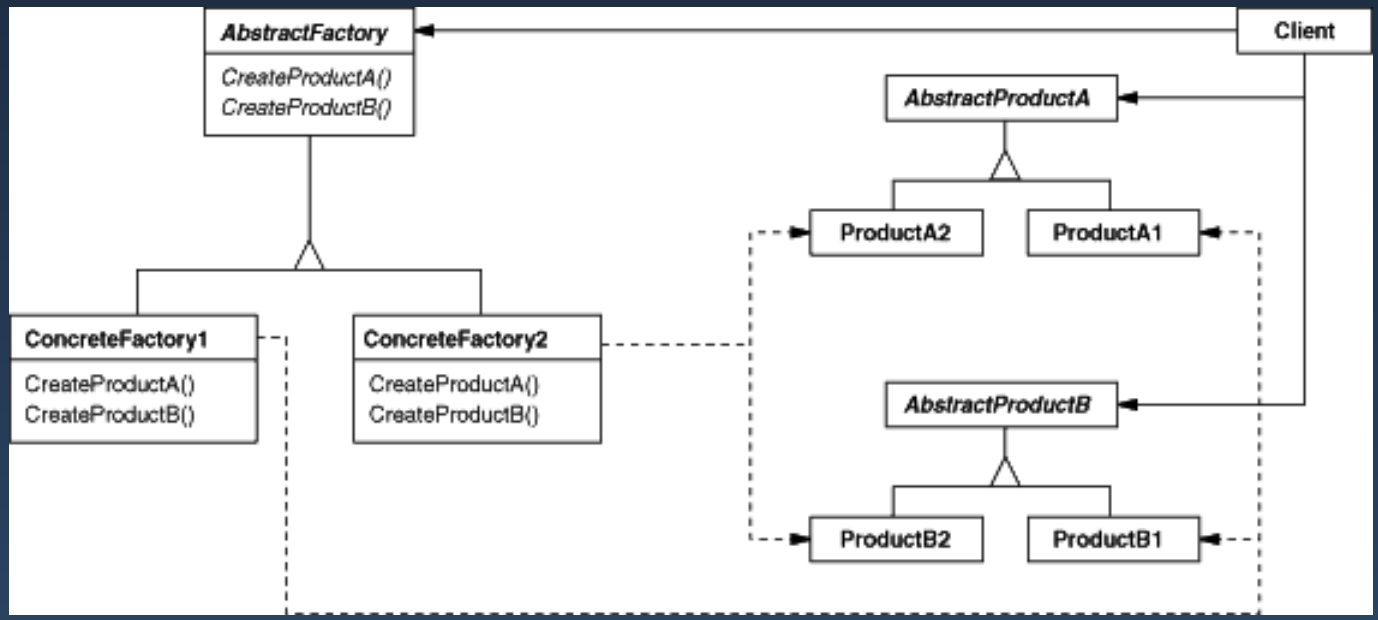
- AbstractFactory
  - declares an interface for operations that create product objects
- ConcreteFactory
  - implements the operations to create concrete product objects

Structure



- AbstractProduct
  - declares an interface for a type of product object
- Product
  - defines a product to be created by the corresponding concrete factory
  - implements the AbstractProduct interface

Structure



- Client
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes

Sample Code

```
class MazeFactory {
    Maze makeMaze() { return new Maze(); }
    Wall makeWall() { return new Wall(); }
    Room makeRoom(int r) { return new Room(r); }
    Door makeDoor(Room r1, Room r2)
        { return new Door(r1,r2); }
}
```

Consequences

- It isolates concrete classes
  - Helps control the classes of objects that an application creates
  - Isolates clients from implementation classes
  - Clients manipulate instances through abstract interfaces
  - Product classes are isolated in the implementation of the concrete factory
    - they do not appear in the client code
- It makes exchanging product families easy
  - The class of a concrete factory appears only once in the application
    - where it is instantiated
  - Easy to change the concrete factory an application uses
  - The whole product family changes at once
- It promotes consistency among products
  - When products are designed to work together, it is important that an application use objects only from one family at a time
  - AbstractFactory makes this easy to enforce

Consequences

Supporting new kinds of products is difficult

- Extending AbstractFactory to produce new product types is not easy
  - extend factory interface
  - extend all concrete factories
  - add a new abstract product
  - + the usual (implement new class in each family)
- Possible solution to extensible factories
  - Add parameter to operations that create products
    - need only `make()`
    - less safe
    - more flexible

