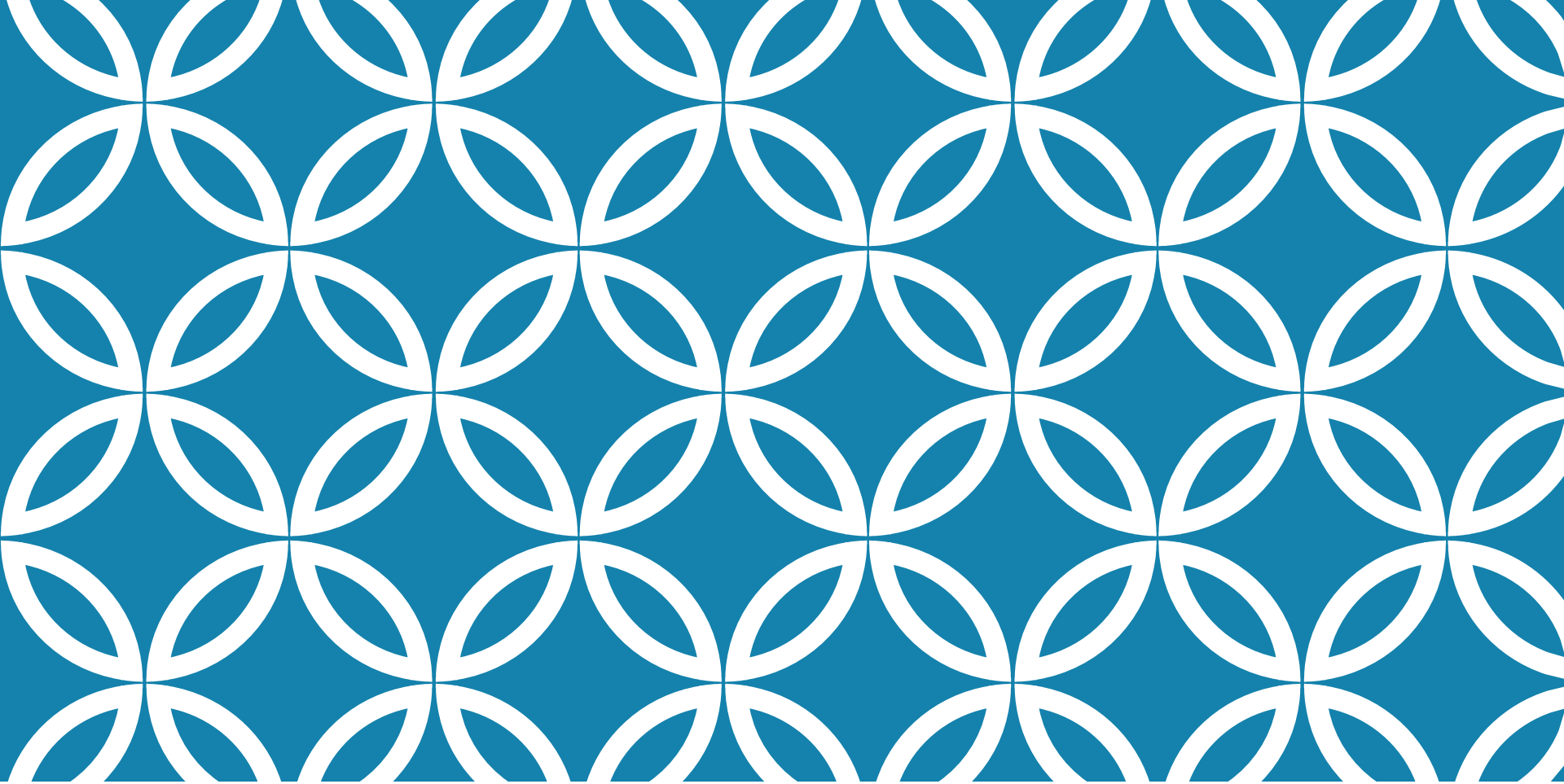


CREATIONAL DESIGN PATTERNS

Lecture 9

CONTENT (OF THE NEXT LECTURES)

Design Patterns

- Creational Patterns
  - Factory Method
  - Prototype
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Composite
  - Decorator
  - Proxy
  - Bridge
- Behavioral Patterns

REFERENCES

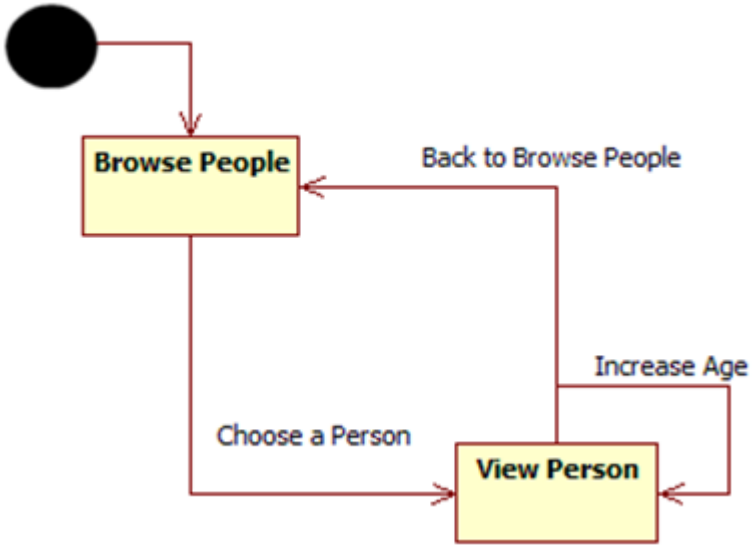
- **Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.**

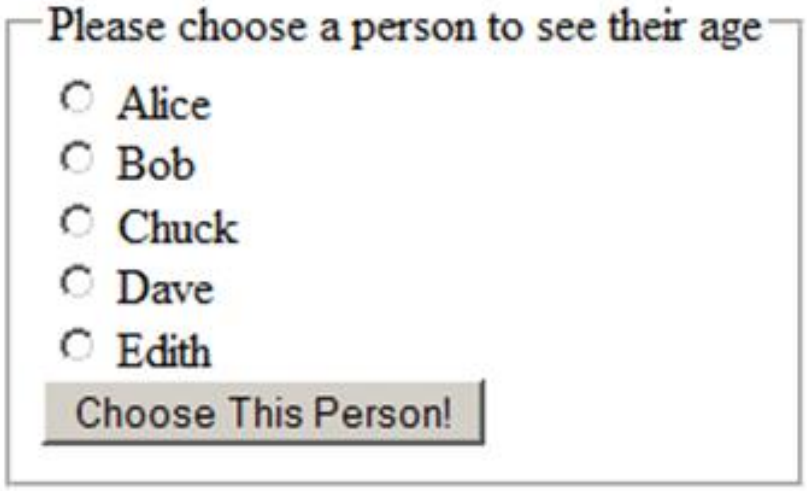
- Stuart Thiel, Enterprise Application Design Patterns: Improved and Applied, MSc Thesis, Concordia University Montreal, Quebec, Canada [Thiel]

RECAP LAYERED SOLUTIONS [THIEL]

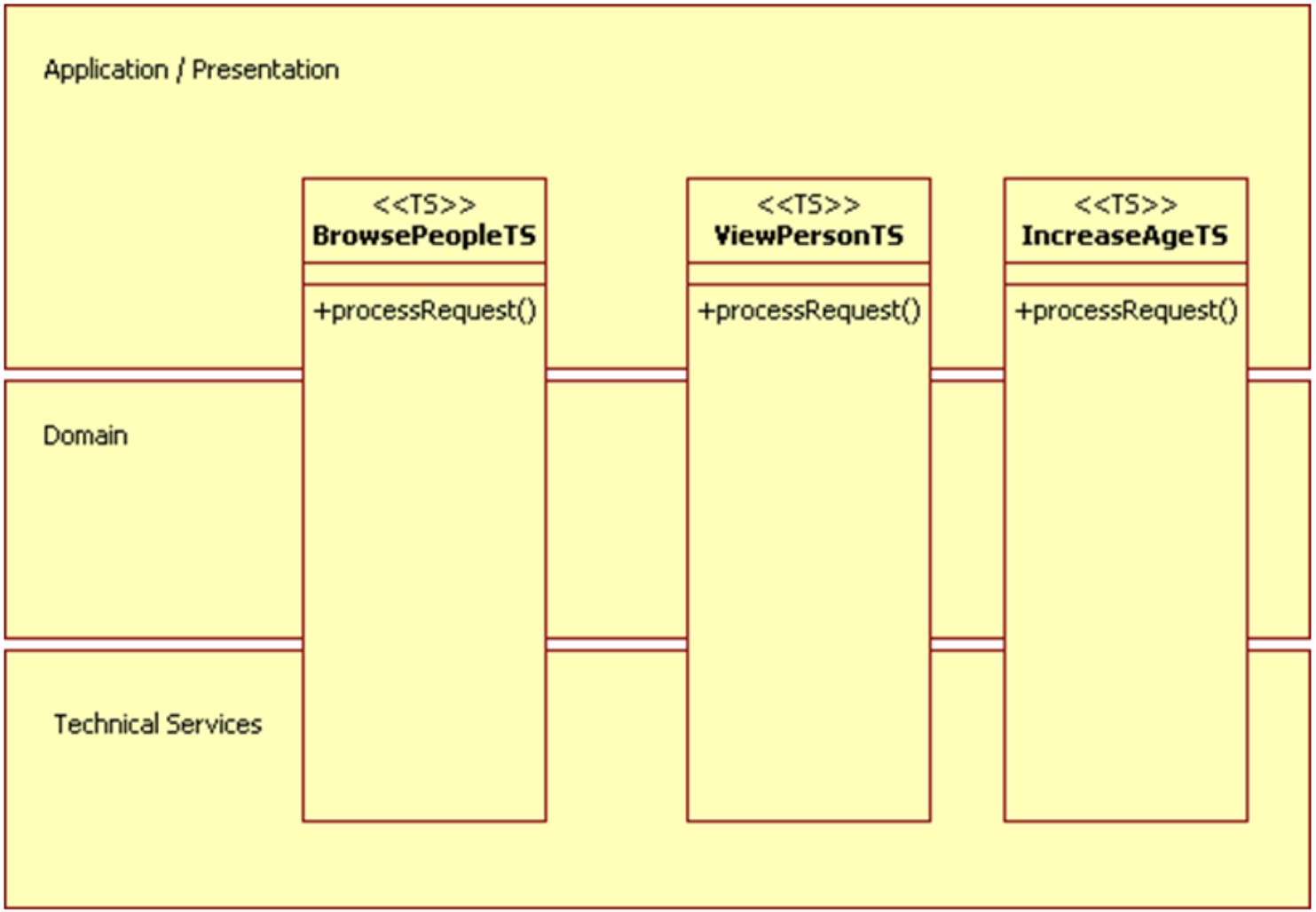
People management web app

- Browse people
- View person
- Change person data (ex. Increase age)

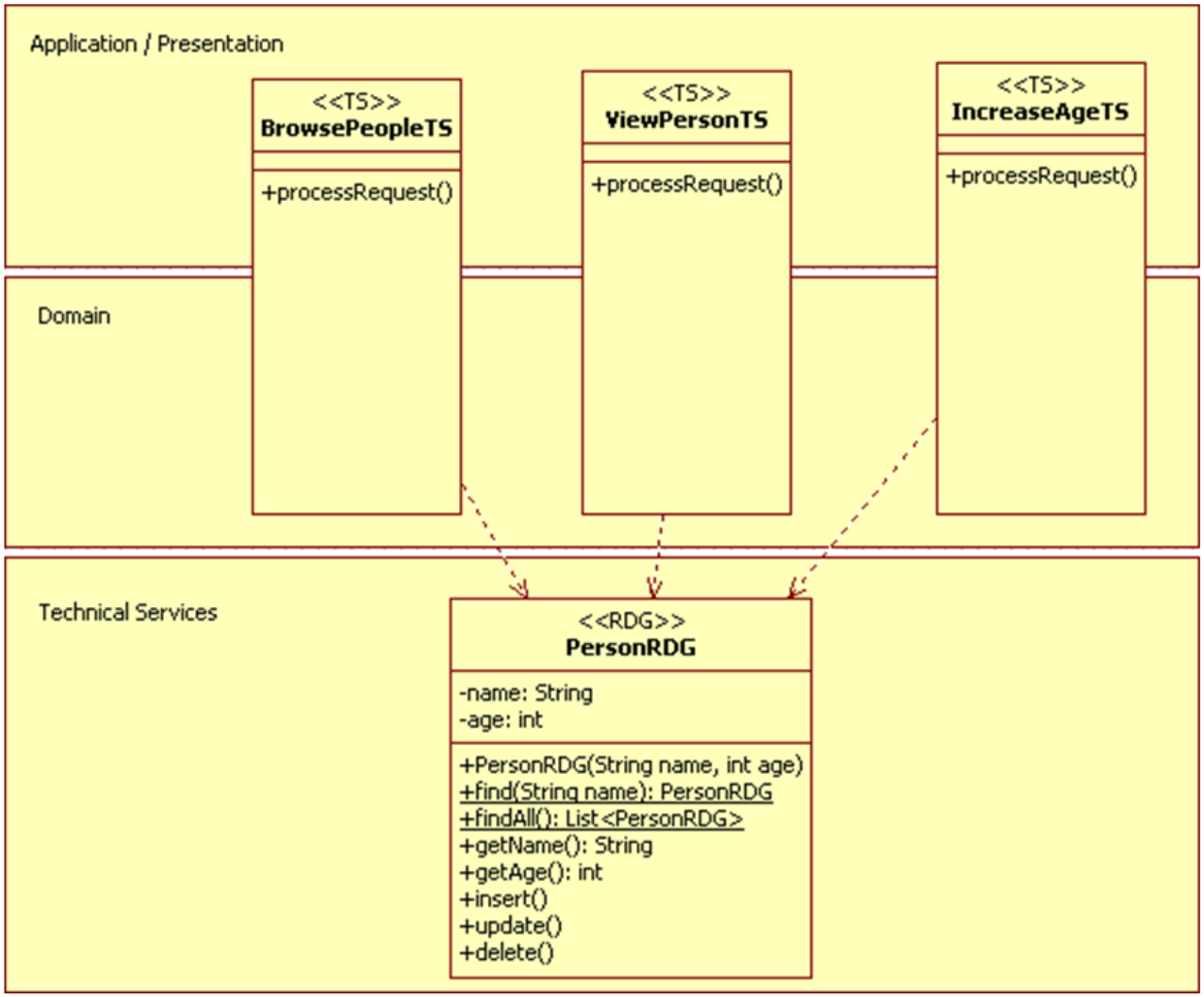




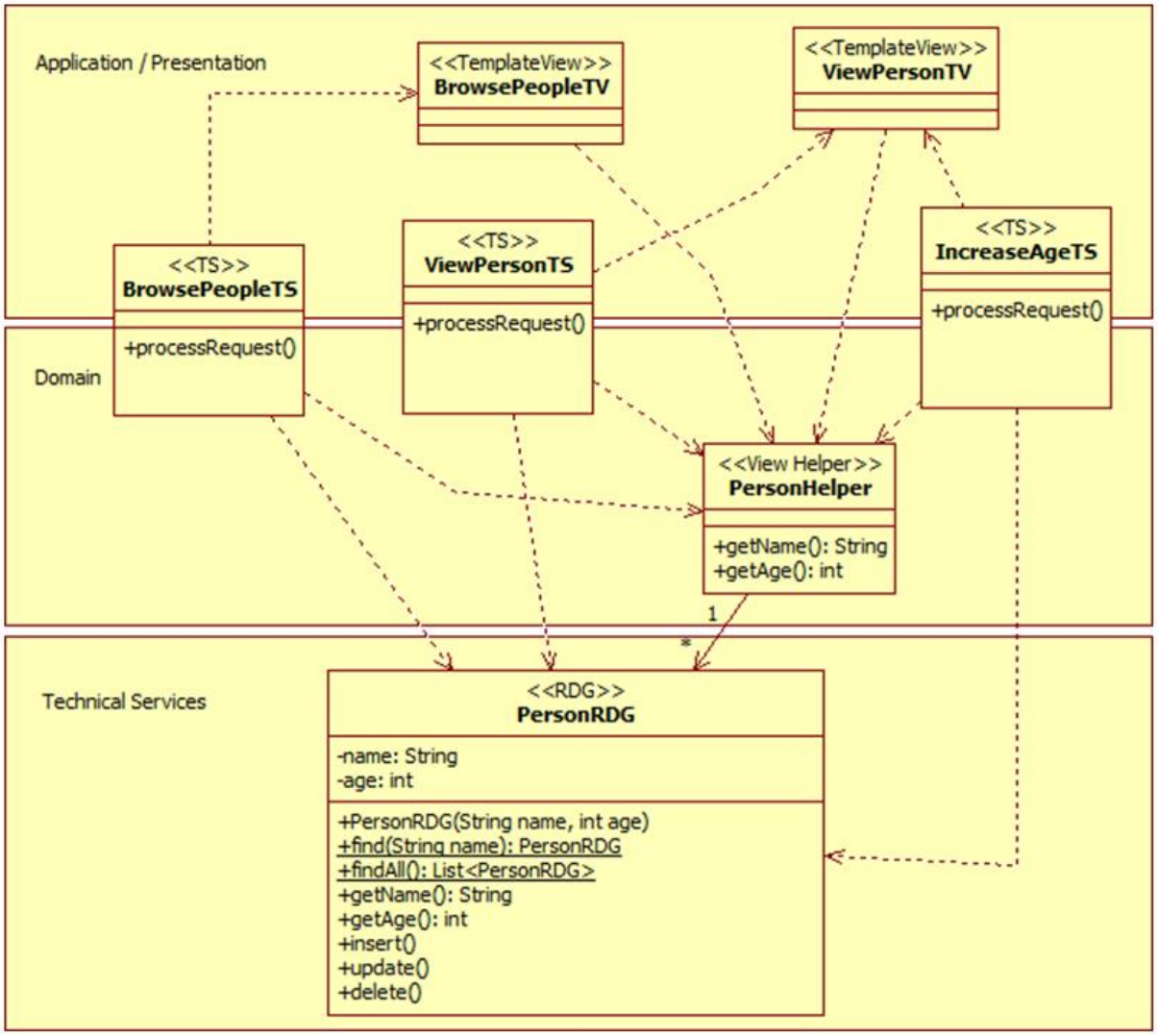
JUST TRANSACTION SCRIPT (FUNCTIONAL DECOMPOSITION)



