Design Patterns

Part 1

Tamlin Love

Today’s Topics

- Design Patterns Introduction
- Types of Design Pattern
- Anti-Patterns
- Creational Design Patterns
  - Factory
  - Prototype
  - Abstract Factory
  - Builder
  - Singleton
- Design Pattern Pitfalls

I Am Devloper

@iamdevloper

manager: we need to design an admin system for a veterinary centre

dev: ok, this is it, remember your training

class Dog extends Animal {}

Design Patterns

- A **design pattern** is a general, reusable solution to a commonly occurring software design problem
  - A design pattern is not a ready-made code package
  - Rather, it is a template that a developer can modify and implement for their particular applications

Design Patterns

- Design patterns are different from architecture styles:
  - Architecture styles focus on components and how they relate to each other
  - Design patterns focus on low-level code, objects and classes
- If someone says “X” is a design pattern, expect code, class diagrams, etc,
- If someone says “Y” is an architecture style, expect higher level diagrams (component diagrams, activity diagrams, etc.)
- The line between architecture and design is blurry

What do Software Developers do?

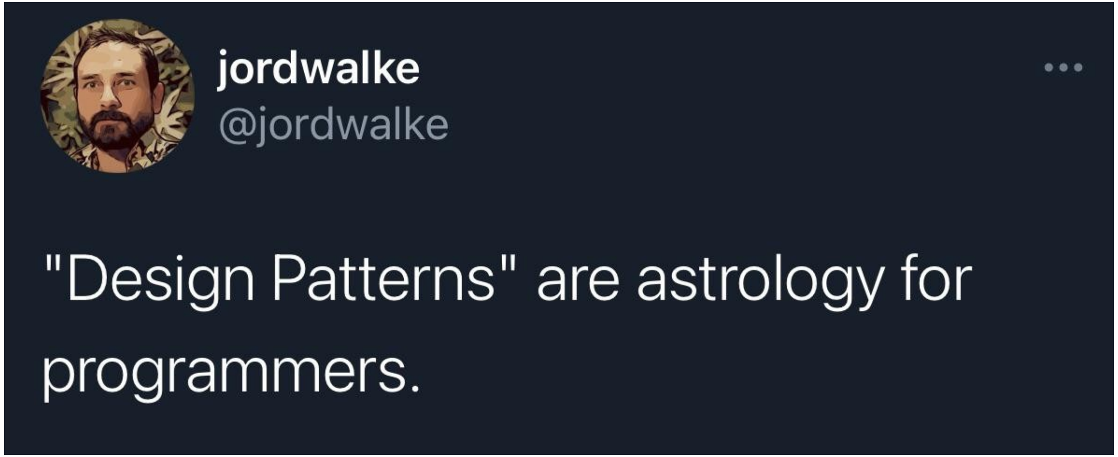
- Develop software of course!
  - Code
  - Design (requirements engineering, architecture, etc.)
- But a developer’s work is never done
  - Understand existing software
  - Maintain software (fix bugs)
  - Upgrade (add new features)
- In other words, software is constantly changing

Why Design Patterns?

- Design patterns help anticipate change
  - Change is needed to keep up with reality
    - Change may be triggered by changing business environment or by deciding to refactor classes that are deemed potentially problematic
  - Change may have bad consequences when there are unrelated reasons to change a software module/class
    - Unrelated reasons are usually because of unrelated responsibilities
    - Change in code implementing one responsibility can unintentionally lead to faults in code for another responsibility
- Design patterns can also target complex conditional logic
- Design patterns provide a common language to communicate solutions with other developers

Other Considerations

- When is a design pattern needed/applicable?
- How can we tell if a pattern is making things better or worse?
- When should we avoid patterns that will make things worse?



jordwalke

@jordwalke

"Design Patterns" are astrology for programmers.

Types of Design Patterns

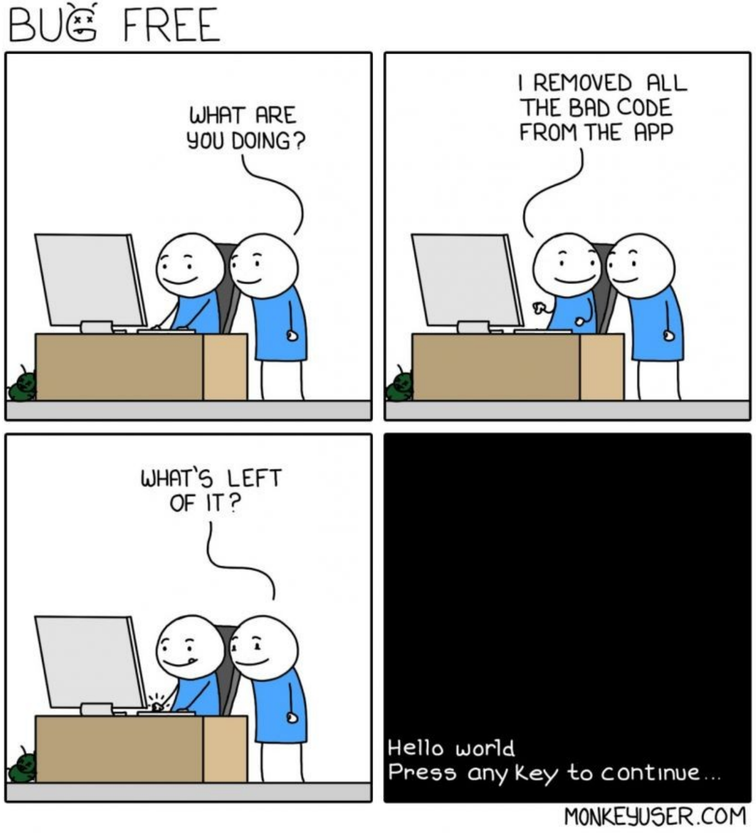
- **Creational design patterns** deal with object creation mechanisms
  - Examples: Abstract Factory, Builder, Factory, Prototype, Singleton
- **Structural design patterns** deal with organising objects to form larger structures and provide new functionality
  - Examples: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- **Behavioral design patterns** deal with recognising and realising common communication patterns between objects
  - Examples: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor
- **Concurrency design patterns** deal with multi-threaded programs
  - Examples: Active Object, Double-Checked Locking, Monitor Object, Reactor, Thread-Specific Storage
  - Not covered in this course

References

- Design Patterns: Elements of Reusable Object-Oriented Software
  - Written in 1994, it is a very influential text in software design
  - Written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
    - The so-called “Gang of Four” (GoF)
  - Online version
- Refactoring Guru – Design Patterns

Anti-Patterns

- An **anti-pattern** is a design pattern that is ineffective and counterproductive
  - Anti-patterns represent common pitfalls in software design