Relation with other patterns

- Factory as a Singleton
  - An application typically needs only one instance of a ConcreteFactory per product family
  - Best implemented as a Singleton *(another design pattern)*

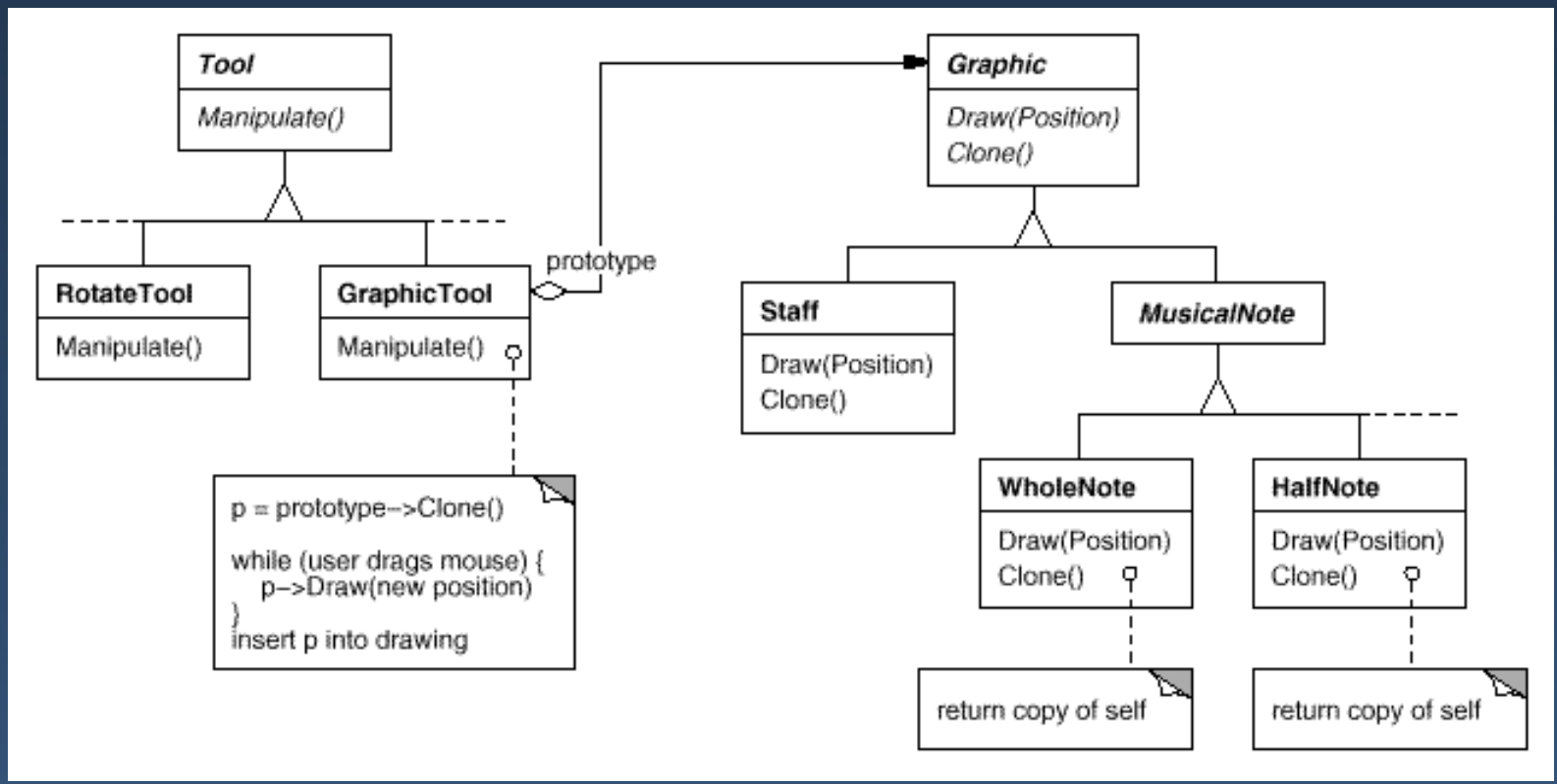
Relation with other patterns

Creating the products

- AbstractFactory declares an interface for product creation
- ConcreteFactory implements it. How?
  - Factory Method
    - virtual overrides for creation methods
    - simple
    - requires new concrete factories for each family, even if they only differ slightly
  - Prototype
    - concrete factory is initialized with a prototypical instance of each product in the family
    - creates new products by cloning
    - doesn't require a new concrete factory class for each product family
    - variant: can register class objects