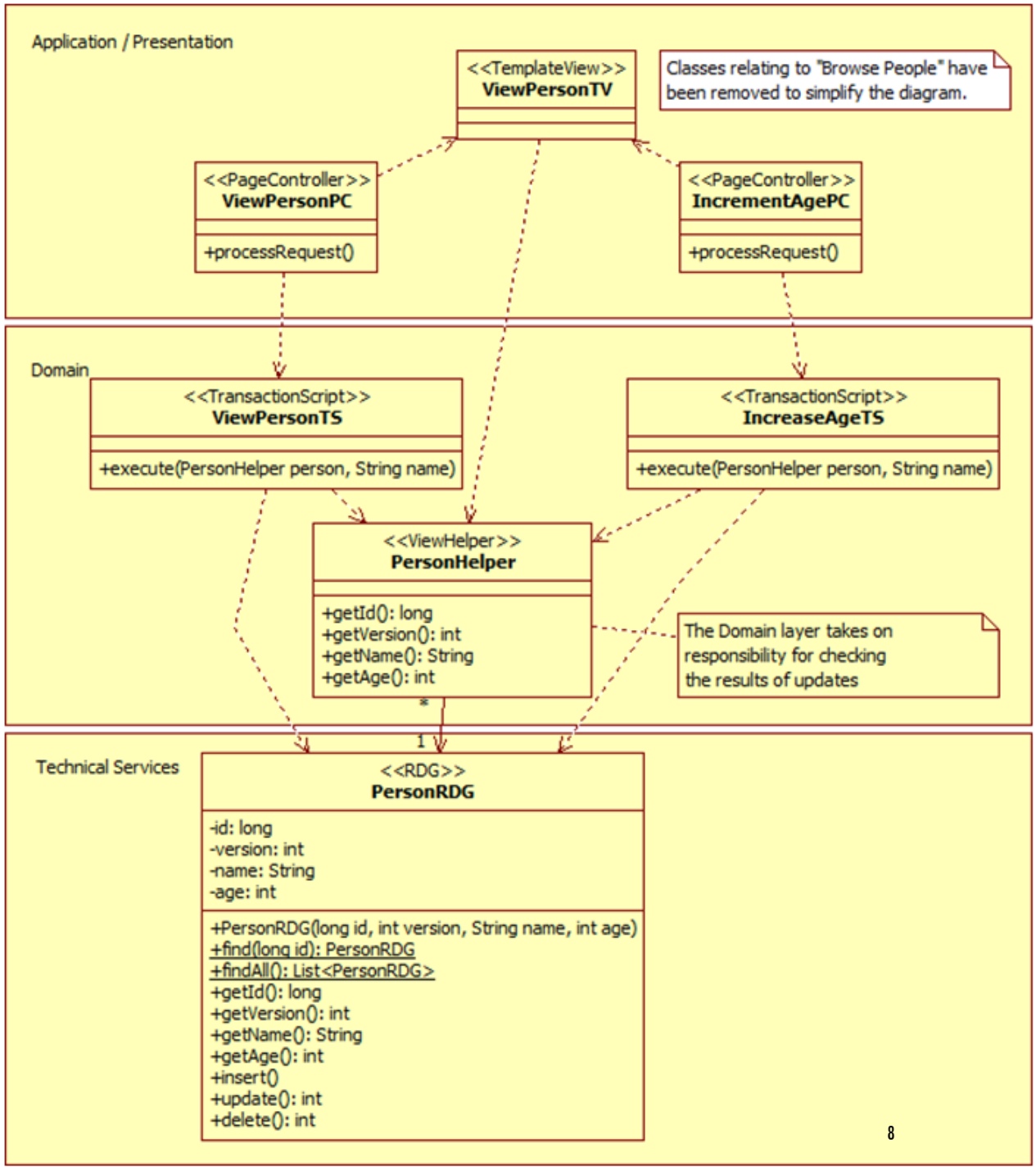
TS + RDG (DAO)


Application / Presentation
<<TS>>
BrowsePeopleTS
+processRequest()
<<TS>>
ViewPersonTS
+processRequest()
<<TS>>
IncreaseAgeTS
+processRequest()
Domain
Technical Services
<<RDG>>
PersonRDG
-name: String
-age: int
+PersonRDG(String name, int age)
+find(String name): PersonRDG
+findAll(): List<PersonRDG>
+getName(): String
+getAge(): int
+insert()
+update()
+delete()

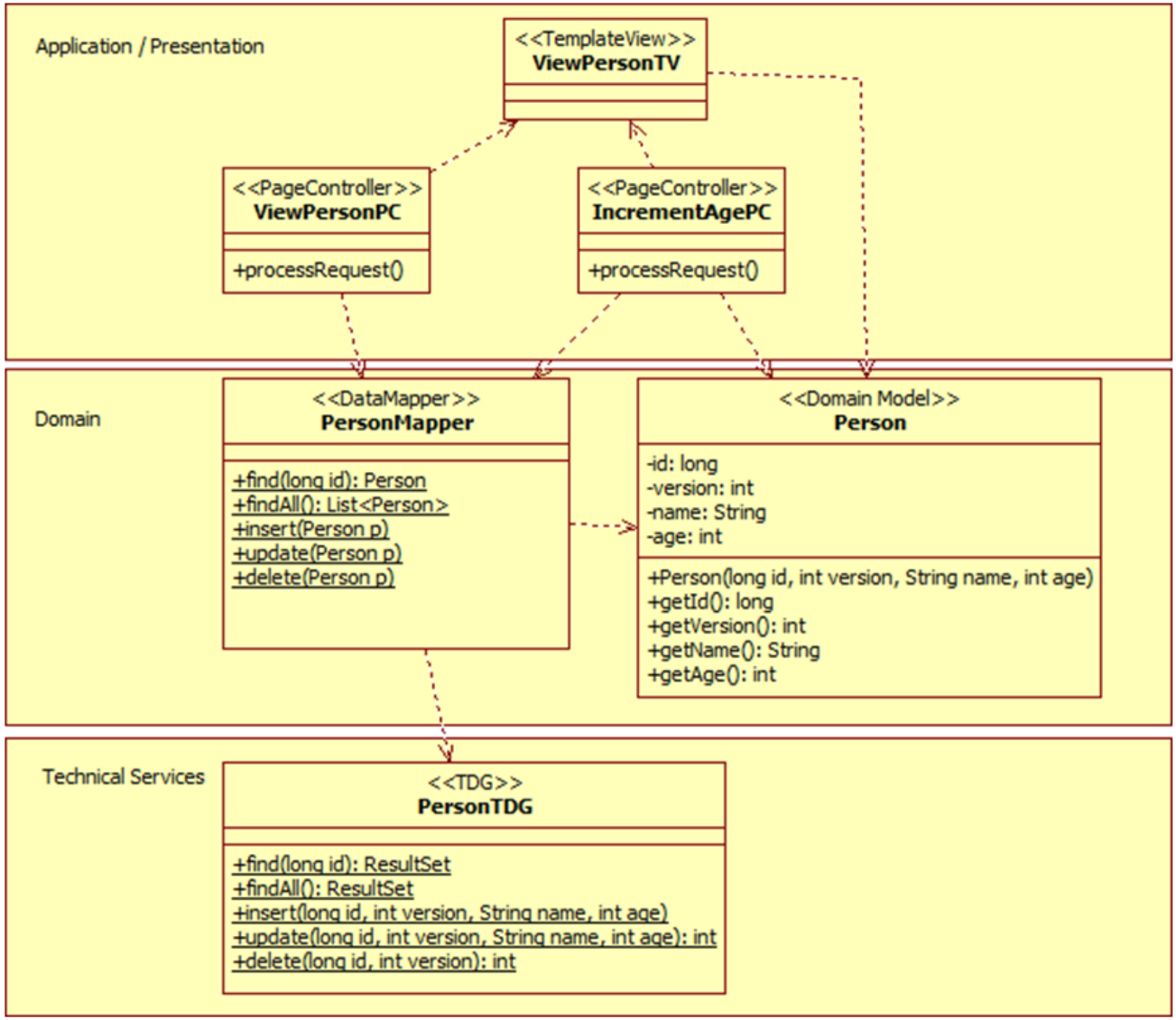
TS + RDG + TV


Application / Presentation
<<TemplateView>>
BrowsePeopleTV
<<TemplateView>>
ViewPersonTV
<<TS>>
BrowsePeopleTS
+processRequest()
<<TS>>
ViewPersonTS
+processRequest()
<<TS>>
IncreaseAgeTS
+processRequest()
Domain
<<View Helper>>
PersonHelper
+getName(): String
+getAge(): int
1
*
Technical Services
<<RDG>>
PersonRDG
-name: String
-age: int
+PersonRDG(String name, int age)
+find(String name): PersonRDG
+findAll(): List<PersonRDG>
+getName(): String
+getAge(): int
+insert()
+update()
+delete()

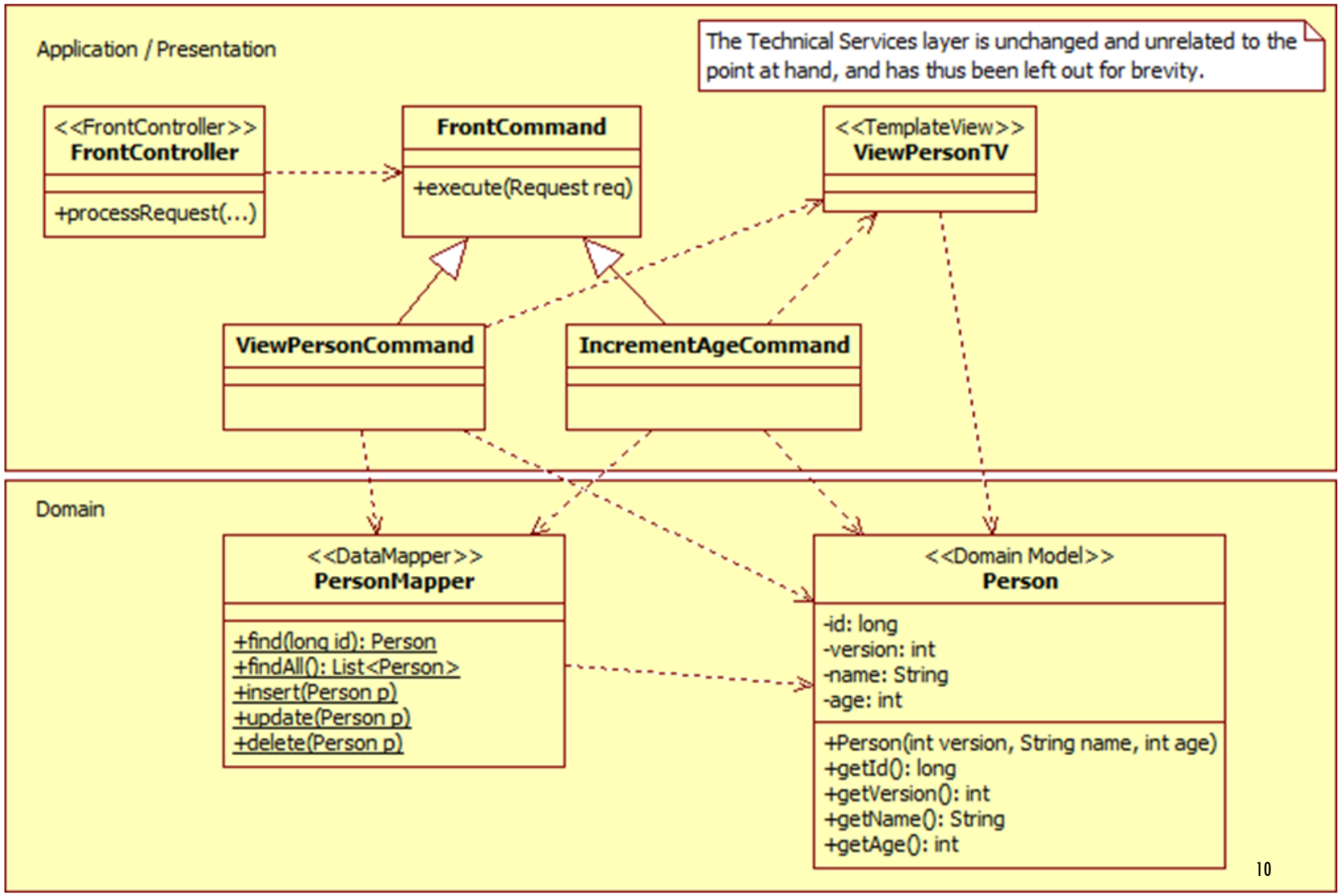
TS+RDG +TV+PC


Application / Presentation
<<TemplateView>>
ViewPersonTV
Classes relating to "Browse People" have been removed to simplify the diagram.
<<PageController>>
ViewPersonPC
+processRequest()
<<PageController>>
IncrementAgePC
+processRequest()
Domain
<<TransactionScript>>
ViewPersonTS
+execute(PersonHelper person, String name)
<<TransactionScript>>
IncreaseAgeTS
+execute(PersonHelper person, String name)
<<ViewHelper>>
PersonHelper
+getId(): long
+getVersion(): int
+getName(): String
+getAge(): int
The Domain layer takes on responsibility for checking the results of updates
*
1
Technical Services
<<RDG>>
PersonRDG
-id: long
-version: int
-name: String
-age: int
+PersonRDG(long id, int version, String name, int age)
+find(long id): PersonRDG
+findAll(): List<PersonRDG>
+getId(): long
+getVersion(): int
+getName(): String
+getAge(): int
+insert()
+update(): int
+delete(): int