Anti-Patterns

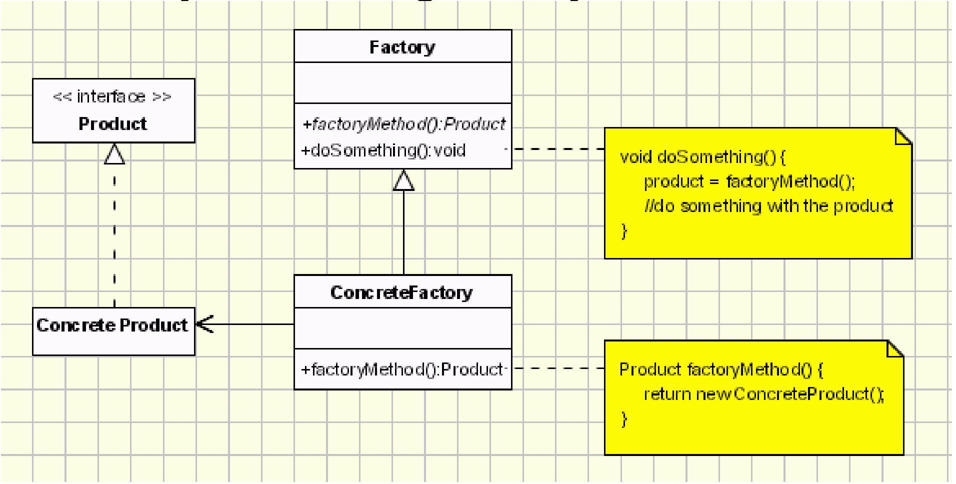
- **Spaghetti Code -** ad hoc software structure makes it difficult to extend and optimize code
  - a.k.a. **Big Ball of Mud**
- **Lava Flow** – unready code is put into production and is added to while still in an unfinished state
- **Golden Hammer** – obsessively applying a familiar tool to every software problem
  - "If all you have is a hammer, everything looks like a nail"
- **Boat Anchor** – code that doesn't do anything is left in the codebase "just in case"
- **God Class** – one class taking on too many responsibilities
- **Poltergeist Class -** useless classes with no real responsibility of their own, often used to just invoke methods in another class or add an unneeded layer of abstraction.

Creational Patterns

- **Creational design patterns** deal with object creation mechanisms
- With typical object creation (e.g. new Object):
  - Client must know which class is being created (no polymorphism)
  - Client must have complete control over the object creation (tight coupling)
- With creational design patterns:
  - Object creation can be abstracted
  - Client may not need to know which class to create (polymorphism)
  - Client may not need to know how to create an object (single responsibility)
  - New classes can be added without changing client (open-closed)
  - Shifts emphasis away from pure inheritance to composition and interfaces (can facilitate dependency inversion)

Factory

- The **factory pattern**, also known as the **virtual constructor pattern**, uses methods (called **factory methods**) to create create objects without having to specify which class is being instantiated
  - A factory method is called instead of a constructor
- Moves the responsibility of creating an object from the client to a factory class

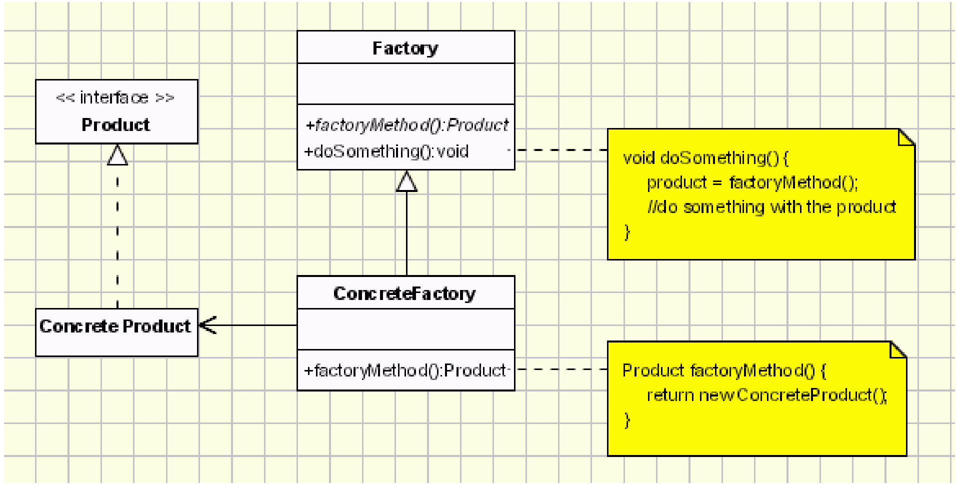


```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

Factory

- **Product** – defines the interface of objects created by the factory
- **ConcreteProduct** – implements the Product interface
- **Factory** – declares a factory method that returns a Product. May have other methods that use the factory method
- **ConcreteFactory** – overrides the factory method to return a ConcreteProduct

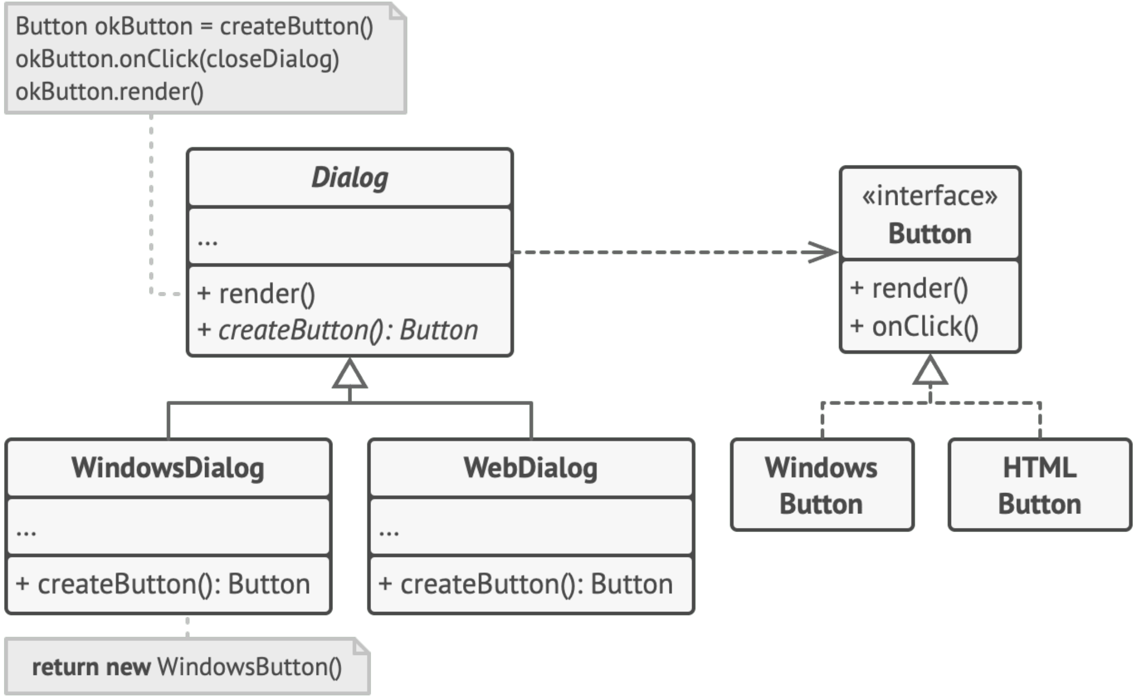