Prototype

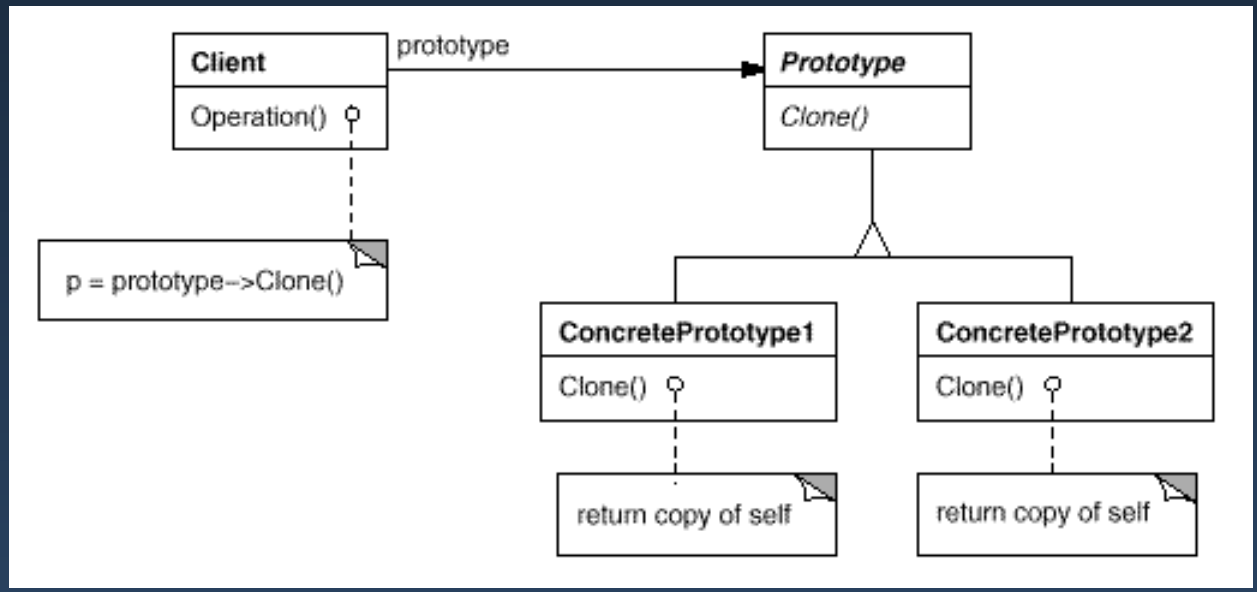
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
  - e.g., reduce # of classes (# of tools) by initializing a generic tool with a prototype



Applicability

- Use When:
  - the classes to be instantiated are specified at run-time
    - e.g., for dynamic loading
  - to avoid building a class hierarchy of factories to parallel the hierarchy of products
  - when instances can have only one of a few states
    - may be better to initialize once, and then clone prototypes

Structure



- Prototype
  - declares an interface for cloning itself
- ConcretePrototype
  - implements an operation for cloning itself
- Client
  - creates a new object by asking a prototype to clone itself

Consequences

Many of the same as AbstractFactory: can add and remove products at run-time

- new objects via new values
  - setting state on a prototype is analogous to defining a new class
- new structures
  - a multi-connected prototype + deep copy
- reducing subclassing
  - no need to have a factory or creator hierarchy
- dynamic load
  - cannot reference a new class's constructor statically
  - must register a prototype

- Disadvantage
  - implement clone() all over the place (can be boring…)