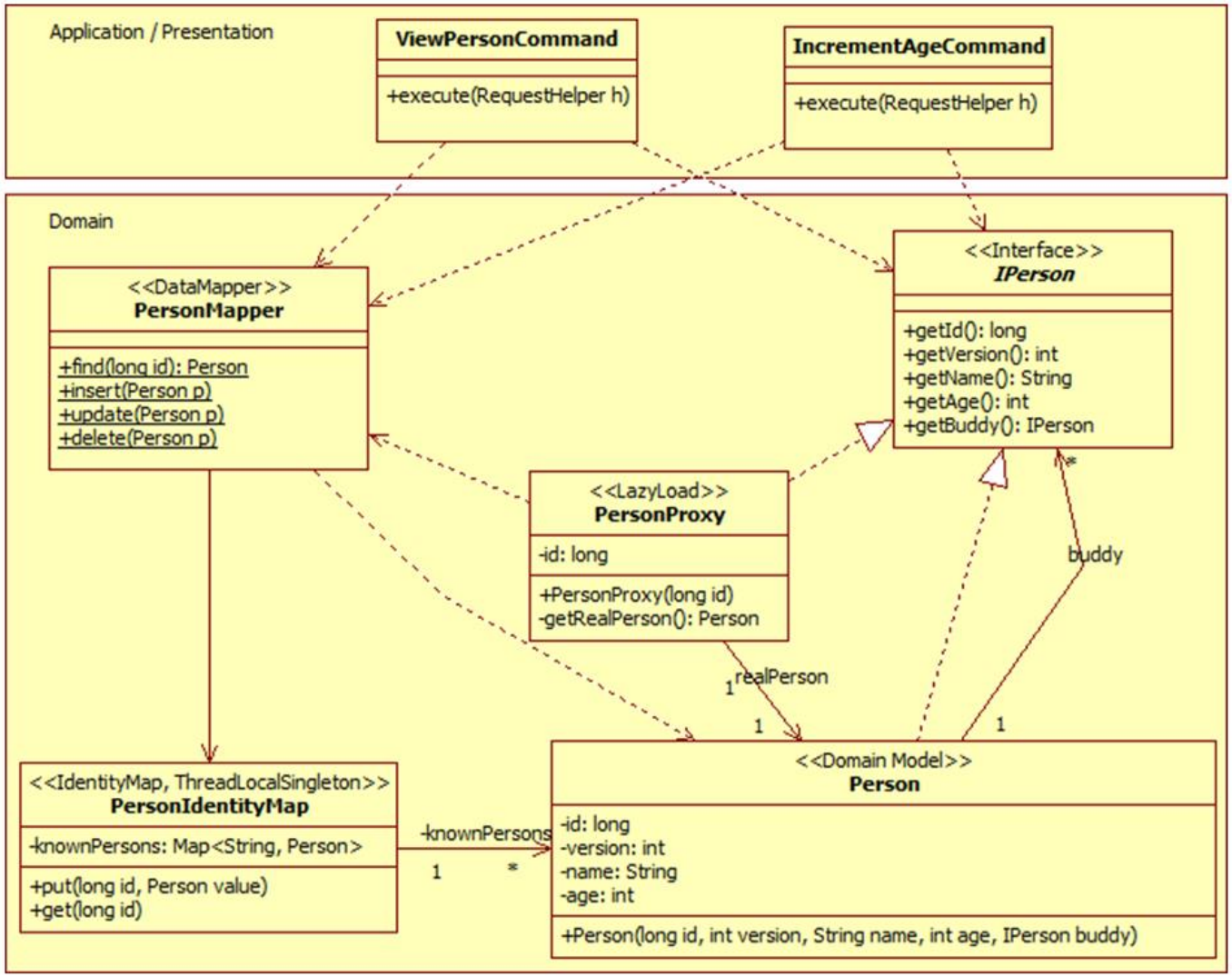
DM+TDG+DMAPPER+TV+PC

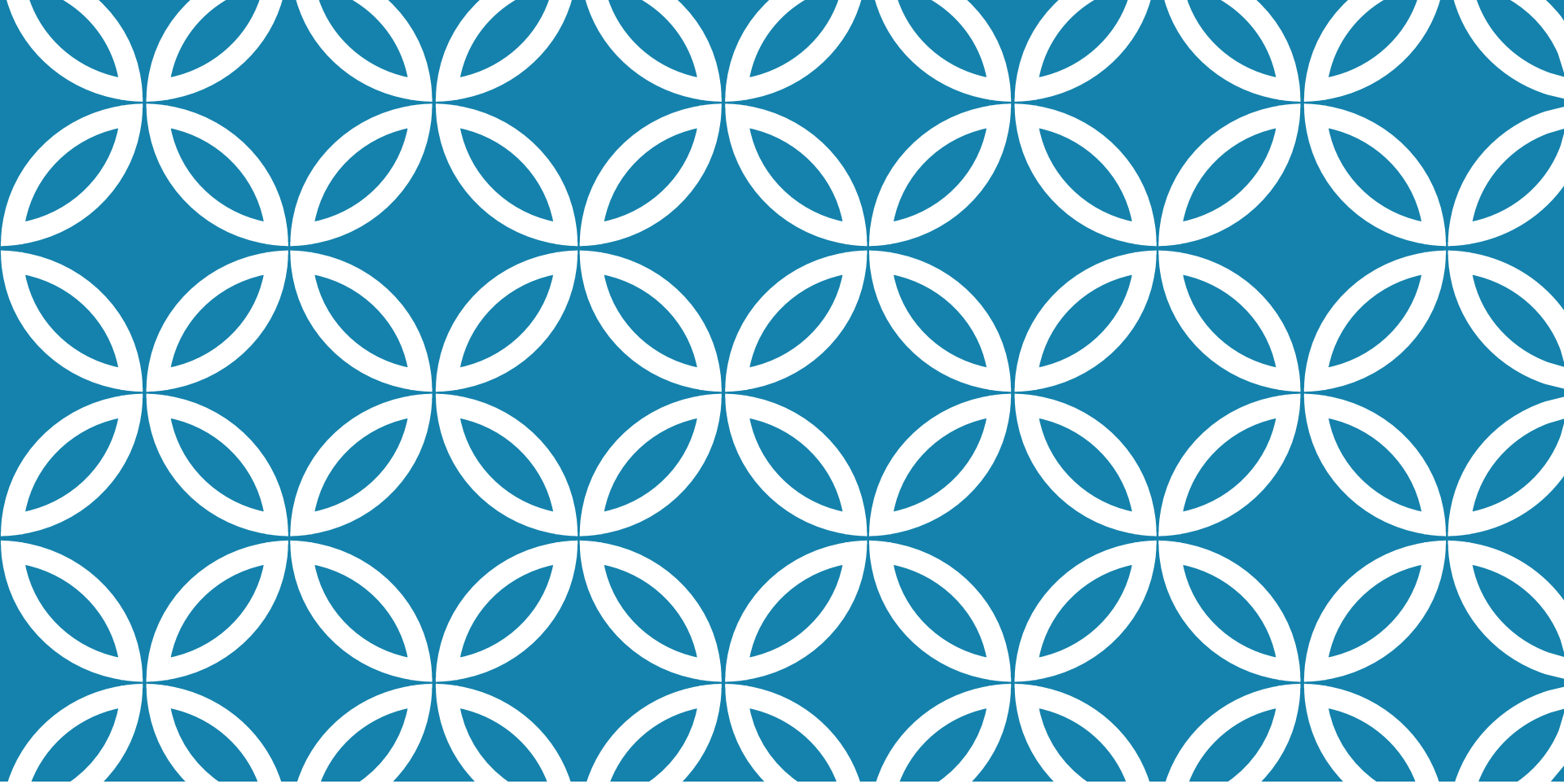


...OR DM+...+FC


Application / Presentation
The Technical Services layer is unchanged and unrelated to the point at hand, and has thus been left out for brevity.
<<FrontController>>
FrontController
+processRequest(...)
FrontCommand
+execute(Request req)
<<TemplateView>>
ViewPersonTV
ViewPersonCommand
IncrementAgeCommand
Domain
<<DataMapper>>
PersonMapper
+find(long id): Person
+findAll(): List<Person>
+insert(Person p)
+update(Person p)
+delete(Person p)
<<Domain Model>>
Person
-id: long
-version: int
-name: String
-age: int
+Person(int version, String name, int age)
+getId(): long
+getVersion(): int
+getName(): String
+getAge(): int

...+ LAZY LOAD + IM


Application / Presentation
ViewPersonCommand
+execute(RequestHelper h)
IncrementAgeCommand
+execute(RequestHelper h)
Domain
<<DataMapper>>
PersonMapper
+find(long id): Person
+insert(Person p)
+update(Person p)
+delete(Person p)
<<Interface>>
IPerson
+getId(): long
+getVersion(): int
+getName(): String
+getAge(): int
+getBuddy(): IPerson
<<LazyLoad>>
PersonProxy
-id: long
+PersonProxy(long id)
-getRealPerson(): Person
buddy
*
1
realPerson
1
1
<<IdentityMap, ThreadLocalSingleton>>
PersonIdentityMap
-knownPersons: Map<String, Person>
+put(long id, Person value)
+get(long id)
-knownPersons
1
*
<<Domain Model>>
Person
-id: long
-version: int
-name: String
-age: int
+Person(long id, int version, String name, int age, IPerson buddy)



DESIGN PATTERNS

CLASSIFICATION OF DESIGN PATTERNS

Creational Patterns

- deal with initializing and configuring classes and objects
- *how am I going to create my objects?*

Structural Patterns

- deal with decoupling the interface and implementation of classes and objects
- *how classes and objects are composed to build larger structures*

Behavioral Patterns

- deal with dynamic interactions among societies of classes and objects
- *how to manage complex control flows (communications)*

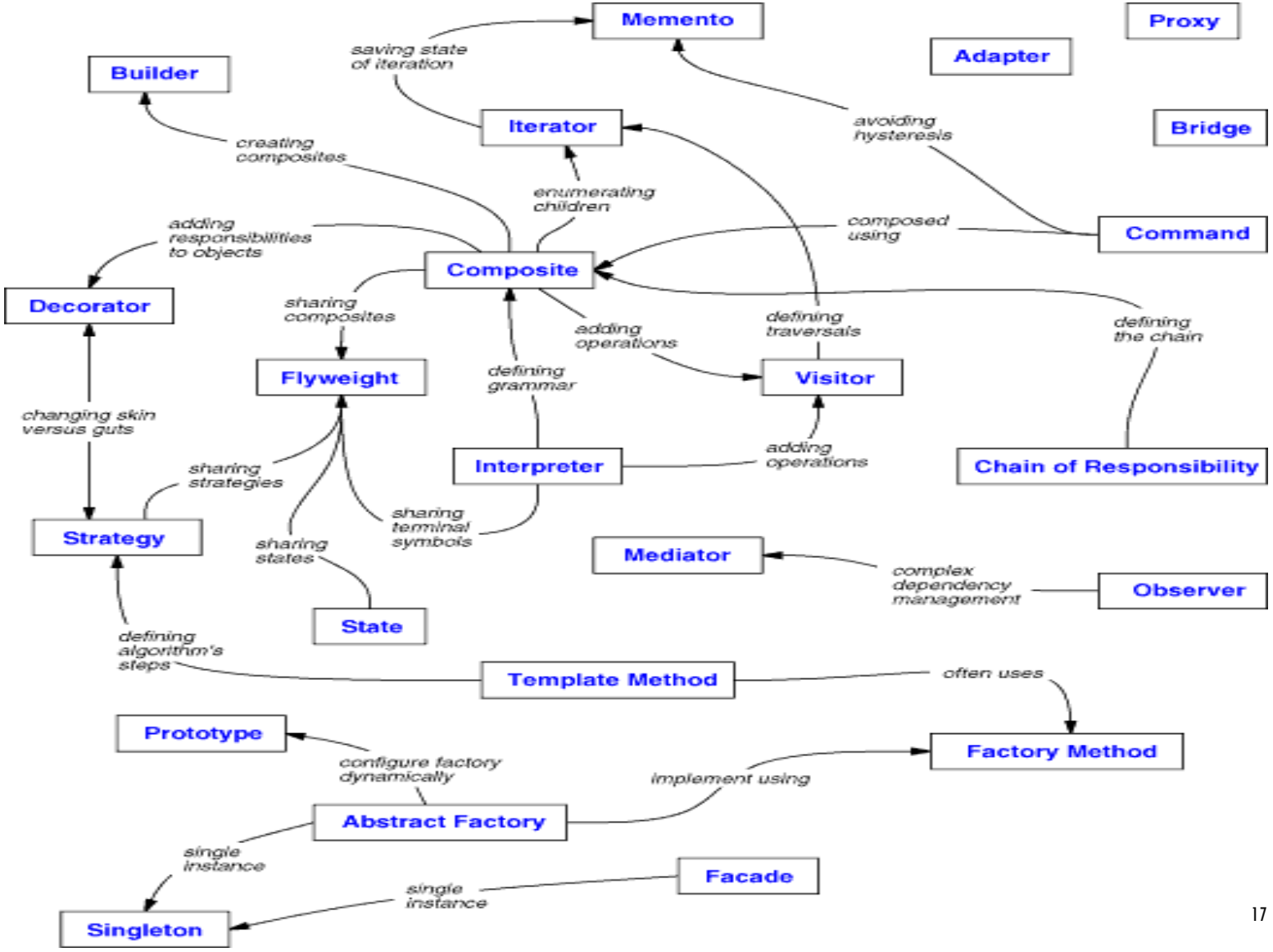
DRAWBACKS OF DESIGN PATTERNS

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from patterns overload
- Integrating patterns into a software development process is a human-intensive activity

DESIGN PATTERNS SPACE

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

DP RELATIONSHIPS



CREATIONAL DP

- Abstract instantiation process
- System independent of how objects are created, composed, and represented

- **Class** creational patterns use inheritance to vary class instantiation
- **Object** creational patterns delegate instantiation to another object

- Focus on defining small behaviors and combining into more complex ones

RECURRING THEMES

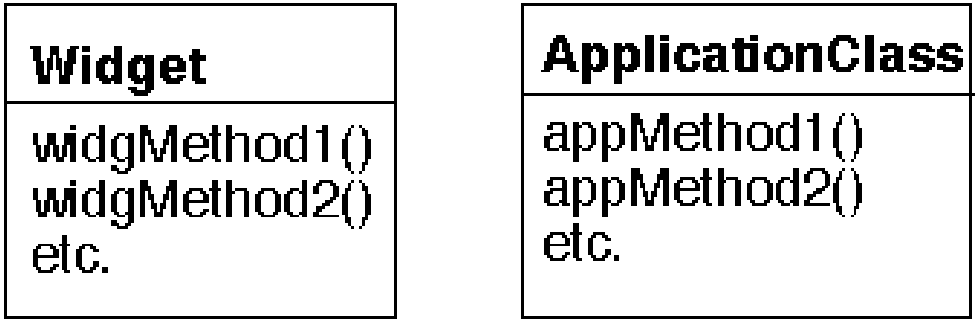
- Encapsulate knowledge about which concrete classes the system uses

- Hide how instances of these classes are created and put together

ADVANTAGES

- Flexibility in what gets created
  - *Who* creates it
  - *How* and *when* it creates it
- Configure system with "product" objects that vary in structure and functionality
- Configuration can be static or dynamic
- Create standard or uniform structure

LET'S START SIMPLE…




```
class ApplicationClass {
      public appMethod1() { ..
            Widget w = new Widget();….
      }…}
```

We can modify the internal **`Widget`** code without modifying **`ApplicationClass`**

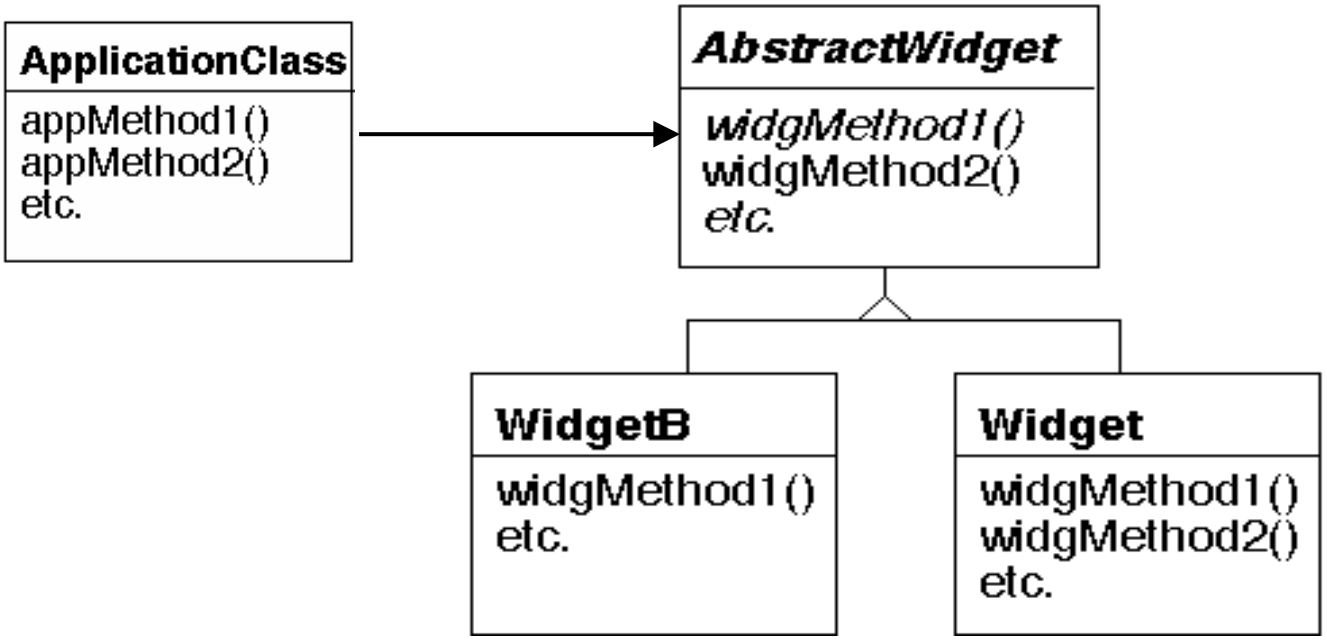
PROBLEMS WITH CHANGES

What happens when we discover a new widget and would like to use in the **`ApplicationClass`**?