```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

Factory

- Example: cross-platform UI elements

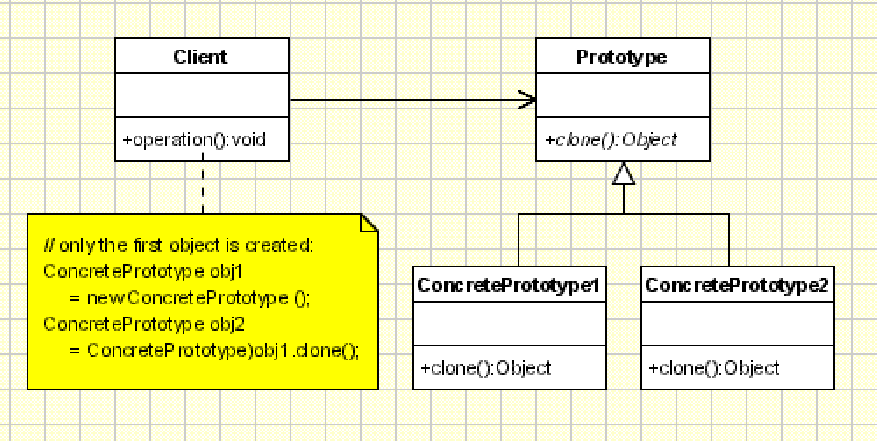


Factory

- Use factory when:
  - a class can’t anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - you want to localise knowledge about object creation
- Pros:
  - Modular expandability – can create new concrete factories and products without breaking client functionality (open-closed principle)
  - Delegates object creation responsibilities to a separate class (single responsibility principle)
  - Straightforward to test
- Cons:
  - Requires more classes than just using a straightforward constructor call

Prototype

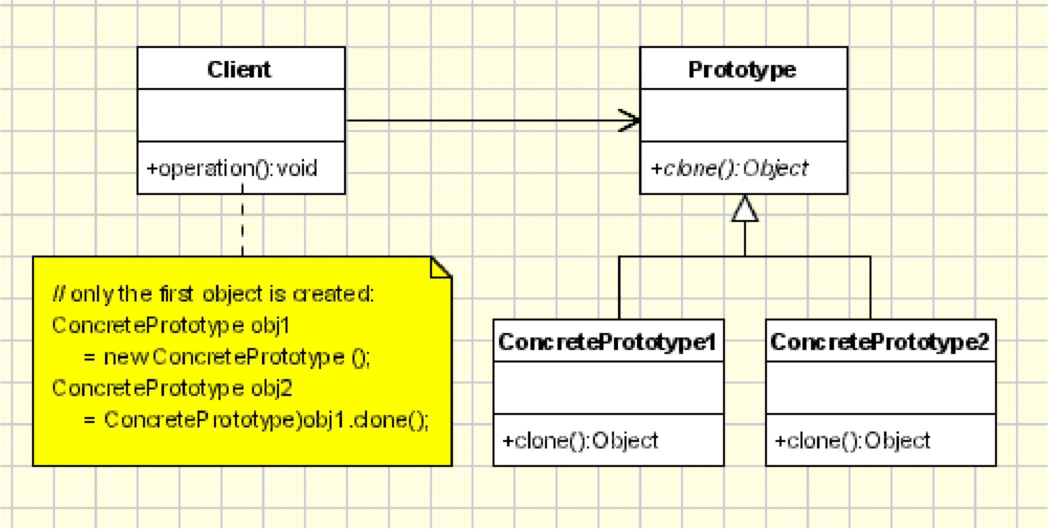
- The **prototype pattern** creates instances of an object by cloning a **prototypical instance**
  - Like factory, prototype abstracts the creation of objects
    - Factory and prototype often used together
  - Cloning existing objects is often computationally cheaper than creating new ones
    - Cloning an object copies its encapsulated attributes as well, which may not be known by the client



```
// only the first object is created:
ConcretePrototype obj1
    = new ConcretePrototype ();
ConcretePrototype obj2
    = ConcretePrototype)obj1.clone();
```

Prototype

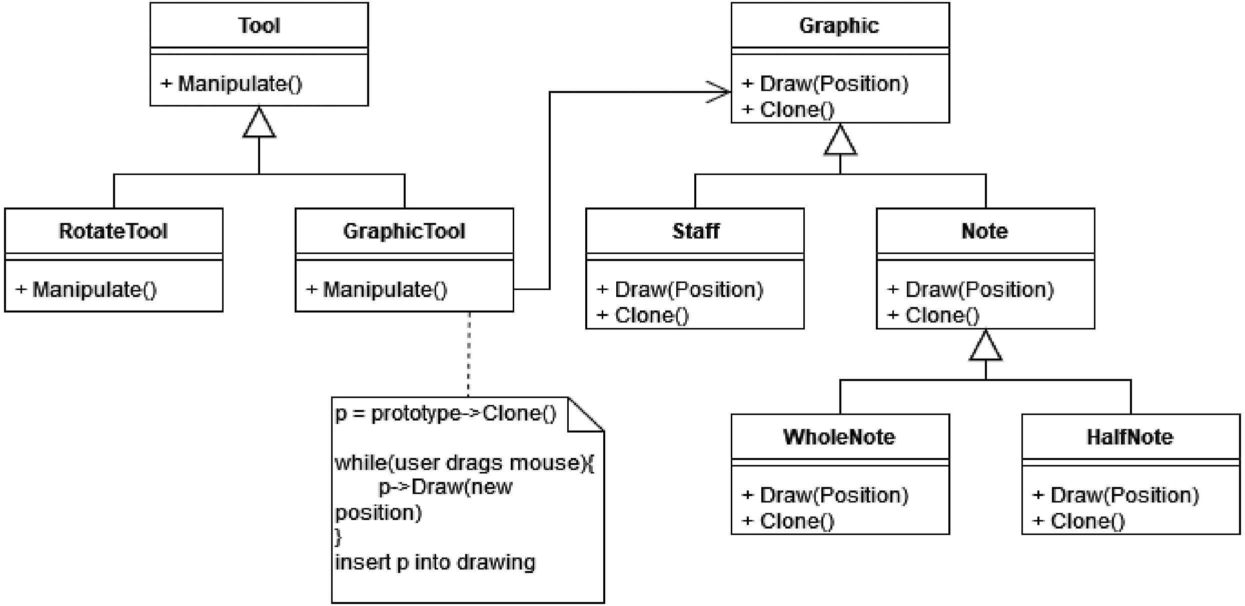
- **Prototype** – defines the interface of cloned objects
- **ConcretePrototype** – implements the Prototype class
- **Client** – creates one instance of the ConcretePrototype, then clones it to make any more



```
// only the first object is created:
ConcretePrototype obj1
    = new ConcretePrototype ();
ConcretePrototype obj2
    = ConcretePrototype)obj1.clone();
```

Prototype

- Example: musical notation software



Tool
+ Manipulate()

RotateTool
+ Manipulate()

GraphicTool
+ Manipulate()

```
p = prototype->Clone()

while(user drags mouse){
        p->Draw(new
position)
}
insert p into drawing
```

Graphic
+ Draw(Position)
+ Clone()

Staff
+ Draw(Position)
+ Clone()

Note
+ Draw(Position)
+ Clone()

WholeNote
+ Draw(Position)
+ Clone()

HalfNote
+ Draw(Position)
+ Clone()