Singleton

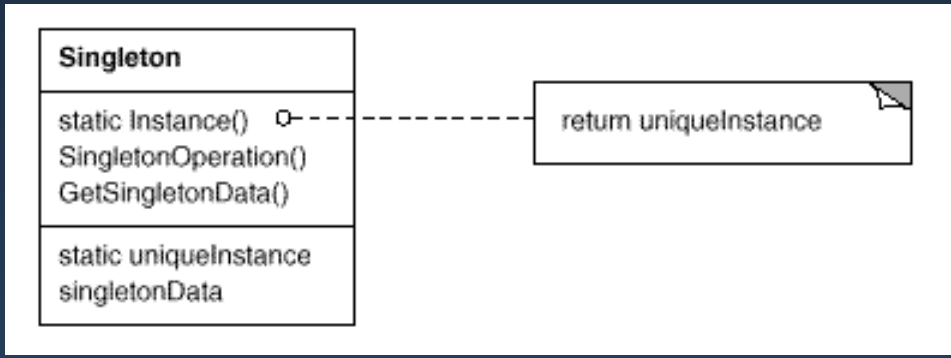
Ensure a class only has one instance, and provide a global point of access to it

- Many times need only one instance of an object
  - one RunManager
  - one EventManager
  - …
- How do we ensure there is exactly one instance, and that the instance is easily accessible?
  - Global variable is accessible, but can still instantiate multiple instances
  - NEVER USE GLOBAL VARIABLES
  - make the class itself responsible

Applicability

- Use when:
  - there must be exactly one instance accessible from a well-known access point
  - the sole instance should be extensible via subclassing
    - clients should be able to use the extended instance without modifying their code

Structure



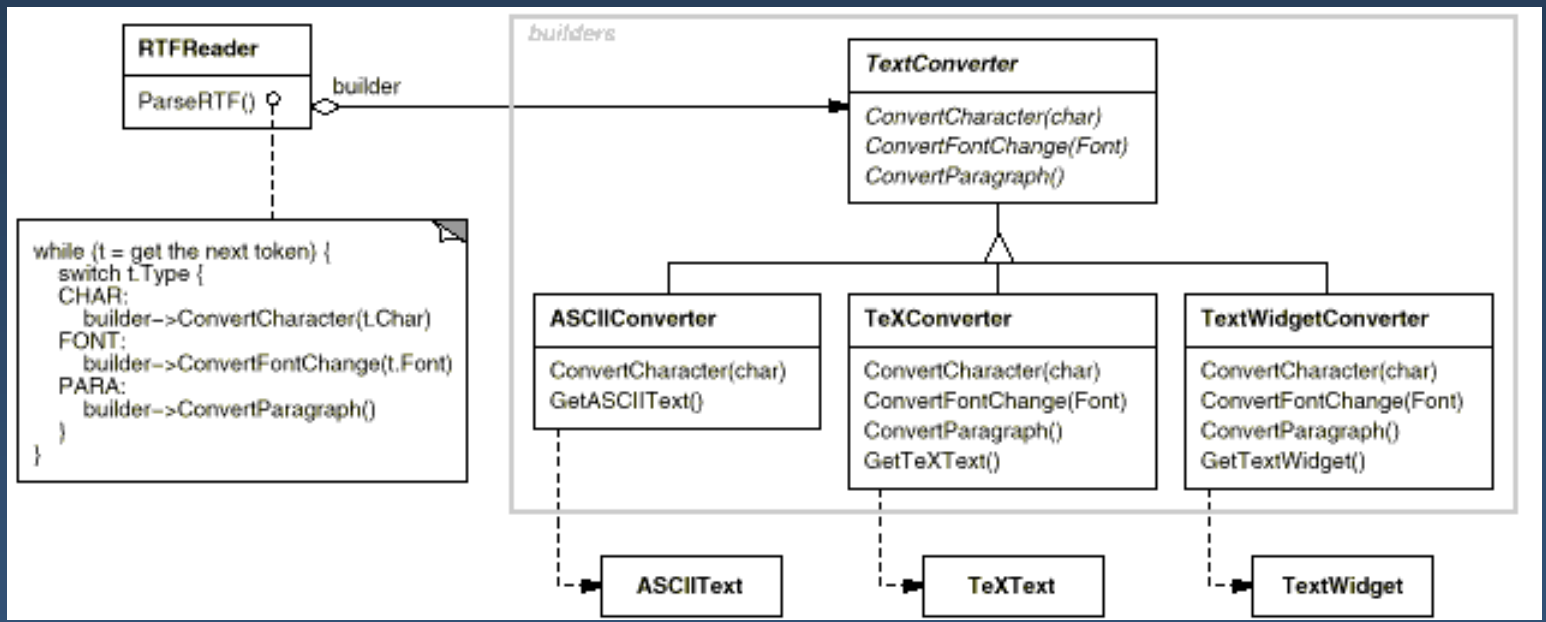
- Singleton
  - defines a class-scoped instance() operation that lets clients access its unique instance
  - may be responsible for creating its own unique instance