Multiple coupling between **`Widget`** and **`ApplicationClass`**

- **`ApplicationClass`** knows the interface of **`Widget`**
- **`ApplicationClass`** explicitly uses the **`Widget`** type
- hard to change because **`Widget`** is a concrete class
- **`ApplicationClass`** explicitly creates new Widgets in many places
- if we want to use the new **`Widget`** instead of the initial one, changes are spread all over the code

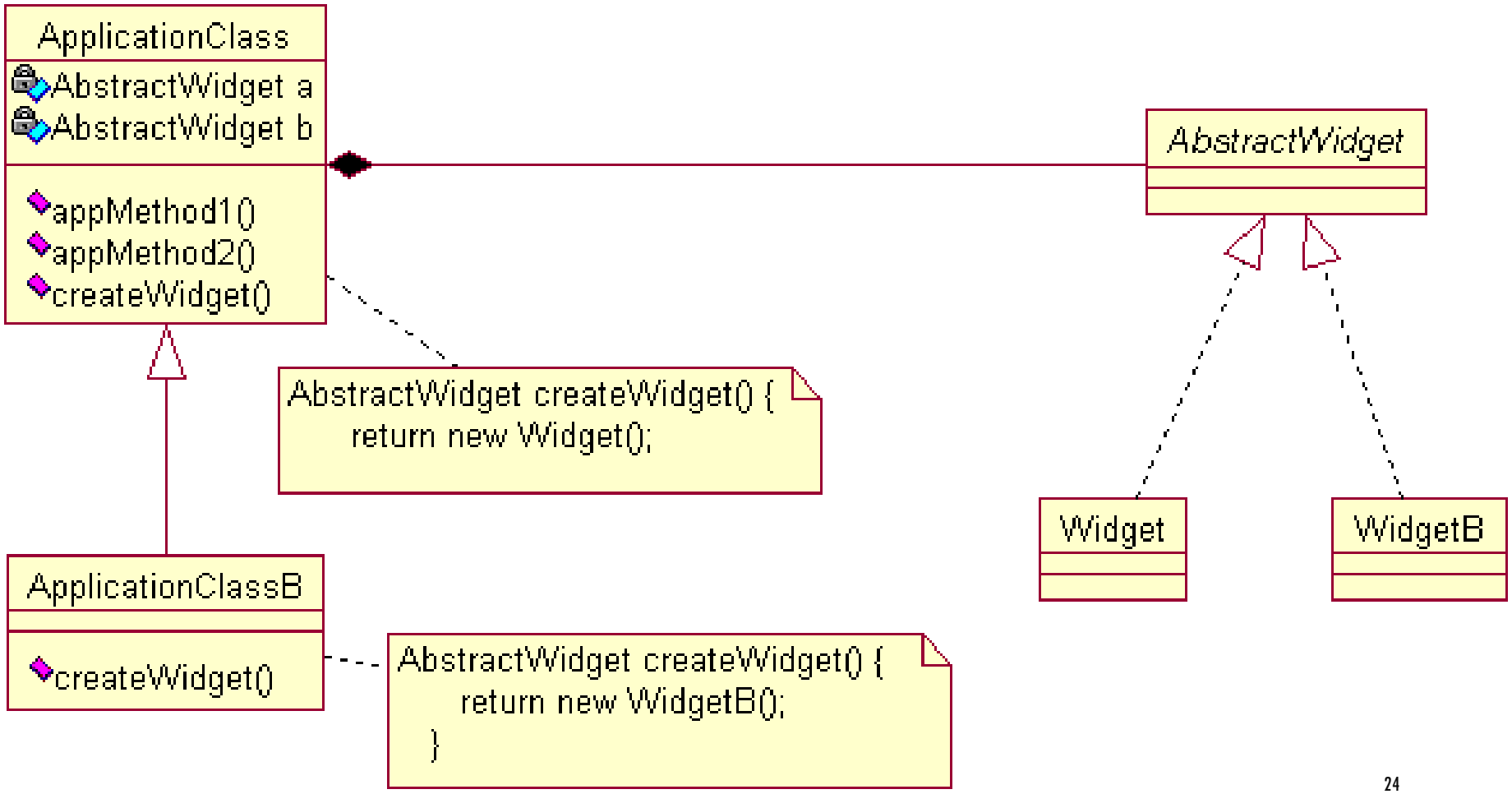
APPLY "DEPEND ON ABSTRACTIONS"

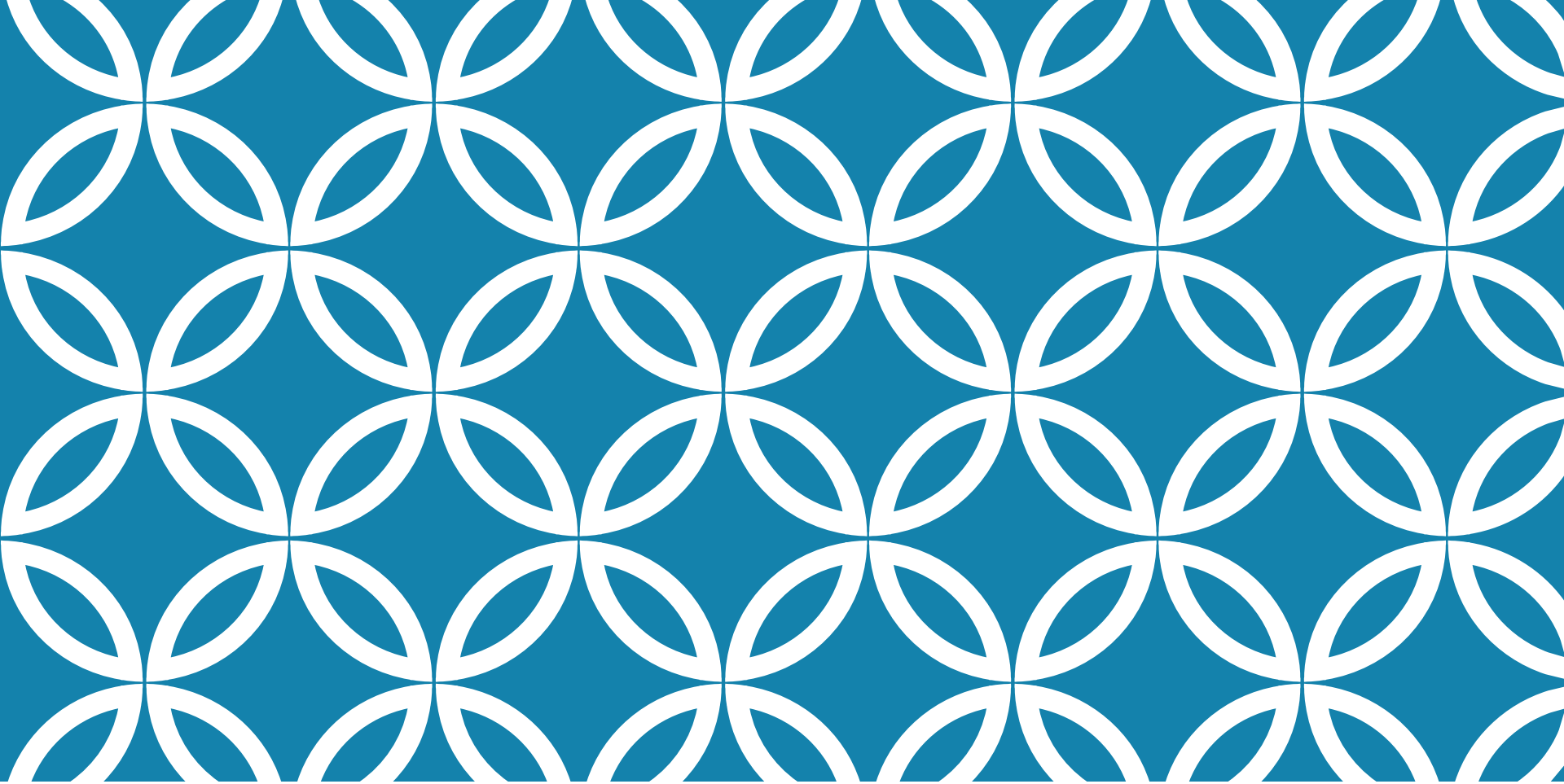


`ApplicationClass` depends now on an (abstract) interface

But we still have hard coded which widget to create

USE A FACTORY METHOD





FACTORY METHOD

BASIC ASPECTS

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses

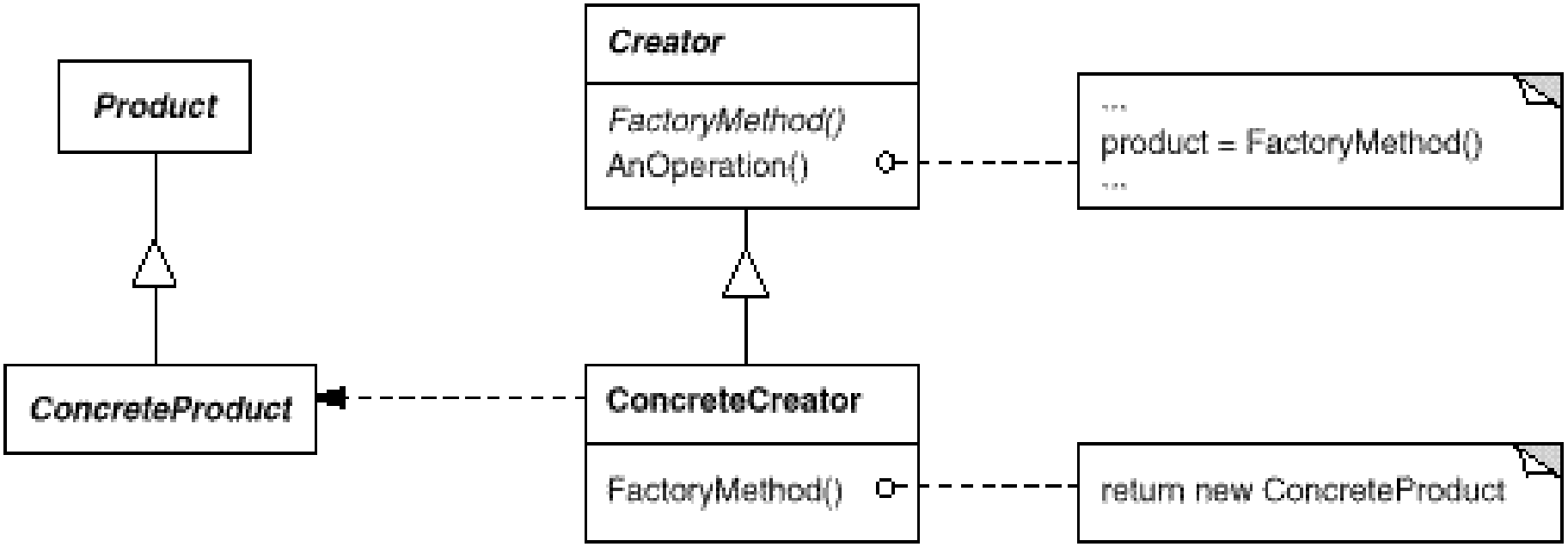
Also Known As

- Virtual Constructor

Applicability

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsibility to one of several helper subclasses

STRUCTURE


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

PARTICIPANTS & COLLABORATIONS

Product

- defines the interface of objects that will be created by the FM
- Concrete Product implements the interface

Creator

- declares the FM, which returns a product of type Product.
- may define a default implementation of the FM

ConcreteCreator

- overrides FM to provide an instance of ConcreteProduct

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct

CONSEQUENCES

Eliminates binding of application specific classes into your code.

- creational code only deals with the Product interface

Provides hooks for subclassing

- subclasses can change this way the product that is created

Clients might have to subclass the Creator just to create a particular ConcreteProduct object.

IMPLEMENTATION ISSUES

Varieties of Factory Methods

- Creator class is **abstract**
  - does not provide an implementation for the FM it declares
  - requires subclasses
- Creator is a **concrete** class
  - provides default implementation
  - FM used for flexibility
  - create objects in a separate operation so that subclasses can override it

Parameterization of Factory Methods

- A variation on the pattern lets the factory method create multiple kinds of products
- a *parameter* identifies the type of Product to create
- all created objects share the Product interface

PARAMETERIZING THE FACTORY

```
class Creator {
  public Product create(productId id)
  {
   if (id == MINE) return new MyProduct();
   if (id == YOURS) return new YourProduct();
   …
  }
};

class MyCreator extends Creator {
  public Product create(productId id) {
   if (id == MINE) return new YourProduct();
   if (id == YOURS) return new MyProduct();
   if (id == THEIRS) return new TheirProduct();
   return super.create(id); // called if others fail
}
```

selectively *extend* or *change* products that get created

EVALUATION OF FACTORY METHOD SOLUTION

Explicit creation of **`Widget`** objects is not anymore dispersed

- easier to change

Functional methods in **`ApplicationClass`** are decoupled from various concrete implementations of widgets

Avoid ugly code duplication in **`ApplicationClassB`**

- subclasses reuse the functional methods, just implementing the concrete *Factory Method* needed

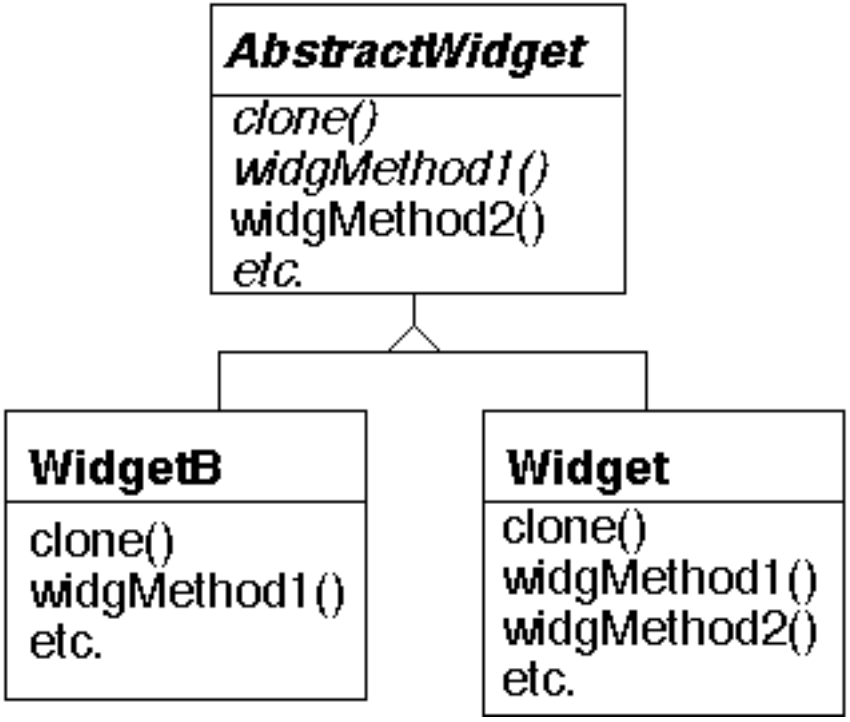
Disadvantages

- create a subclass only to override the factory-method
- can't change the **`Widget`** at run-time