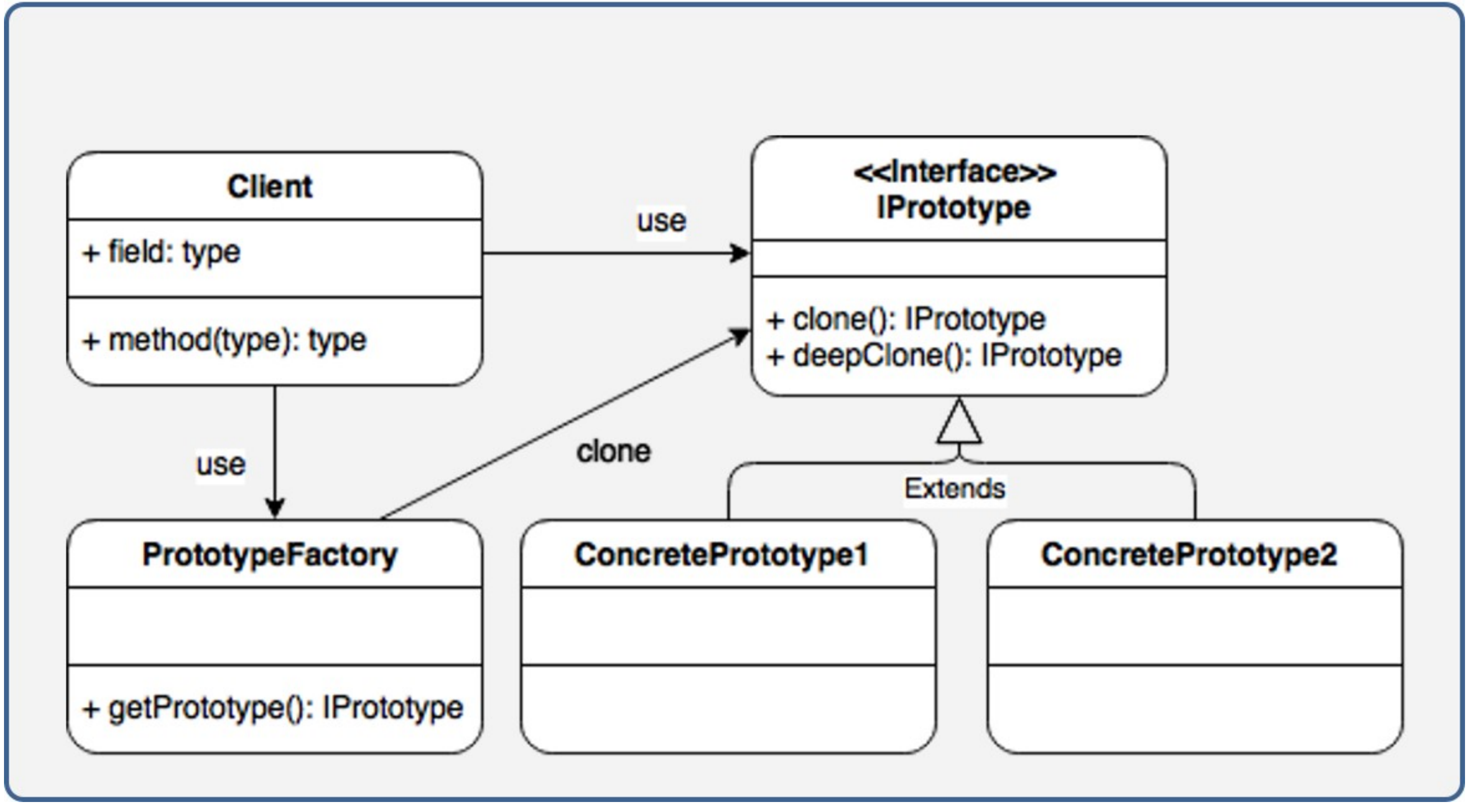
Prototype

- Use prototype when:
  - a system should be independent of how its products are created, composed, and represented
  - a class to be instantiated is selected at run-time (e.g. dynamic loading)
  - you want to avoid building a class hierarchy of factories that parallels the class hierarchy of products
  - instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Prototype

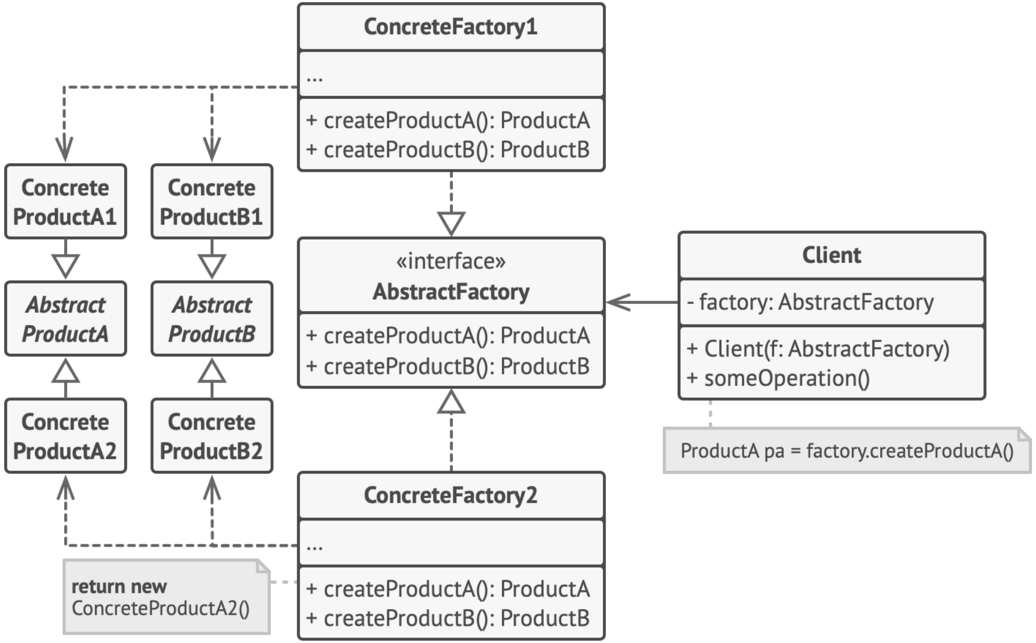
- Pros:
  - You can clone objects without coupling to their concrete classes
  - You can get rid of repeated initialization code in favor of cloning pre-built prototypes
  - You can produce complex objects more conveniently
  - You get an alternative to inheritance when dealing with configuration presets for complex objects
- Cons:
  - Cloning complex objects that have circular references might be very tricky

Combo: Prototype Factory



Abstract Factory

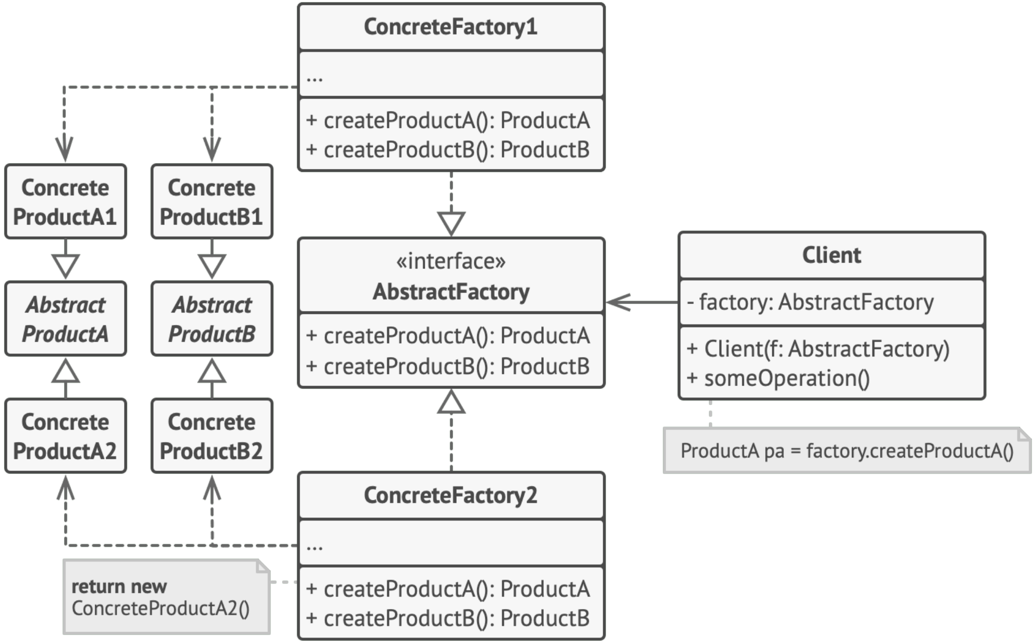
- The **abstract factory pattern**, also known as the **kit pattern**, provides an interface for creating families of related or dependent objects without specifying their concrete classes
- Note: factories can be implemented using factory methods or prototypes (i.e. cloning)



ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
Abstract ProductA
Abstract ProductB
Concrete ProductA2
Concrete ProductB2
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

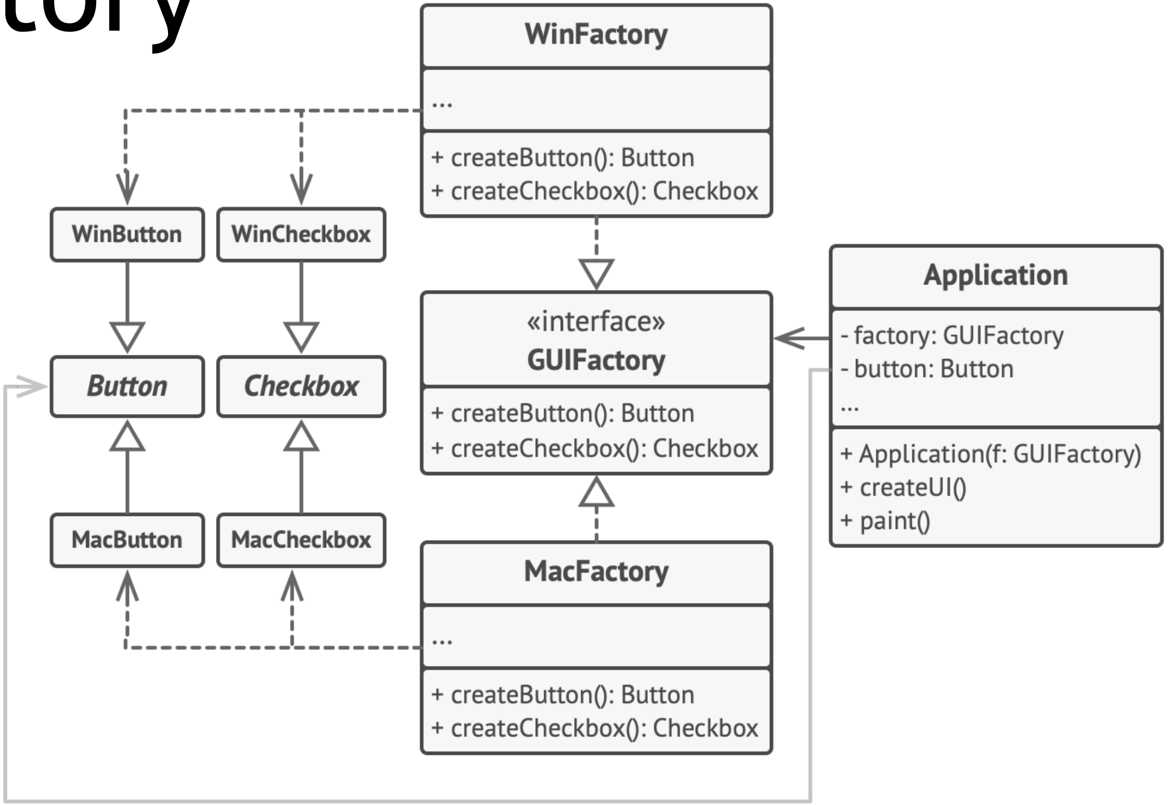
Abstract Factory

- **AbstractProduct** – interface for products
- **ConcreteProduct** – implements AbstractProduct
- **AbstractFactory** – declares an interface for factories that create AbstractProducts
- **ConcreteFactory** – implements AbstractFactory, creates ConcreteProducts
- **Client** – uses interfaces provided by AbstractFactory and AbstractProduct


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
Abstract ProductA
Abstract ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Abstract Factory

- Example: a UI toolkit for different operating systems



Abstract Factory

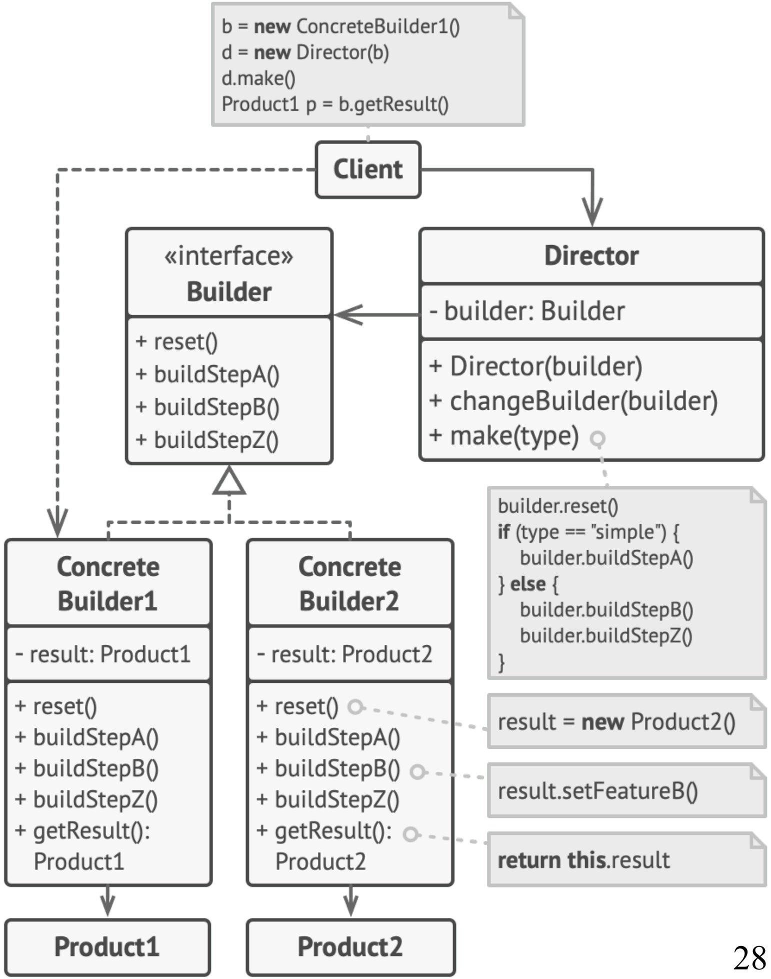
- Use abstract factory when:
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products
  - a family of related product objects is designed to be used together, and you need to enforce this constraint
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory

- Pros:
  - You can be sure that the products you’re getting from a factory are compatible with each other
  - You avoid tight coupling between concrete products and client code
  - You can extract the product creation code into one place, making the code easier to support
    - Single Responsibility Principle
  - You can introduce new variants of products without breaking existing client code
    - Open-Closed Principle
- Cons:
  - The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern

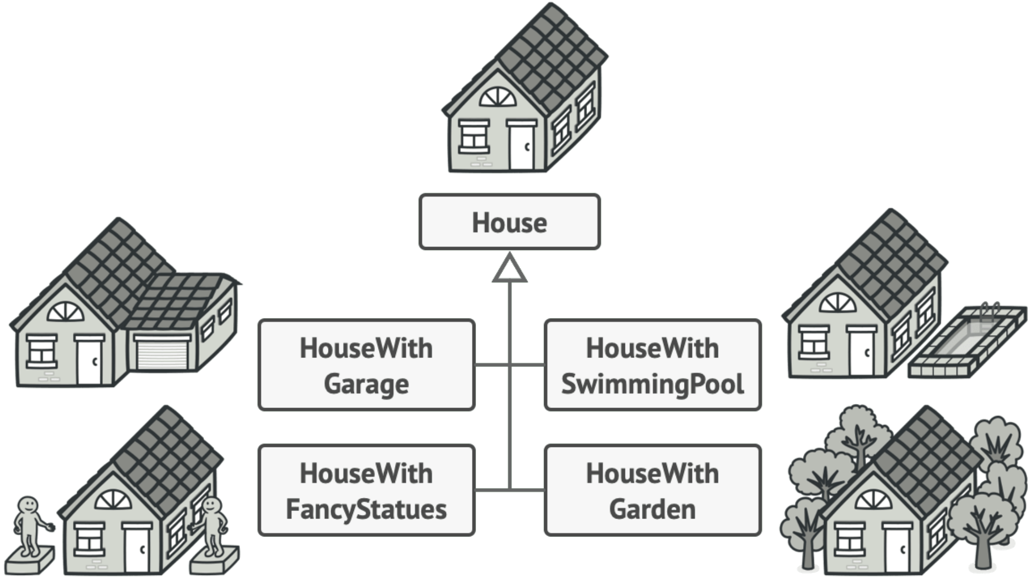
Builder

- The **builder pattern** separates the construction of a complex object from its representation



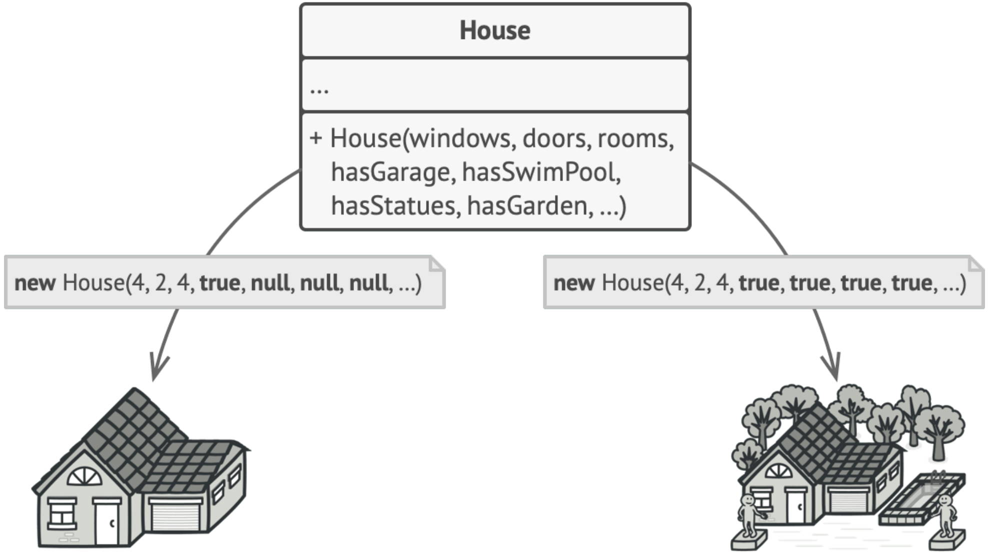
Builder

- Motivation: Consider building a house – houses can have multiple configurations
  - We could represent these configurations using subclasses
    - Number of subclasses spiral out of control



Builder

- Motivation:
  - We could represent these configurations as attributes passed to constructor
    - Object creation becomes very very messy



House

...

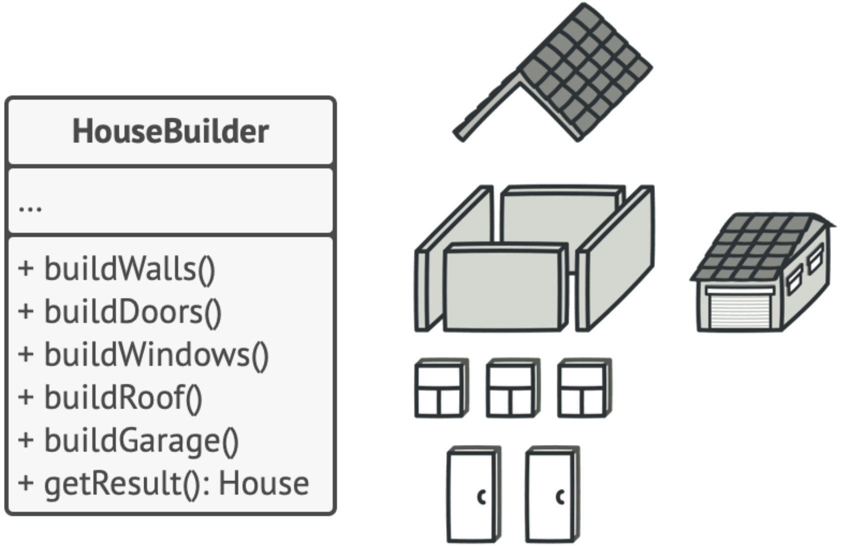
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)

new House(4, 2, 4, **true**, **null**, **null**, **null**, ...)

new House(4, 2, 4, **true**, **true**, **true**, **true**, ...)

Builder

- Motivation:
  - We instead break up object construction into steps and delegate the responsibility of executing these steps to a **builder class**