Consequences

- Controlled access to sole instance
  - Because singleton encapsulates the sole instance, it has strict control
- Reduced name space
  - one access method only
- Variable number of instances
  - easy to configure to have e.g., 5 instances
- Easy to derive and select new classes
  - access controlled through a single point of entry

Builder

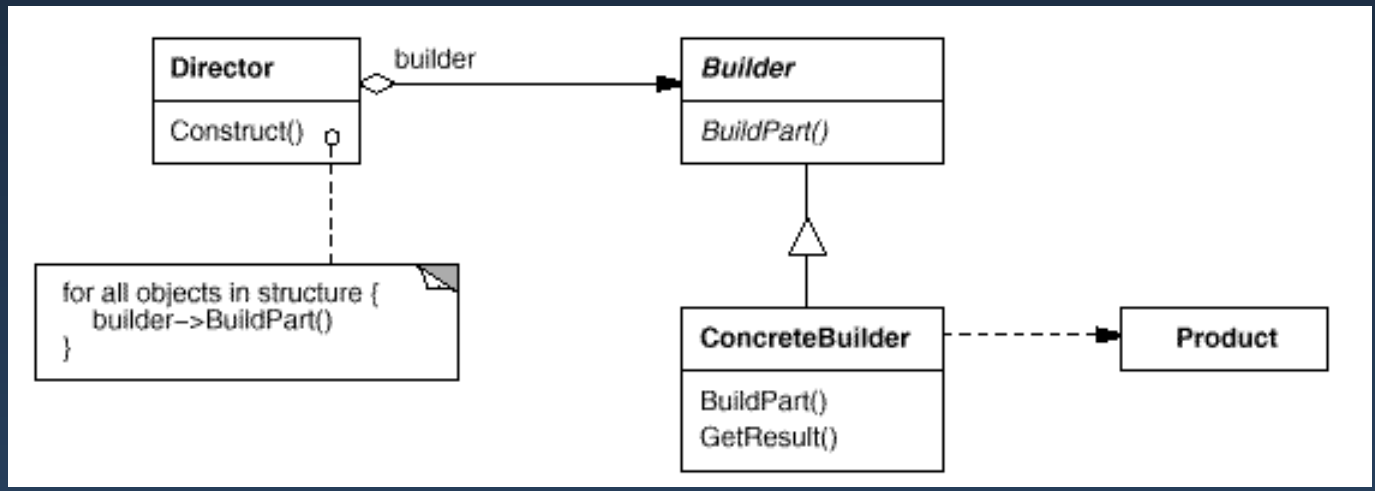
- Separate the construction of a complex object from its representation so that the same construction process can create different representations
  - e.g., read in Rich Text Format, converting to may different formats on load



Applicability

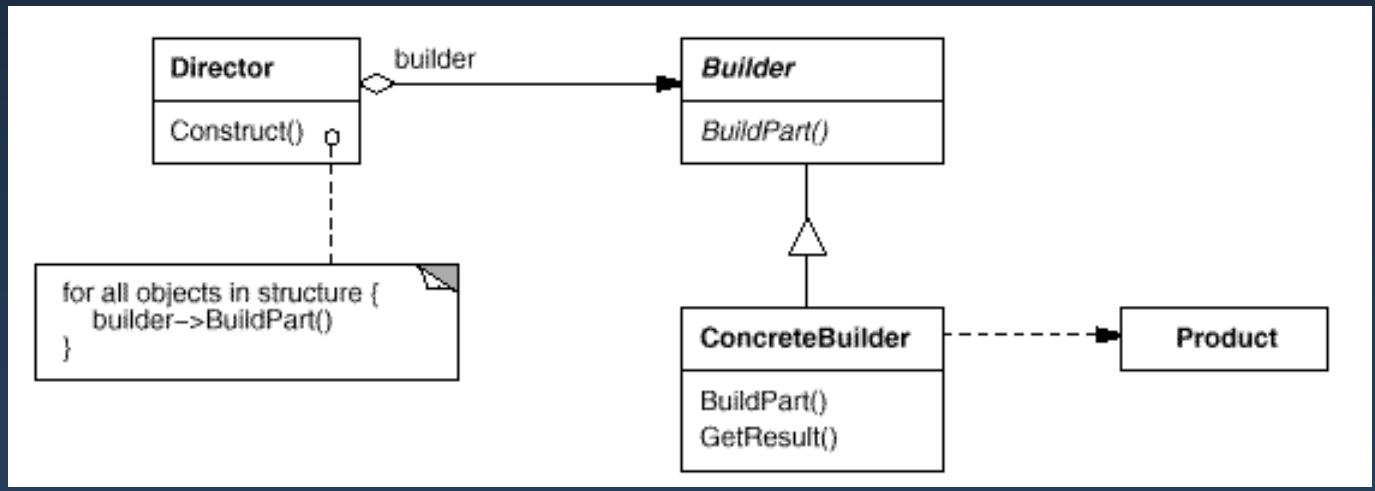
- Use When:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed