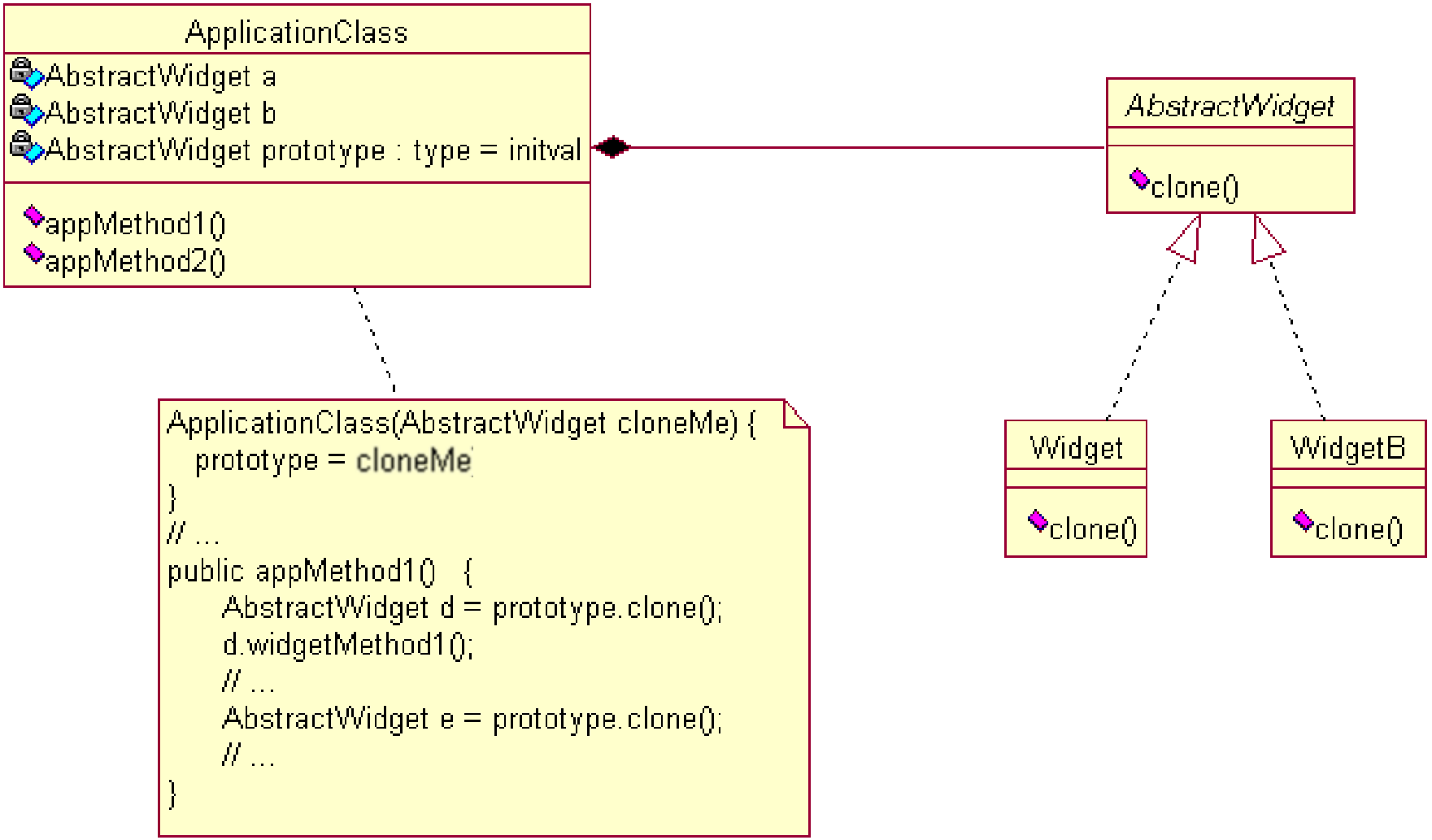
SOLUTION 2: CLONE A PROTOTYPE

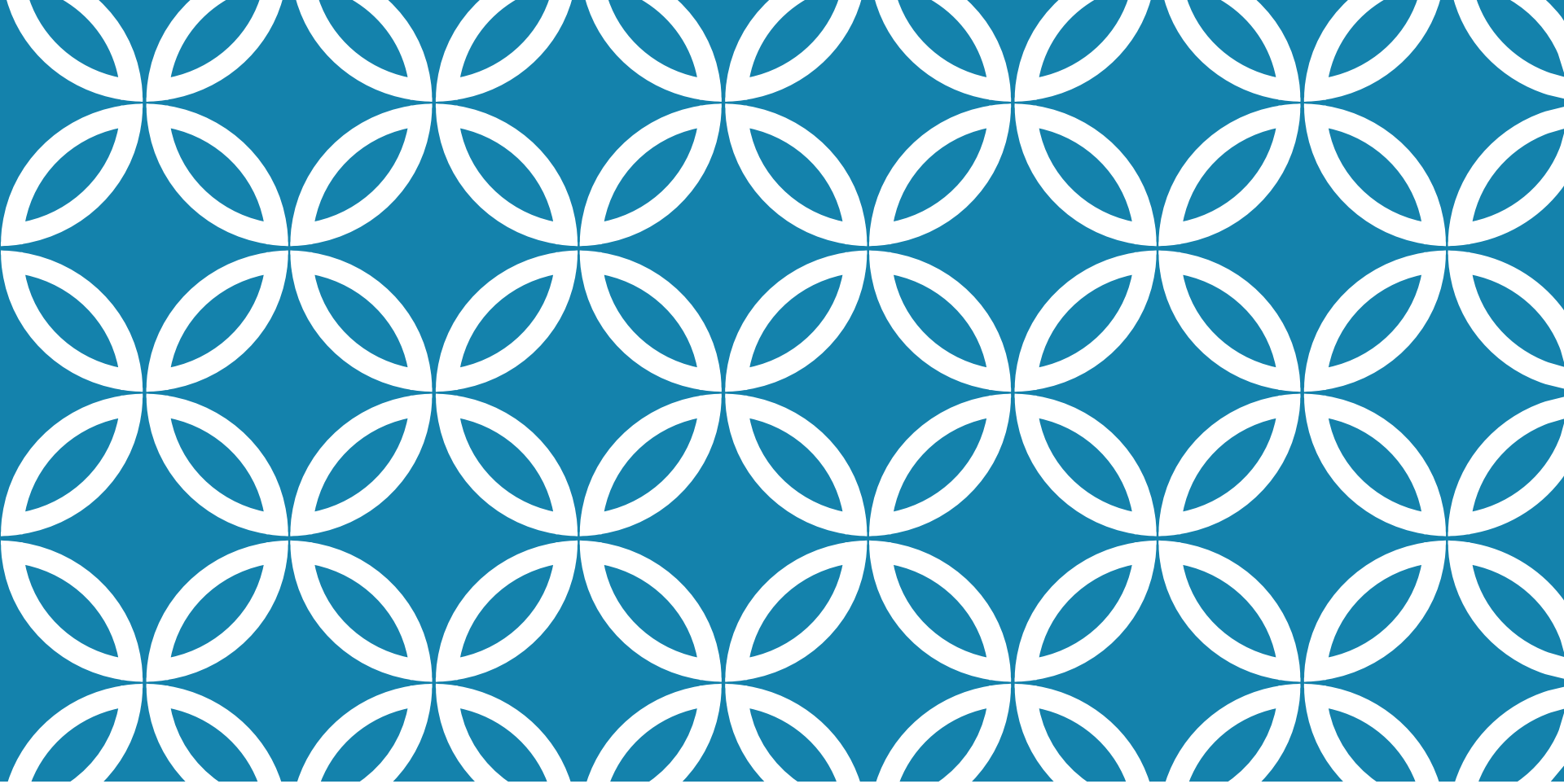
Provide the **`Widgets`** with a clone method

- make a copy of an existing Widget object



USING THE CLONE





THE PROTOTYPE PATTERN

BASIC ASPECTS

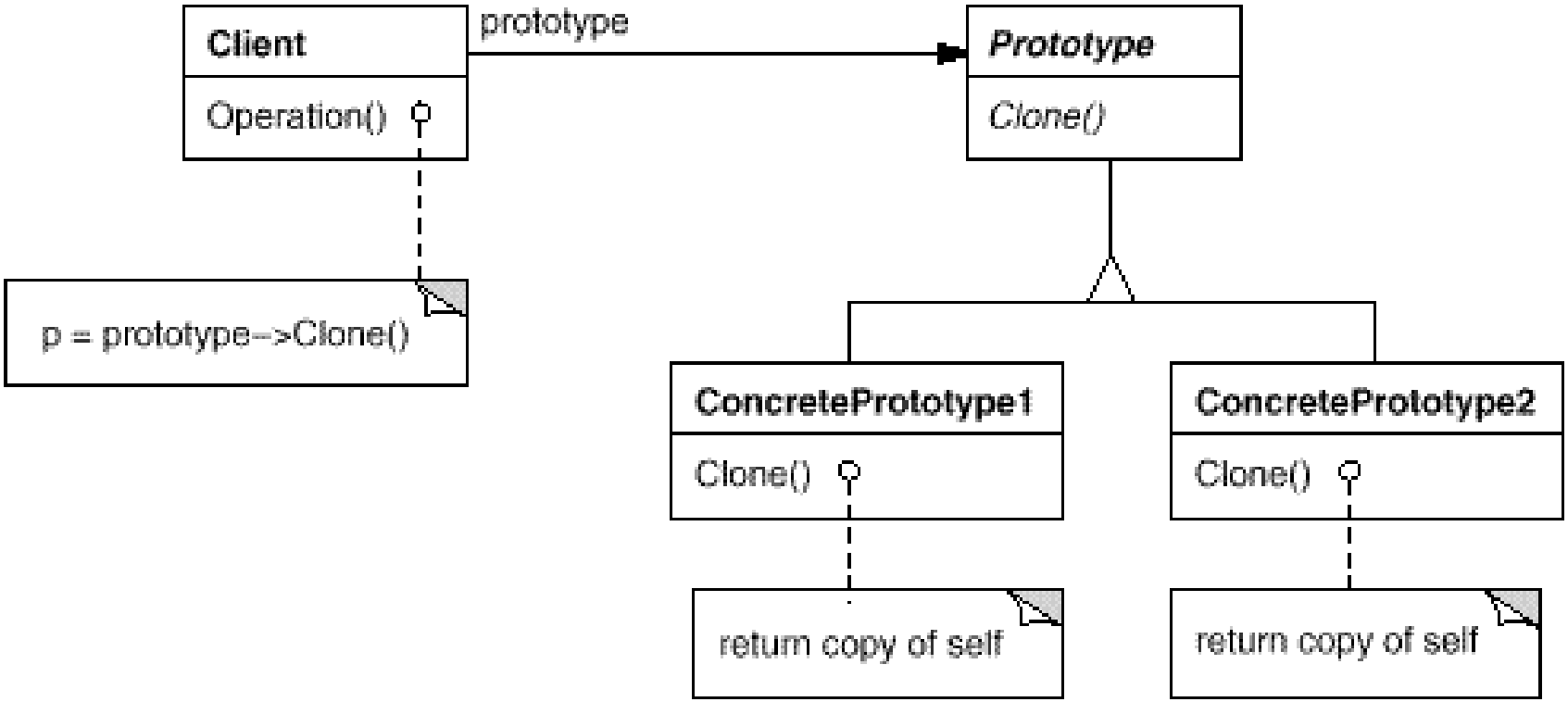
Intent

- Specify the kinds of objects to create using a prototypical instance
- Create new objects by copying this prototype

Applicability

- when a client class should be independent of how its products are created, composed, and represented **and**
- when the classes to instantiate are *specified at run-time*

STRUCTURE


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

PARTICIPANTS & COLLABORATIONS

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.

A client asks a prototype to clone itself.

The client class must initialize itself in the constructor with the proper concrete prototype.

CONSEQUENCES

Adding and removing products at run-time

Reduced subclassing

- avoid parallel hierarchy for creators

Specifying new objects by varying **values of prototypes**

- client exhibits new behavior by delegation to prototype

Each subclass of Prototype must implement **`clone`**

- difficult when classes already exist or
- internal objects don't support copying or have circular references

IMPLEMENTATION ISSUES

Using a Prototype manager

- number of prototypes isn't fixed
  - keep a registry → **prototype manager**
- clients instead of knowing the prototype know a manager

Initializing clones

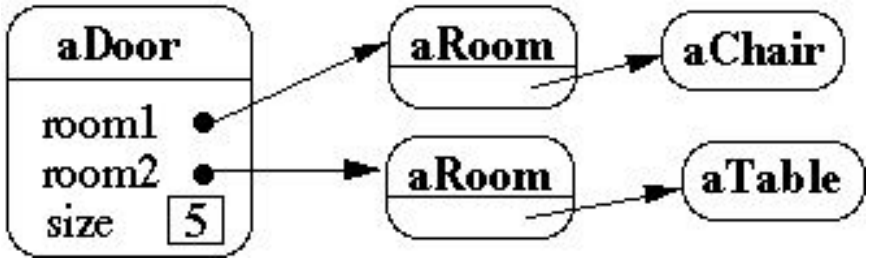
- heterogeneity of initialization methods
- write an **`Initialize`** method

Implementing the **`clone`** operation

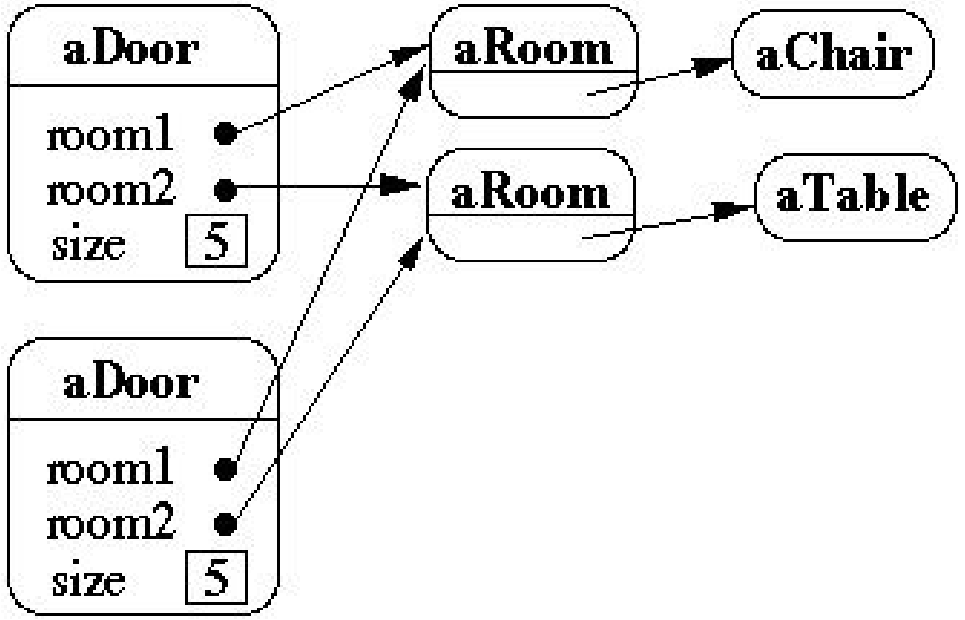
- shallow vs. deep copy

SHALLOW COPY VS. DEEP COPY

Original

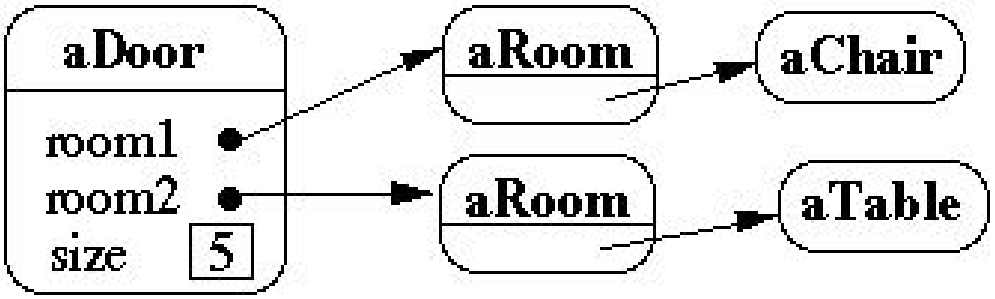


Shallow Copy

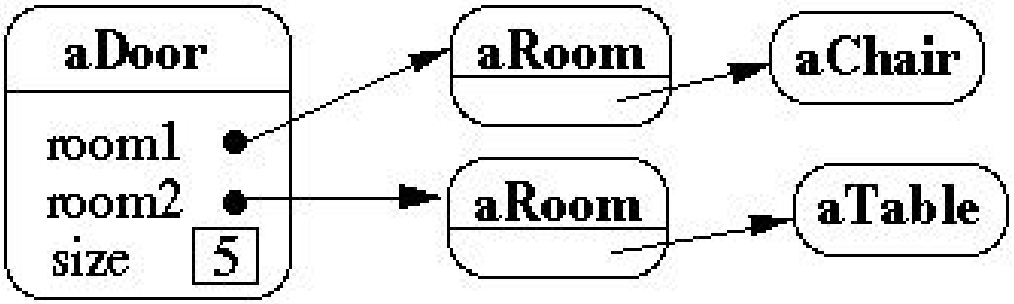


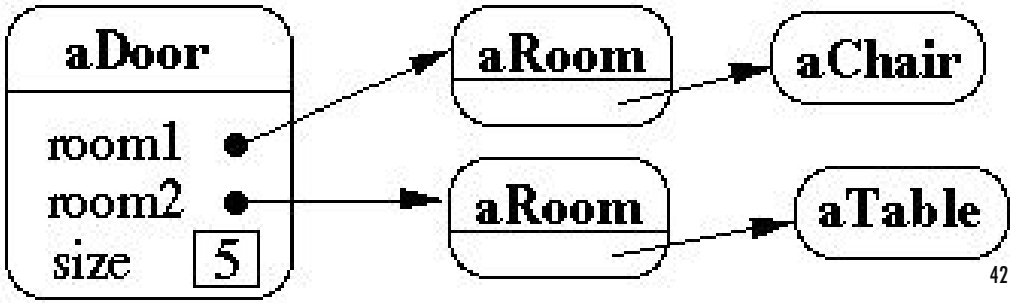
SHALLOW COPY VS. DEEP COPY (2)