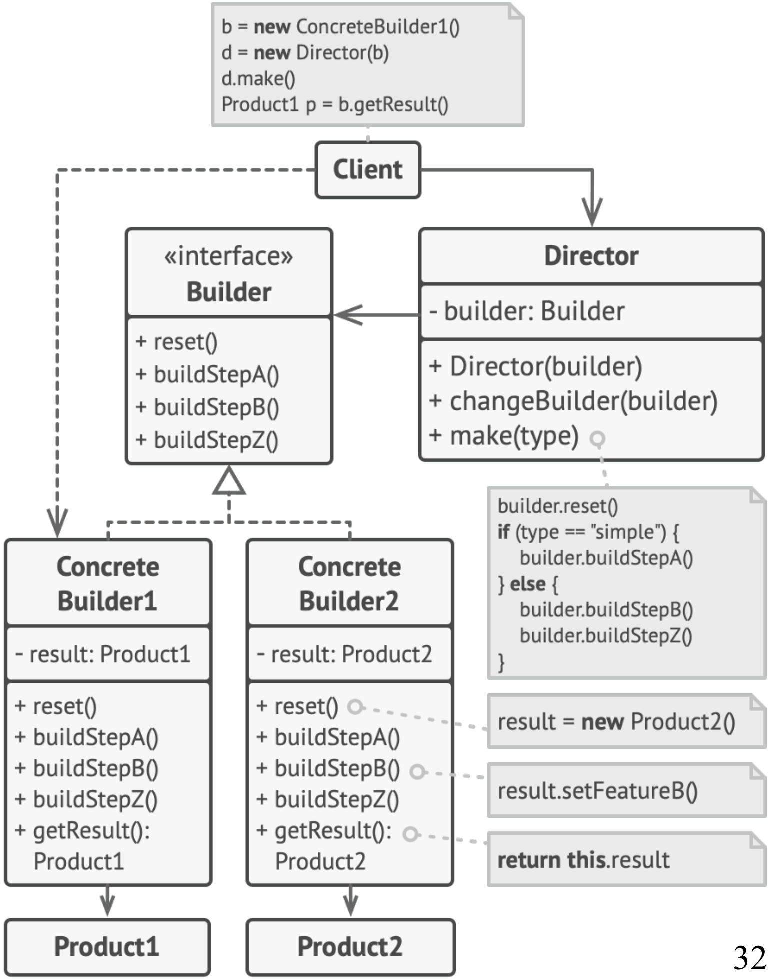
HouseBuilder

...

+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House

Builder

- **Product** – the complex object to be built
- **Builder** – the interface for building Products
- **ConcreteBuilder** – implements Builder to construct parts of Product
- **Director** – optional class that coordinates the Builder



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

Client

«interface»
Builder

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()

Director

- builder: Builder

+ Director(builder)
+ changeBuilder(builder)
+ make(type)

```
builder.reset()
if (type == "simple") {
  builder.buildStepA()
} else {
  builder.buildStepB()
  builder.buildStepZ()
}
```

Concrete Builder1

- result: Product1

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product1

Concrete Builder2

- result: Product2

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product2

```
result = new Product2()
```

```
result.setFeatureB()
```

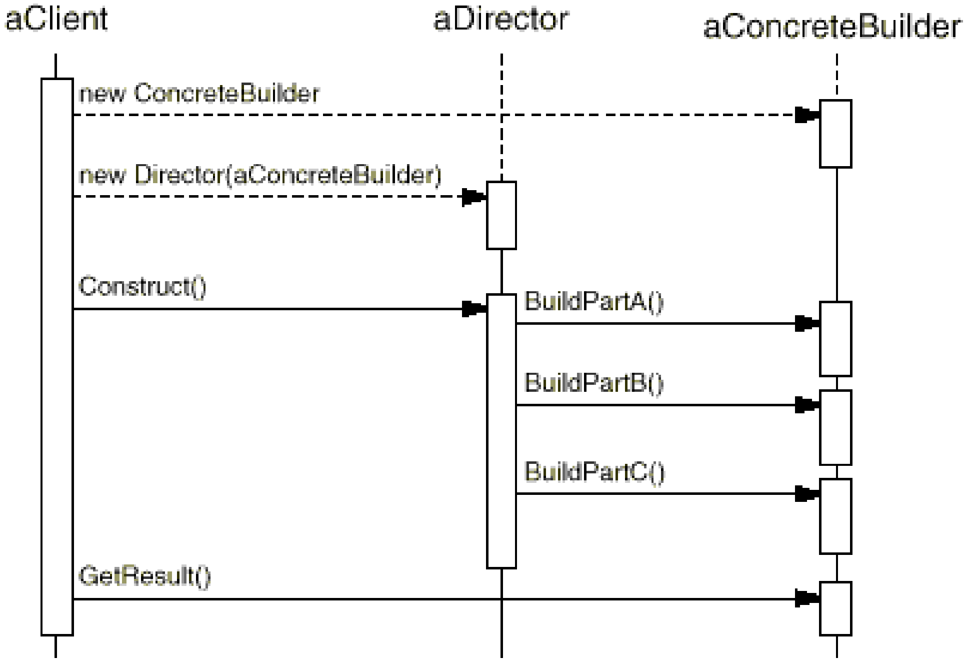
```
return this.result
```

Product1

Product2

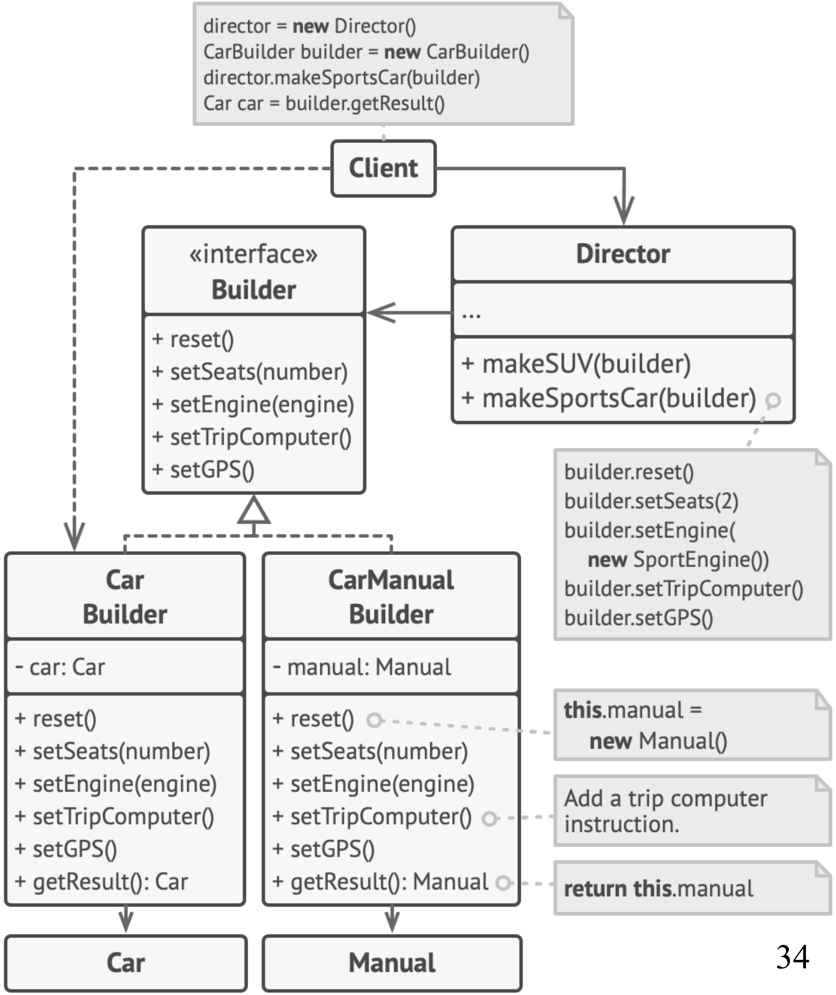
Builder

- Building sequence



Builder

- Example: building cars and car manuals



```
director = new Director()
CarBuilder builder = new CarBuilder()
director.makeSportsCar(builder)
Car car = builder.getResult()
```

Client

Director

...