Structure



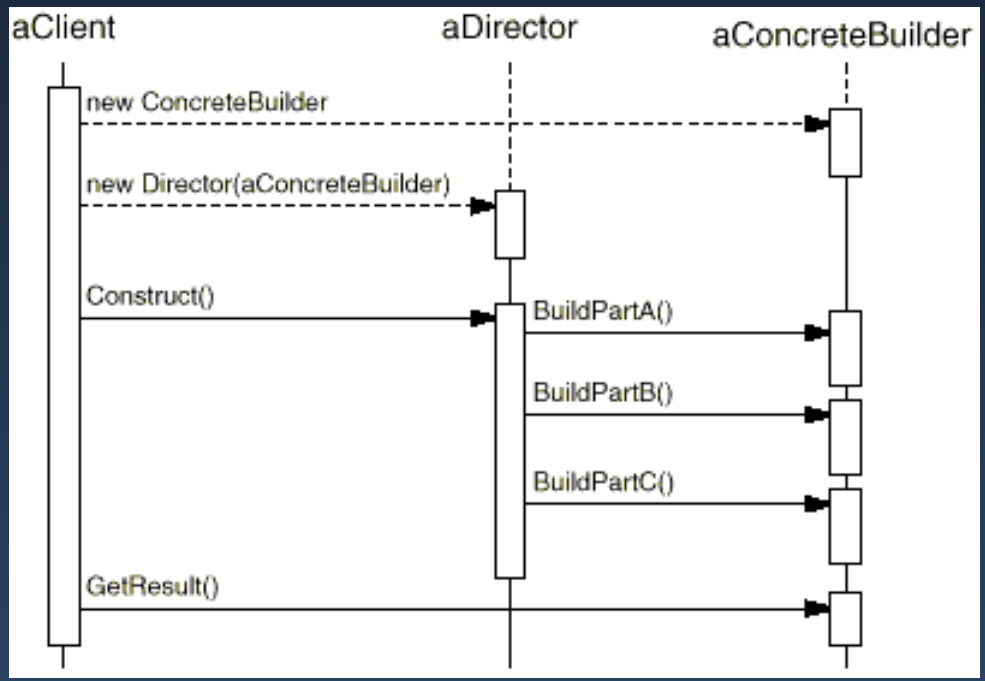
- Builder
  - specifies an abstract interface for creating parts of a Product object
- Concrete Builder
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product

Structure



- Director
  - constructs an object using the Builder interface
- Product
  - represents the complex object under construction.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Collaborations



1. The client creates the Director object and configures it with the Builder object
2. Director notifies the builder whenever a part of the product should be built
3. Builder handles requests from the director and adds parts to the product
4. The client retrieves the product from the builder

Consequences

- Lets you vary a product's internal representation
- Isolates code for construction and representation
- Gives you control over the construction process

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method (class scoped)
- If createMaze() is passed a parameter object to create rooms, walls, …
  - Abstract Factory
- If createMaze() is passed a parameter object to create and connect-up mazes
  - Builder
- If createMaze is parameterized with various prototypical rooms, doors, walls, … which it copies and then adds to the maze
  - Prototype
- Need to ensure there is only one maze per game, and everybody can access it, and can extend or replace the maze without touching other code.
  - Singleton