Original



Deep Copy





CLONING IN C++ – COPY CONSTRUCTORS

```
class Door {
  public:
    Door();
    Door( const Door&);
    virtual Door* clone() const;
    virtual void Initialize( Room*, Room* );
 private:
    Room* room1; Room* room2;
};

//Copy constructor
Door::Door ( const Door& other )  {
  room1 = other.room1; room2 = other.room2;
}

Door* Door::clone() {
  return new Door( *this );
}
```

CLONING IN JAVA – OBJECT CLONE()

```
protected Object clone() throws
  CloneNotSupportedException
```

Creates a clone of the object

- allocate a new instance and,
- place a *bitwise clone* of the current object in the new object.

```
class Door implements Cloneable  {
  public void Initialize( Room a, Room b) {
    room1 = a; room2 = b;
  }

  public Object clone() throws
CloneNotSupportedException {
          return super.clone();
  }
  Room room1, room2;
}
```

ADVANTAGES

Classes to instantiate may be specified dynamically

- client can install and remove prototypes at run-time

We avoided subclassing of **`ApplicationClass`**

- Remember: *Favor Composition over Inheritance*!

Totally hides concrete product classes from clients

- Reduces implementation dependencies

MORE CHANGES

What if **`ApplicationClass`** uses other "products" too...

- e.g. Wheels, etc.

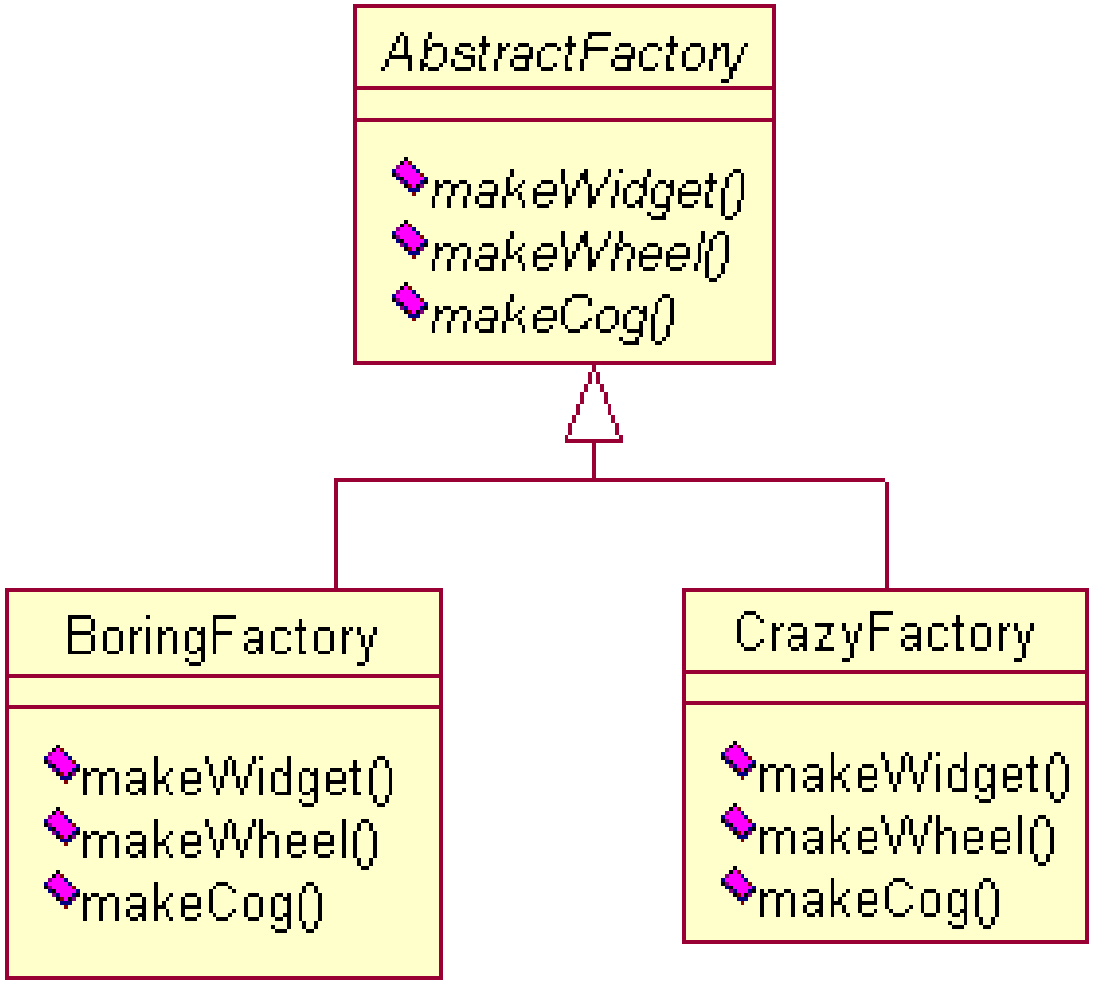
Each one of these stays for an object family

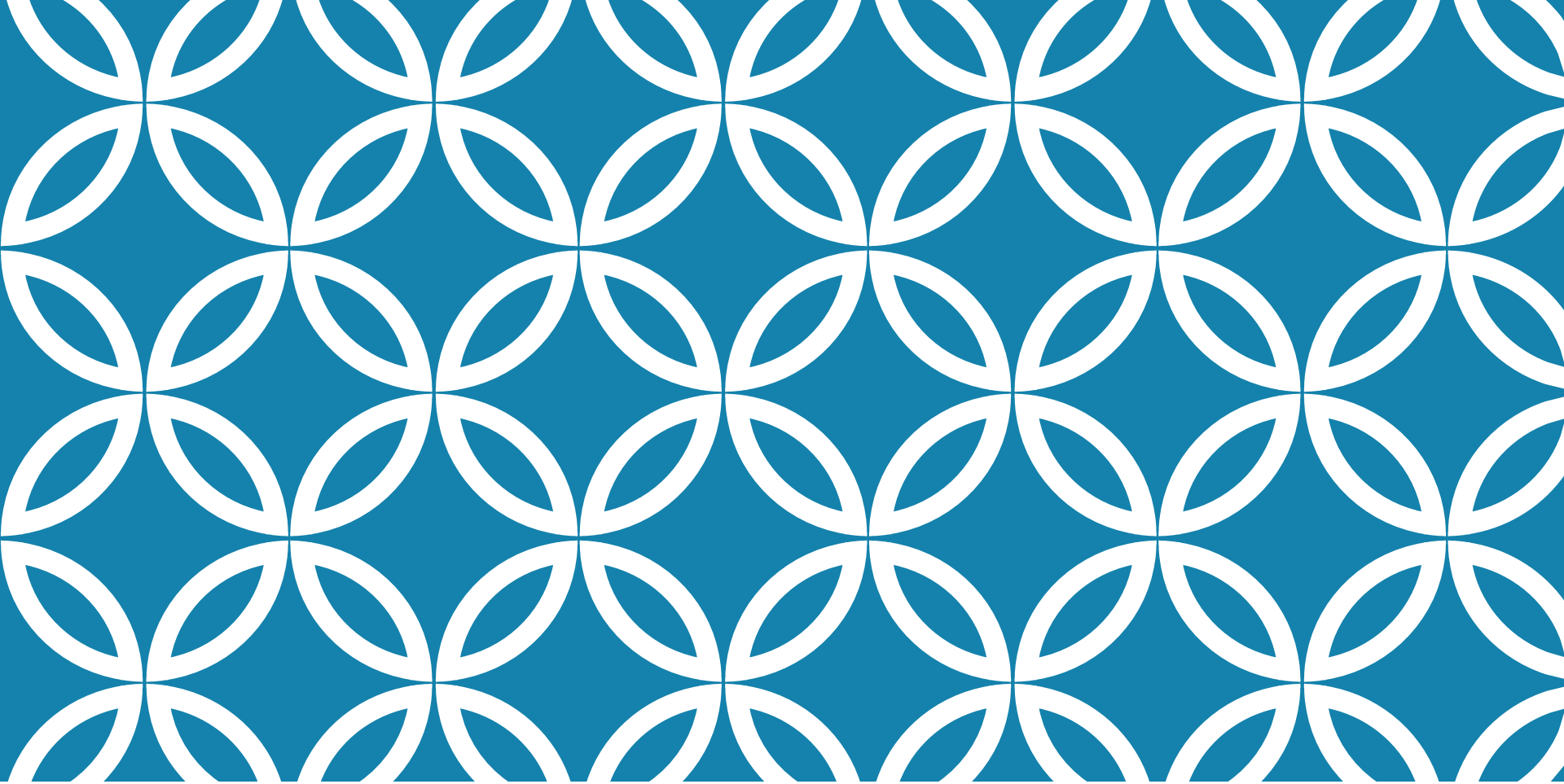
- i.e. all of these have subclasses

Assume that there are restrictions on what type of Widget can be used with which type of Wheel or Cog

Factory Methods or Prototypes can handle each type of product but it gets hard to insure the wrong types of items are not used together

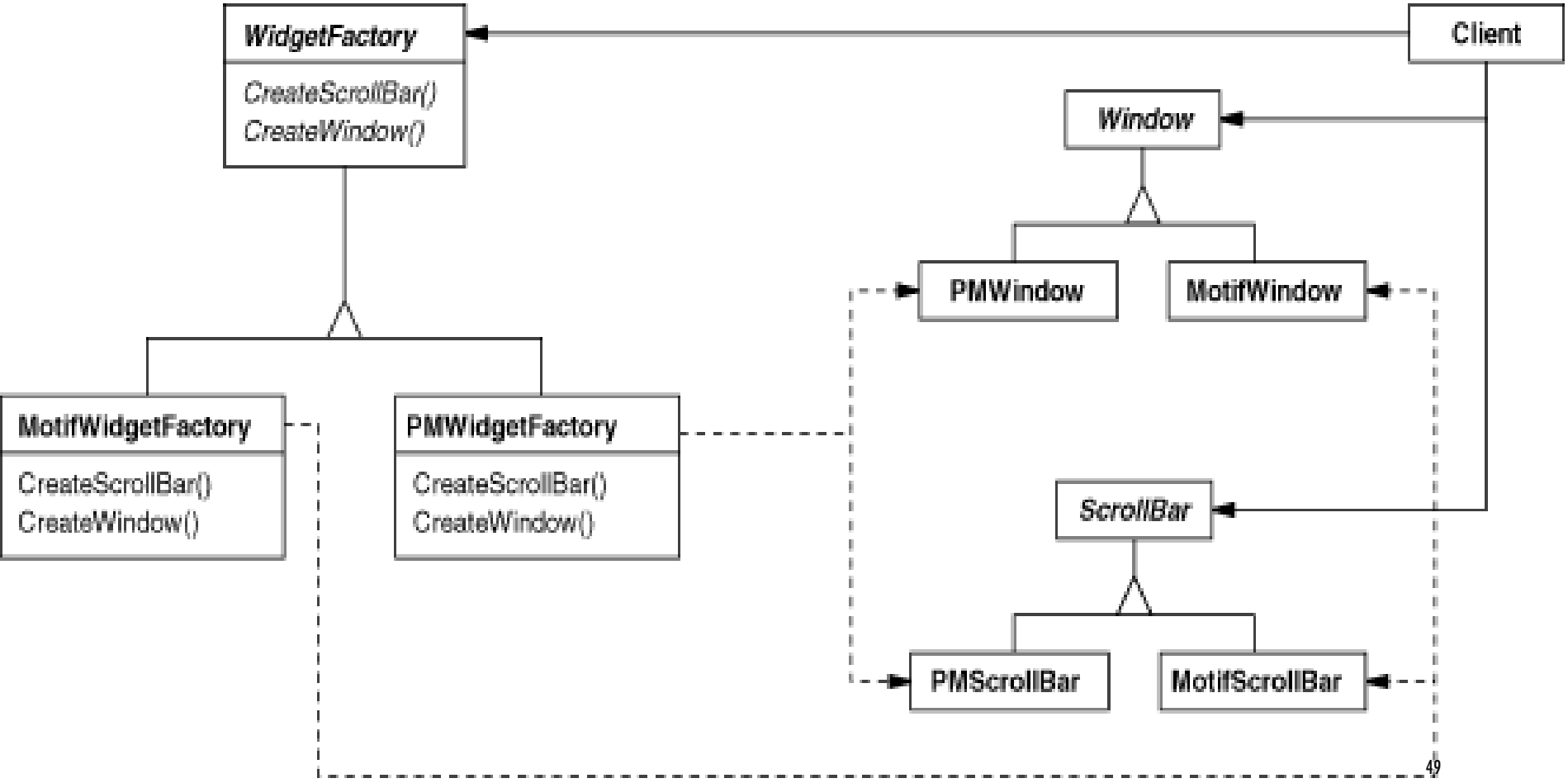
SOLUTION: CREATE AN ABSTRACT FACTORY





ABSTRACT FACTORY

INTRODUCTIVE EXAMPLE



BASIC ASPECTS

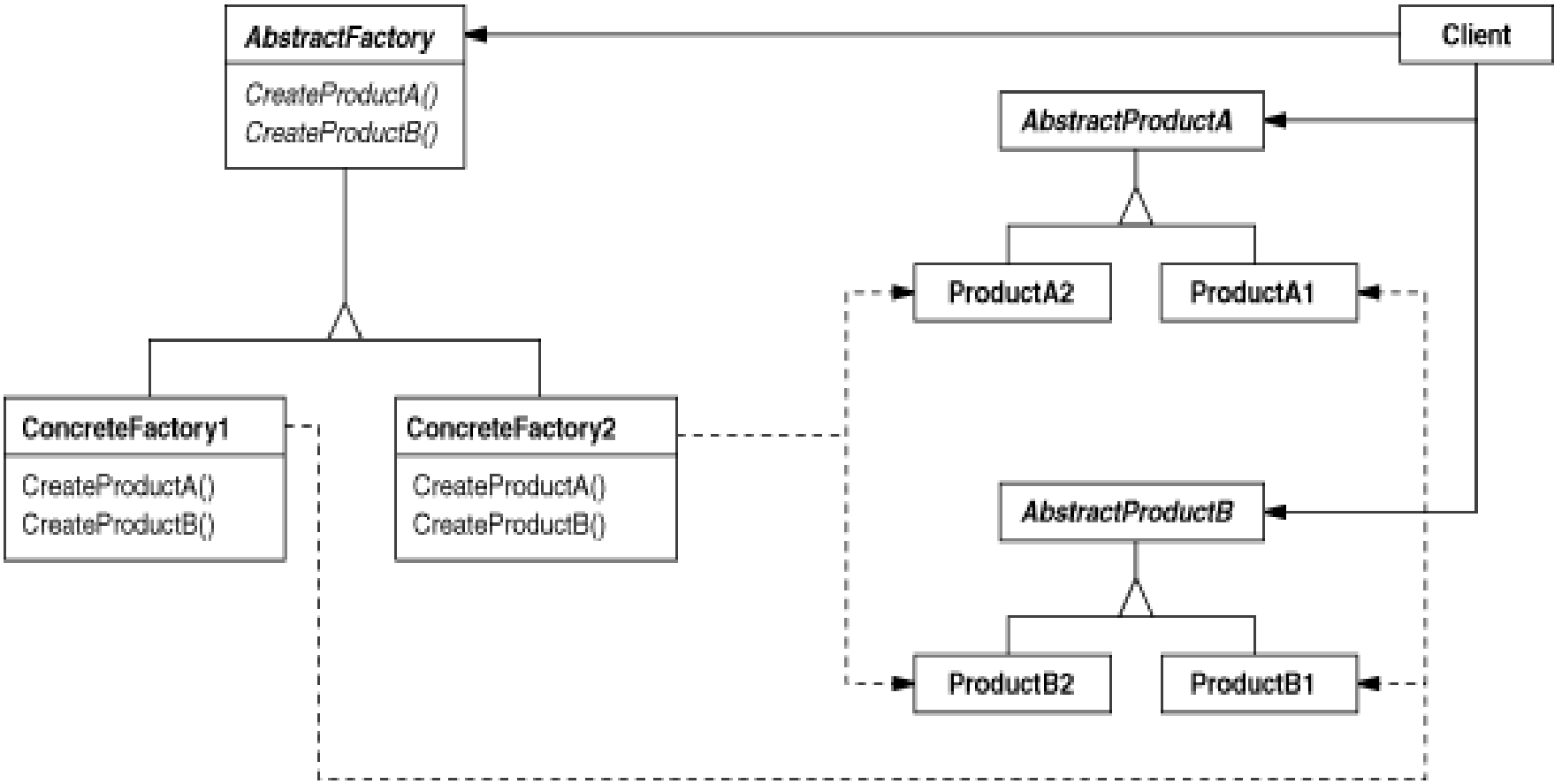
Intent

- Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes

Applicability

- System should be independent of how its products are created, composed and represented
- System should be configured with one of multiple families of products
- Need to **enforce** that a family of product objects is used together