+ makeSUV(builder)
+ makeSportsCar(builder)

```
builder.reset()
builder.setSeats(2)
builder.setEngine(
   new SportEngine())
builder.setTripComputer()
builder.setGPS()
```

«interface»
Builder

+ reset()
+ setSeats(number)
+ setEngine(engine)
+ setTripComputer()
+ setGPS()

Car
Builder

- car: Car

+ reset()
+ setSeats(number)
+ setEngine(engine)
+ setTripComputer()
+ setGPS()
+ getResult(): Car

CarManual
Builder

- manual: Manual

+ reset()
+ setSeats(number)
+ setEngine(engine)
+ setTripComputer()
+ setGPS()
+ getResult(): Manual

```
this.manual =
   new Manual()
```

Add a trip computer instruction.

```
return this.manual
```

Car

Manual

Builder

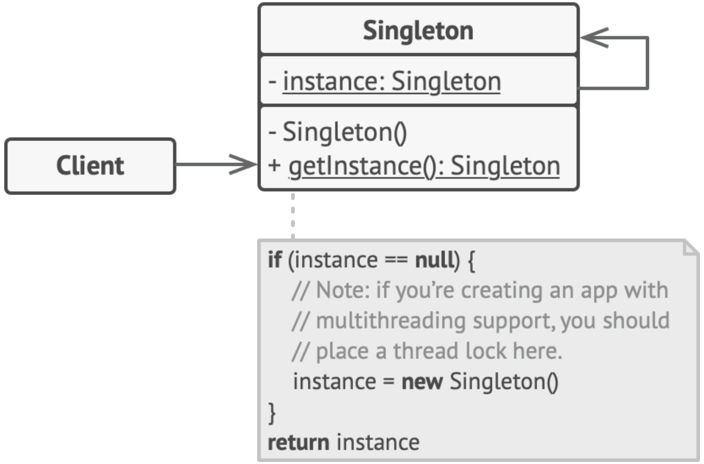
- Use builder when:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed
    - e.g. stone house vs wooden house

Builder

- Pros:
  - You can construct objects step-by-step, defer construction steps or run steps recursively
  - You can reuse the same construction code when building various representations of products
  - You can isolate complex construction code from the business logic of the product
    - Single Responsibility Principle
- Cons:
  - The overall complexity of the code increases since the pattern requires creating multiple new classes

Singleton

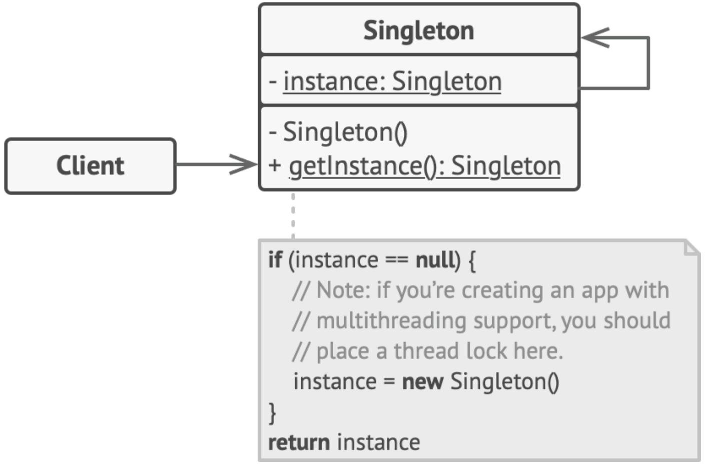
- The **singleton pattern** restricts the instantiation of a class to a single instance
  - It's often useful to have only one instance of a class. Singletons let us ensure that a class isn't needlessly instantiated more than once.
- Often considered an anti-pattern



```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

Singleton

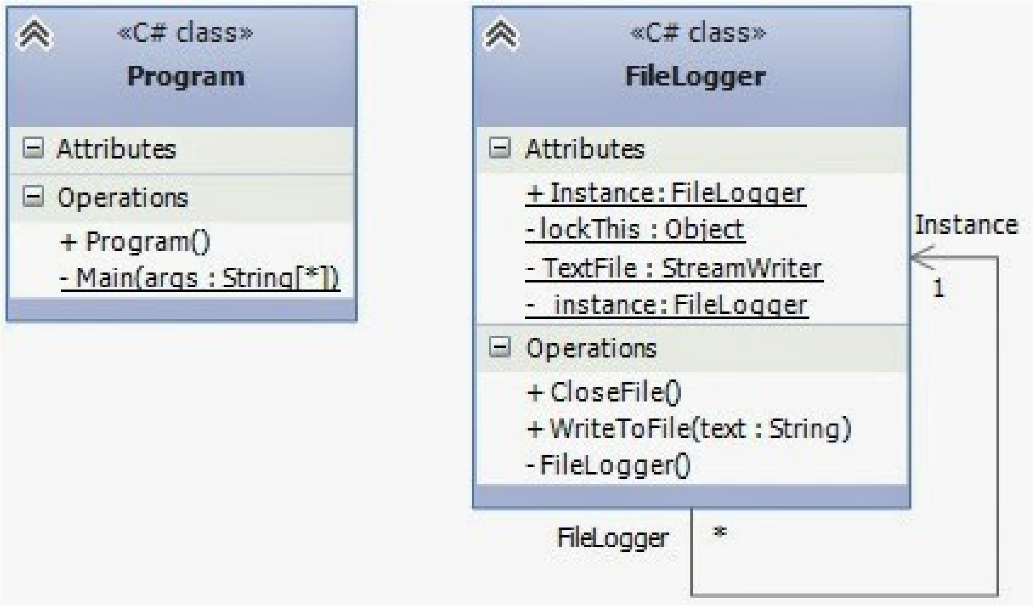
- Implementation:
  - Default constructor is made private
  - Instead, use a static creation method that calls the default constructor the first time and returns the instance on every subsequent call



```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Singleton

- Example: file logger



Singleton

- Singletons vs Global Variables
  - Both singletons and global variables are globally accessible
  - However, singletons can encapsulate and hide information that global variables can’t
  - Global variables can clutter a namespace with unnecessary variables
  - Singletons allow for lazy allocation (only allocate memory when needed), whereas global variables always consume resources
  - Singletons can be subclassed

Singleton

- Use singleton when:
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

Singleton

- Pros:
  - You can be sure that a class has only a single instance
  - You gain a global access point to that instance
  - The singleton object is initialized only when it’s requested for the first time
- Cons:
  - The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other
    - In general, globally accessible components (global variables, singletons, etc.) can result in tighter coupling
  - Hard to execute correctly in multi-threaded environments
  - Hard to create mock singletons when testing
  - Isn’t future proof – what if we decide we need more than one?
  - Violates single responsibility principle – singleton does whatever it’s supposed to do, but also ensures there is only one of it

Other Creational Patterns

- Not in the original GoF list:
  - **Dependency Injection** – object receives objects which it depends on
  - **Lazy Initialisation** – object creation is delayed until the object is actually needed in order to reduce initial load
  - **Multition** – generalises the singleton to multiple instances
  - **Object Pool** – rather than creating and destroying objects, recycle them from a pool

Design Pattern Pitfalls

- Don’t design for patterns!
  - Trade-off between getting a product out quickly and optimising that product
  - Optimisation is secondary
  - Initial designs should be refactored to patterns
    - e.g. Designs often start out using factories (as they are simple to use and easily customisable) and evolve towards abstract factories, prototypes or builders as designs are refined and optimised
- Uncritical use of design patterns can turn them into anti-patterns
- The need for some design patterns can be eliminated through language features
  - Different languages and paradigms will change how you approach design patterns