STRUCTURE


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

PARTICIPANTS & COLLABORATIONS

Abstract Factory

- declares an interface for operations to create abstract products

ConcreteFactory

- implements the operations to create products

AbstractProduct

- declares an interface for a type of product objects

ConcreteProduct

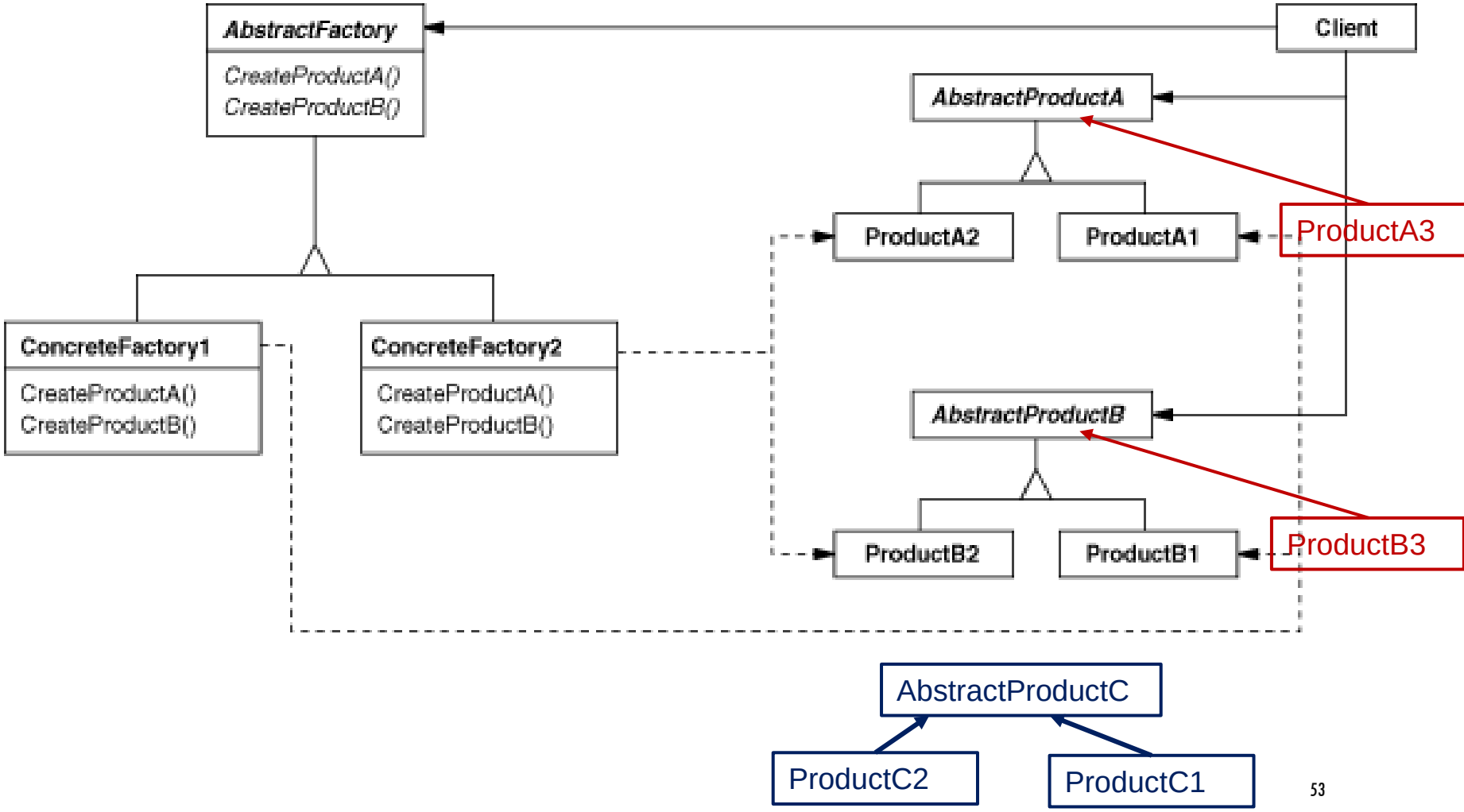
- declares an interface for a type of product objects

Client

- uses only interfaces decl. by AbstractFactory and AbstractProduct

A single instance of a ConcreteFactory created.

WHAT CHANGES ARE SUPPORTED?



CONSEQUENCES

Isolation of concrete classes

- appear in **`ConcreteFactories`** not in client's code

Exchanging of product families becomes easy

- a **`ConcreteFactory`** appears only in one place

Promotes consistency among products

- all products in a family change **at once**, and change **together**

Supporting new kinds of products is difficult

- requires a change in the interface of AbstractFactory
- ... and consequently all subclasses

IMPLEMENTATION ISSUES

Factories as **Singletons**

- to assure that only one ConcreteFactory per product family is created

Creating the Products

- collection of *Factory Methods*
- can be also implemented using *Prototype*
  - define a prototypical instance for each product in ConcreteFactory

Defining Extensible Factories

- a single factory method with parameters
- more flexible, less safe!

CREATING PRODUCTS

...using own factory methods

```
abstract class WidgetFactory {
   public Window createWindow();
   public Menu createMenu();
   public Button createButton();
}

class MacWidgetFactory extends WidgetFactory {
      public Window createWindow()
         { return new MacWidow() }
      public Menu createMenu()
         { return new MacMenu() }
      public Button createButton()
         { return new MacButton()  }
}
```

CREATING PRODUCTS

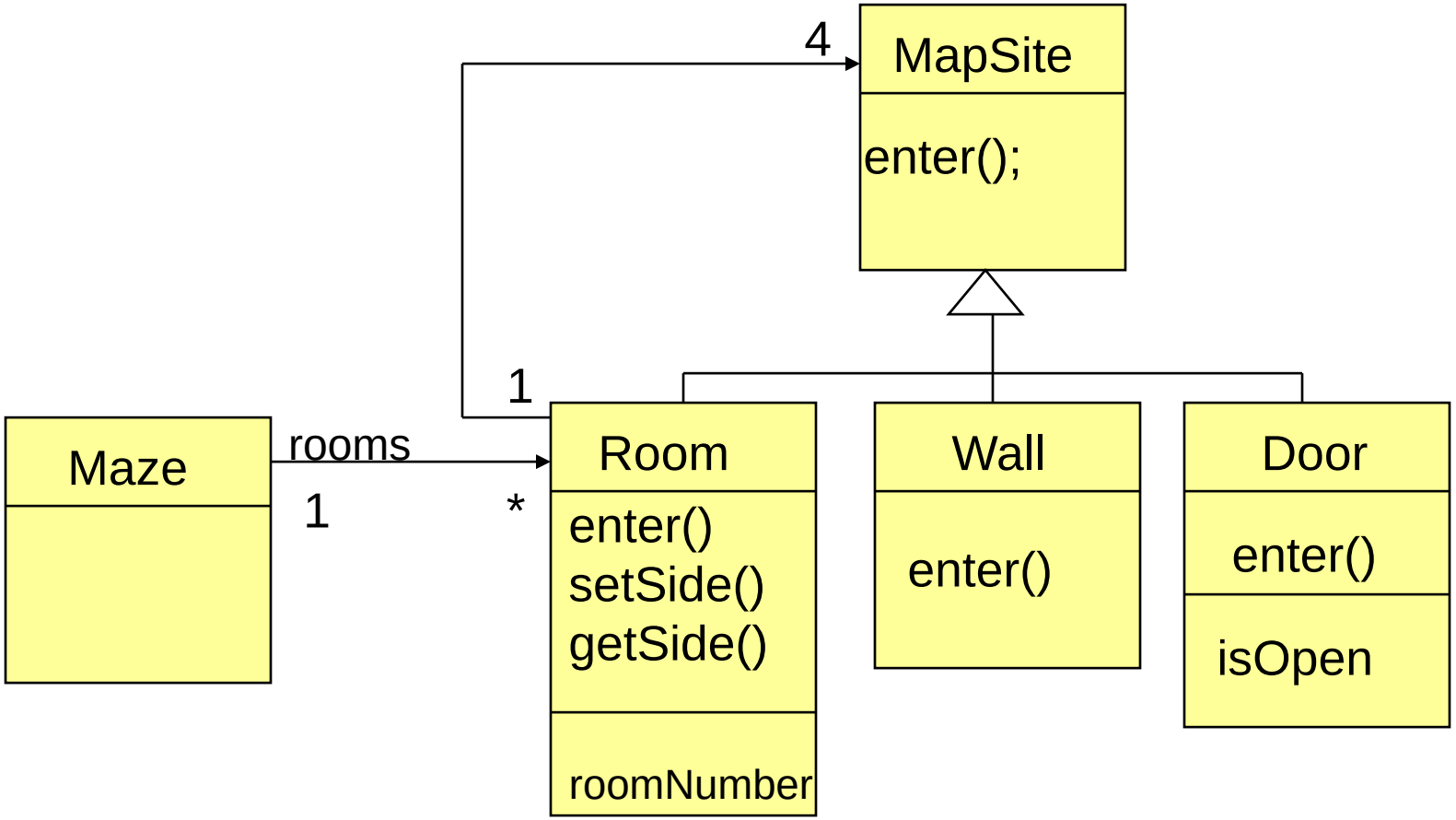
... using product's factory methods

- subclass just provides the concrete products in the constructor
- spares the reimplementation of FM's in subclasses

```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }
   public Menu createMenu();
      { return menuFactory.createWindow() }
   public Button createButton()
      { return buttonFactory.createWindow() }
}

class MacWidgetFactory extends WidgetFactory {
   public MacWidgetFactory() {
      windowFactory = new MacWindow();
      menuFactory = new MacMenu();
      buttonFactory = new MacButton();
   }
}
```

CREATIONAL DP IN ACTION

Maze Game



COMMON ABSTRACT CLASS FOR ALL MAZE COMPONENTS

```
public enum Direction {North, South, East, West};

class MapSite {
    public abstract void enter();
};
```

Meaning of enter() depends on *what* you are entering.

- room → location changes
- door → if door is open, go in

COMPONENTS OF THE MAZE – MAZE

```
class Maze {
    public void addRoom(Room r){...};
    Room getRoom(int) {…};
};
```

A maze is a collection of rooms. Maze can find a particular room given the room number.

roomNo() could do a lookup using a linear search or a hash table or a simple array.

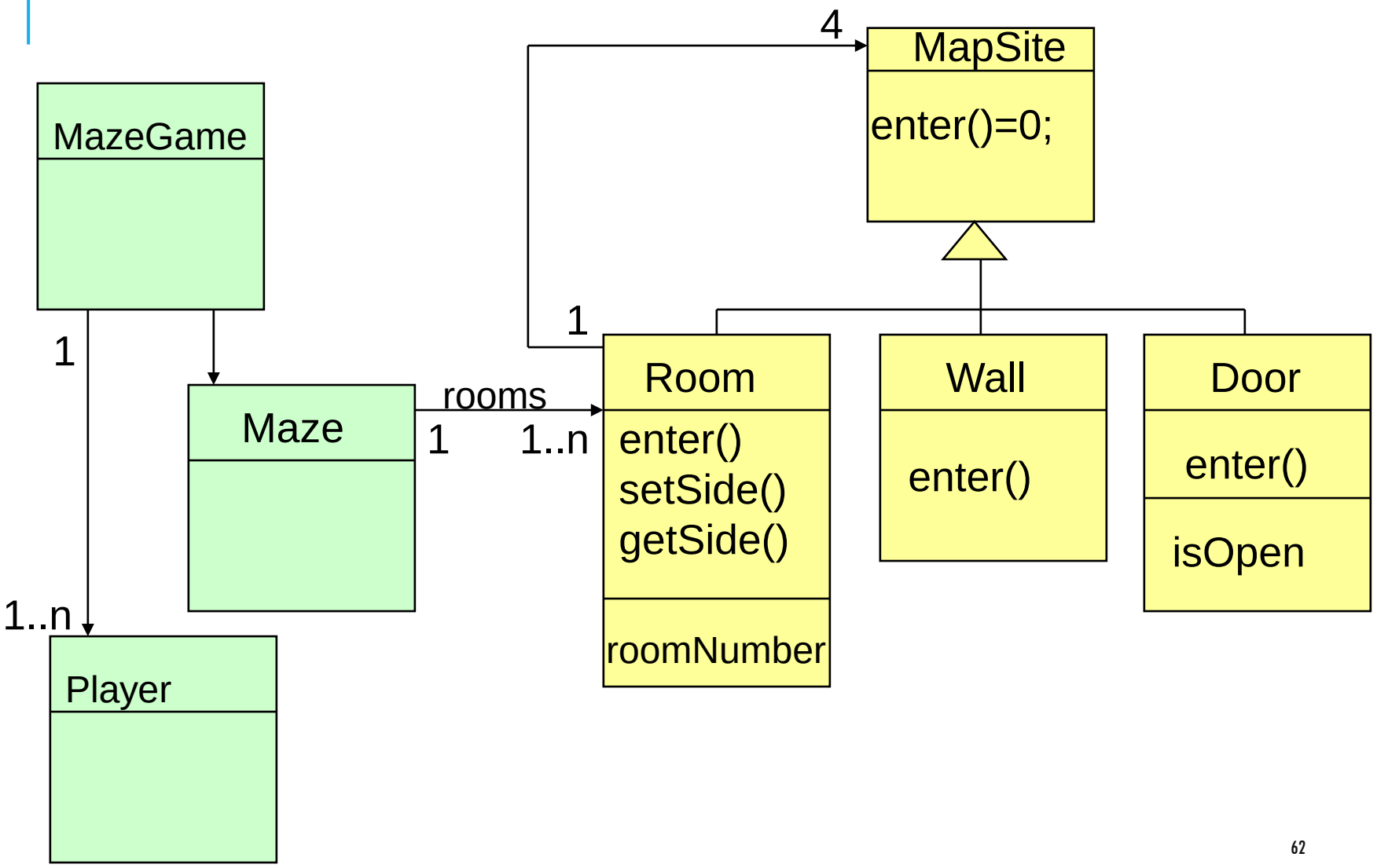
COMPONENTS OF THE MAZE — WALL & DOOR & ROOM

```
public class Room extends MapSite {
   int roomNumber;
   MapSite sides[4];

   public Room(int roomNo) {…};
   public MapSite getSide(Direction
d) {…};
   public void setSide(Direction d,
MapSite m) {…};

   public void enter() {…};
}
```

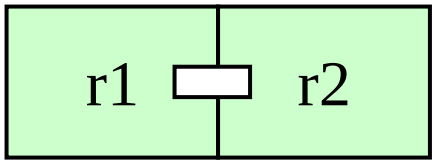
```
public class Wall extends MapSite
{
   public Wall();
   public void enter();
};
```

```
public class Door extends MapSite
{
   Room room1, room2;
   bool isOpen;

   public Door(Room r1, r2){…};
   public void enter() {…};
};
```

WE WANT TO PLAY A GAME!



CREATING THE MAZE




```
public class MazeGame {
   public Maze 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());
...}
```

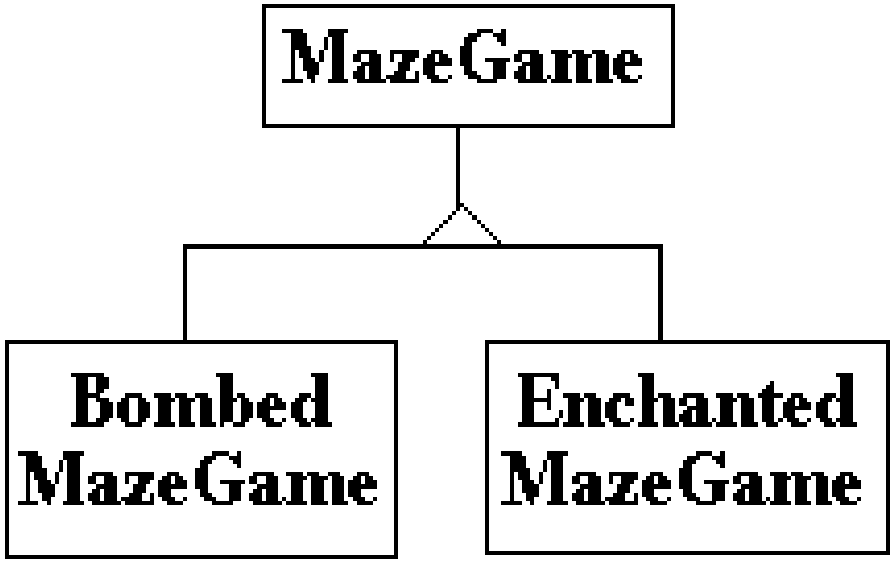
The problem is **inflexibility** due to hard-coding of maze layout

Pattern can make game creation more flexible... *not* smaller!

WE WANT FLEXIBILITY IN MAZE CREATION

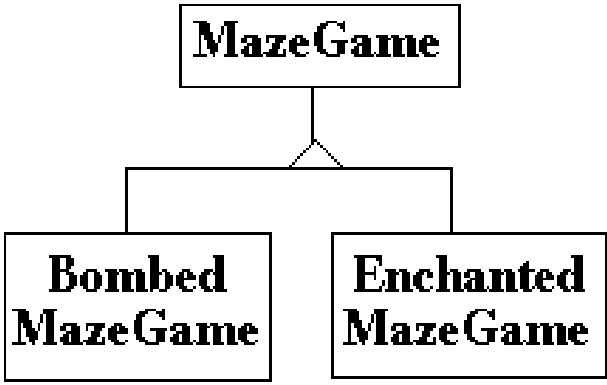
Be able to vary the kinds of mazes

- Rooms with bombs
- Walls that have been bombed
- Enchanted rooms
  - Need a spell to enter the door!



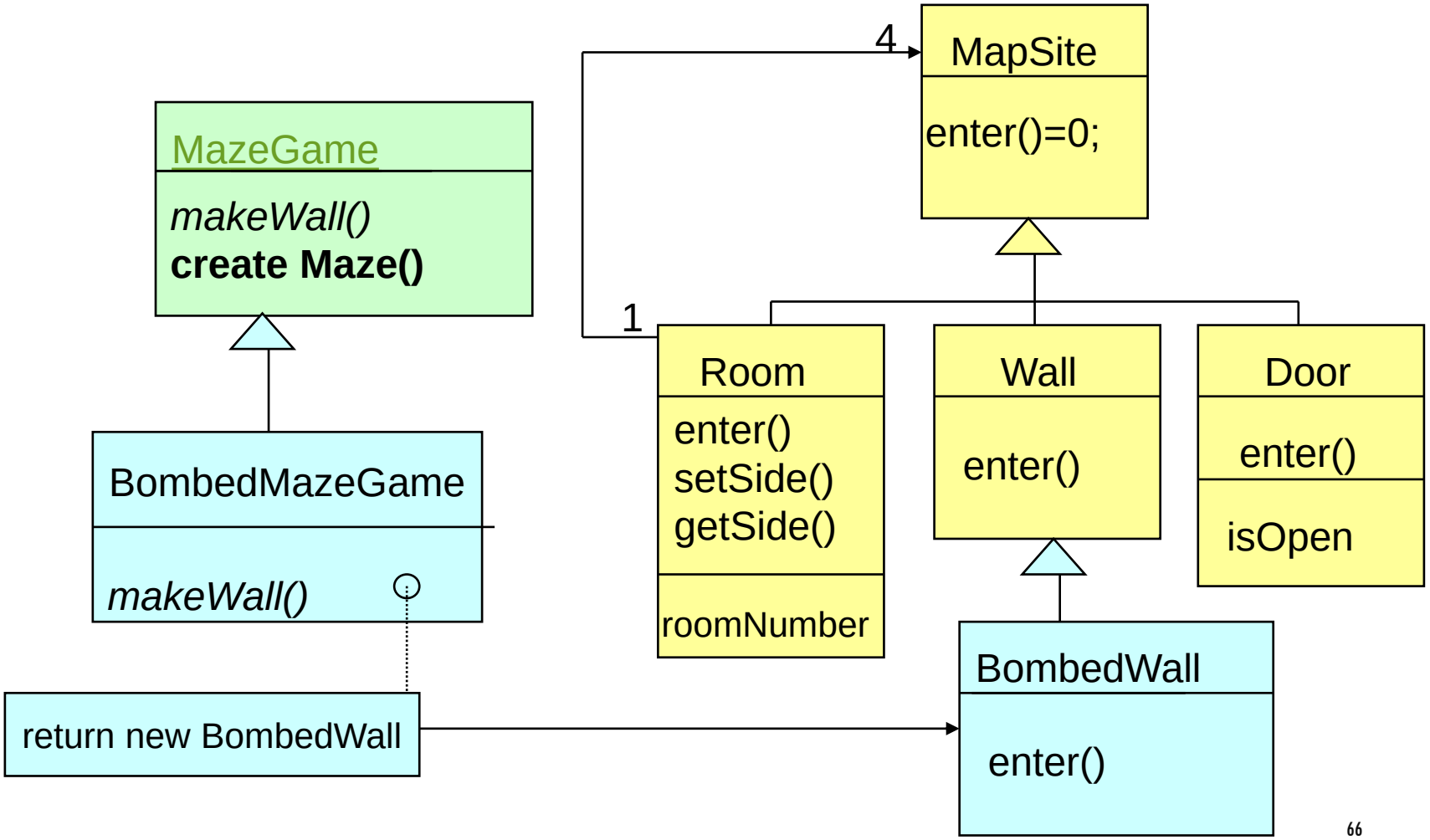
IDEA 1: SUBCLASS MAZEGAME, OVERRIDE CREATEMAZE

```
public class BombedMazeGame {
    public Maze createMaze() {
        Maze aMaze = new Maze;
        Room r1 = new RoomWithABomb(1);
        Room r2 = new RoomWithABomb(2);
        Door theDoor = new Door(r1, r2);
        aMaze.addRoom(r1);
        aMaze.addRoom(r2);

        r1.setSide(North, new BombedWall());
        r1.setSide(East, theDoor);
        r1.setSide(South, new BombedWall());
        r1.setSide(West, new BombedWall());
…}
```



Lots of code duplication… :((

IDEA 2: USE A FACTORY METHOD



APPLYING FACTORY METHOD

```
public class MaseGame {
    public Maze createMaze () {
        Maze aMaze = makeMaze();

        Room r1 = makeRoom(1);
        Room r2 = makeRoom(2);
        Door theDoor = makeDoor(r1, r2);

        aMaze.addRoom(r1);
        aMaze.addRoom(r2);

        r1.SetSide(North, makeWall());
        r1.SetSide(East, theDoor);
        r1.SetSide(South, makeWall());
        r1.SetSide(West, makeWall());

        r2.SetSide(North, makeWall());
        r2.SetSide(East, makeWall());
        r2.SetSide(South, makeWall());
        r2.SetSide(West, theDoor);

        return aMaze;
}}
```

IDEA 3: FACTORY METHOD IN PRODUCT

Make the product responsible for creating itself

- e.g. let the Door know how to construct an instance of it rather than the MazeGame

The client of the product needs a reference to the "creator"

- specified in the constructor

```
class Room extends MapSite  {
  public Room makeRoom(int no)  {
      return new Room(no);
  }
   // ...};

class RoomWithBomb extends Room {
  public Room makeRoom(int no)  {
  return new RoomWithBomb();
  }
   // ... };

class MazeGame {
    private Room roomMaker;
    // ...
    public MazeGame(Room rfactory) {
       roomMaker = rfactory;
    }
    public Maze createMaze() {
       Maze aMaze = new
Maze();
       Room r1 =
roomMaker.makeRoom(1);
     // ...};
```

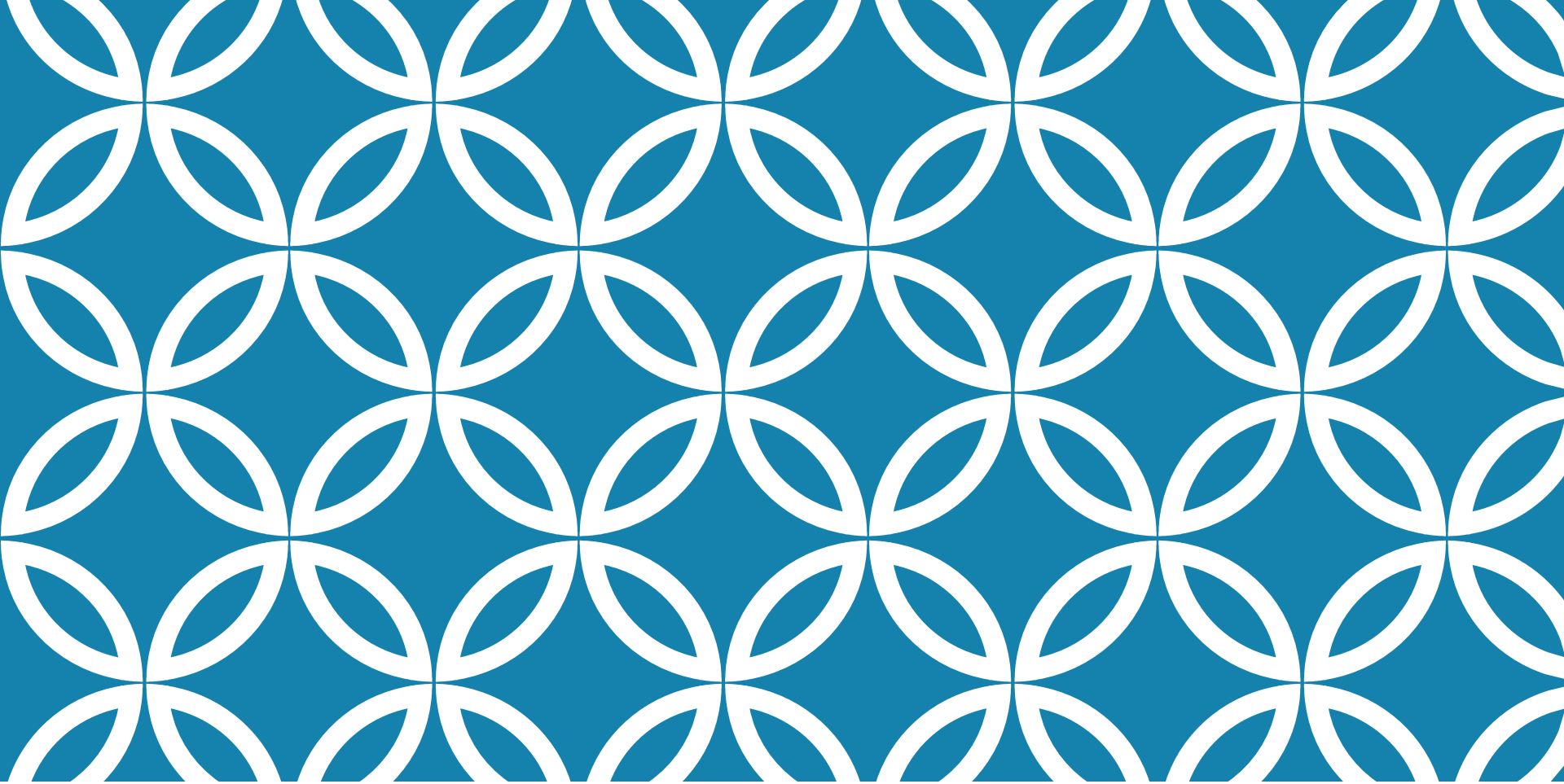
SOLVING THE MAZE PROBLEM

```
class MazePrototypeFactory {
     private Maze _prototypeMaze;
     private Room _prototypeRoom;
     private Wall _prototypeWall;
     private Door _prototypeDoor;

public MazePrototypeFactory(Maze m,
Wall w, Room r, Door d) {
    _prototypeMaze = m; _prototypeWall
= w;
    _prototypeRoom = r; _prototypeDoor
= d;
}

  public Wall makeWall()
  {
    return _prototypeWall.clone();
  }


  public Maze makeMaze() {..}
  public Room makeRoom(int) {…}
```

```
public Door makeDoor(Room r1, r2) {
     Door door =
_prototypeDoor.clone();
     door.initialize(r1, r2);
     return door;
}
```

Creating a maze for a game……..

```
MazePrototypeFactory simpleMazeFactory
= new MazePrototypeFactory (new Maze(),
new Wall(), new Room(), new Door());

MazeGame game;

Maze maze =
game.createMaze(simpleMazeFactory);
```



SINGLETON

BASICS

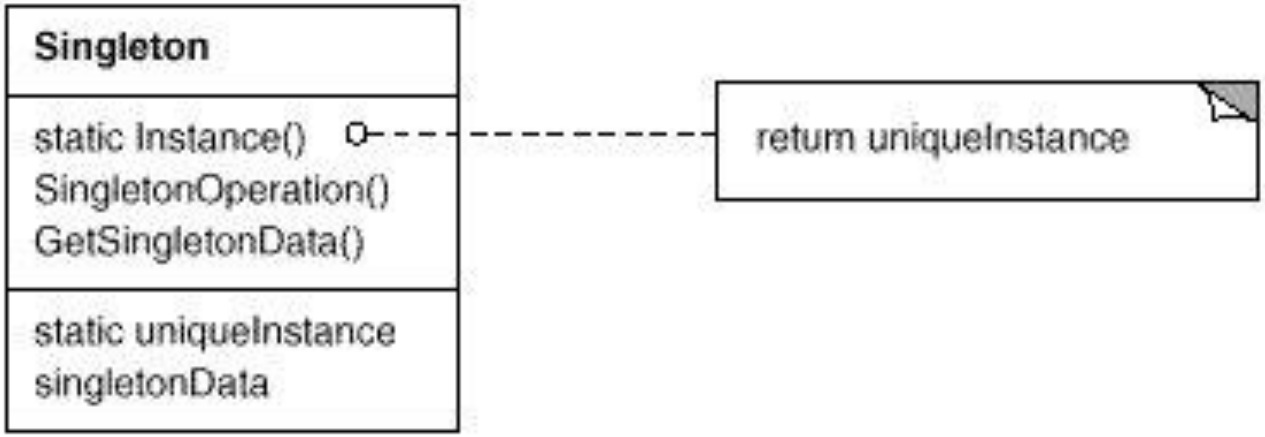
Intent

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

Applicability

- want exactly one instance of a class
- accessible to clients from one point
- want the instance to be extensible
- can also allow a countable number of instances
- improvement over global namespace

STRUCTURE OF THE PATTERN



Put constructor in private/protected data section

PARTICIPANTS AND COLLABORATIONS

Singleton

- defines an **`Instance`** method that becomes the single "gate" by which clients can access its unique instance.
  - **`Instance`** is a class method (i.e. static)
- may be responsible for creating its own unique instance

Clients access Singleton instances solely through the ***`Instance`*** *method*

CONSEQUENCES

- Controlled access to sole instance
- Permits refinement of operations and representation
- Permits a variable (but precise) number of instances
- Reduced global name space

- Singleton as an Anti-pattern – when?