Topic 5 - Introduction to Design Patterns
Creational Patterns (Gamma et al.)

The Pattern Approach

Originated from work of Christopher Alexander in 1970s

- Alexander noticed recurring themes in architecture
- "Pattern" coined to represent these commonalties

OXFORD DICTIONARY DEFINITION OF "PATTERN"

- "a regular or logical form, order, or **arrangement of parts**"
- "a model or design, e.g. of a garment, **from which copies can be made**"
- "an **example** of excellence; an ideal; a model"

Patterns in Software Design

- Have emerged from within the *OO design* community
- First popularised in Computer Science by *Eric Gamma et al. in:*
  Design patterns : elements of reusable object-oriented software Erich Gamma c1995 (avail. in library)
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (AKA the Gang of Four)
  - proposed an interpretation (**design patterns**) of Alexander patterns for software and proposed a format for design patterns (Intent, Motivation, Applicability, Participants, Collaborations, Consequences)
  - proposed a classification for describing different design patterns
  - prepared a catalogue of important design patterns
- Describe the relationships *within* and *between* the *parts*, not just the parts themselves

Benefits of Patterns

- **supports reuse** of *routine design* knowledge
- allows developers to communicate **successful designs**
- allows designer to apply patterns to a problem under investigation in order to **narrow design space** of eventual outcomes
- designer can focus on architectural issues which have been **proven to work**
- novice designer can **act more like an expert**

Creational Patterns

Factory Method

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

Abstract Factory

provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder

separate the construction of a complex object from its representation so that the same construction process can create different representations.

Prototype

specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton

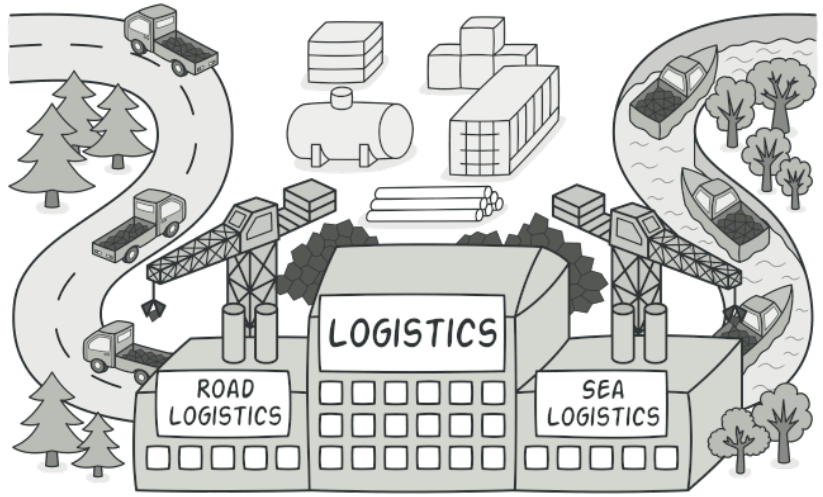
ensure a class only has one instance, and provide a global point of access to it.

Creational Patterns Overview

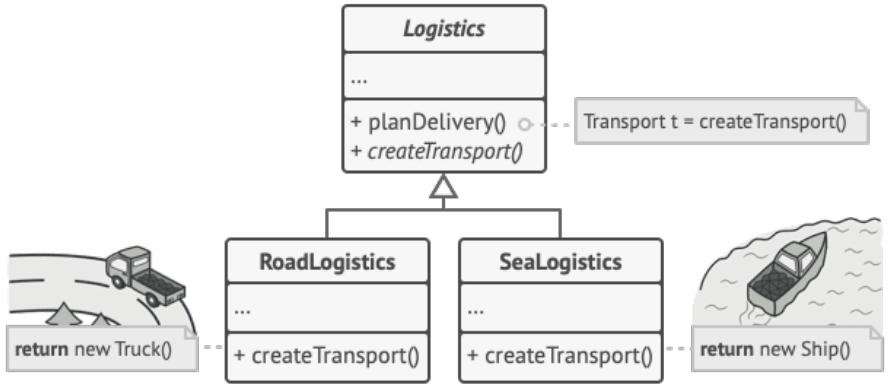
- The standard way to create a Java object is to instantiate an instance of a class with a constructor
- This assumes clients know which concrete class to construct in advance (may not be true for extensible frameworks or where new subtypes are added after the creation of the client code)
- Where default (no arg) constructors are unavailable, clients must have the necessary information to pass as constructor parameters (again problematic for as yet unknown classes)
- The creational design patterns are used to alleviate these two challenges
- Extends the advice given by the GRASP *Creator* and *Information Expert* patterns

Factories

- What is a factory?
  - Some code other than a constructor responsible for creating one or more objects for clients
- Separates knowledge of “how to use” from knowledge of “what exactly to create”
  - Client knows first part, factory second part
- Factory provides objects that meet an interface **but** client need not know concrete class of that object
- Facilitates caching, pooling and reuse of instances by the framework without client intervention (closely related to Flyweight pattern)
- Factories can be realised in a number of forms
  - factory method (often static) within a class to create instances of that class
  - factory method in a separate Factory class independent from the classes it creates
  - abstract Factory which creates instances of related classes (product families)



https://refactoring.guru/design-patterns/factory-method



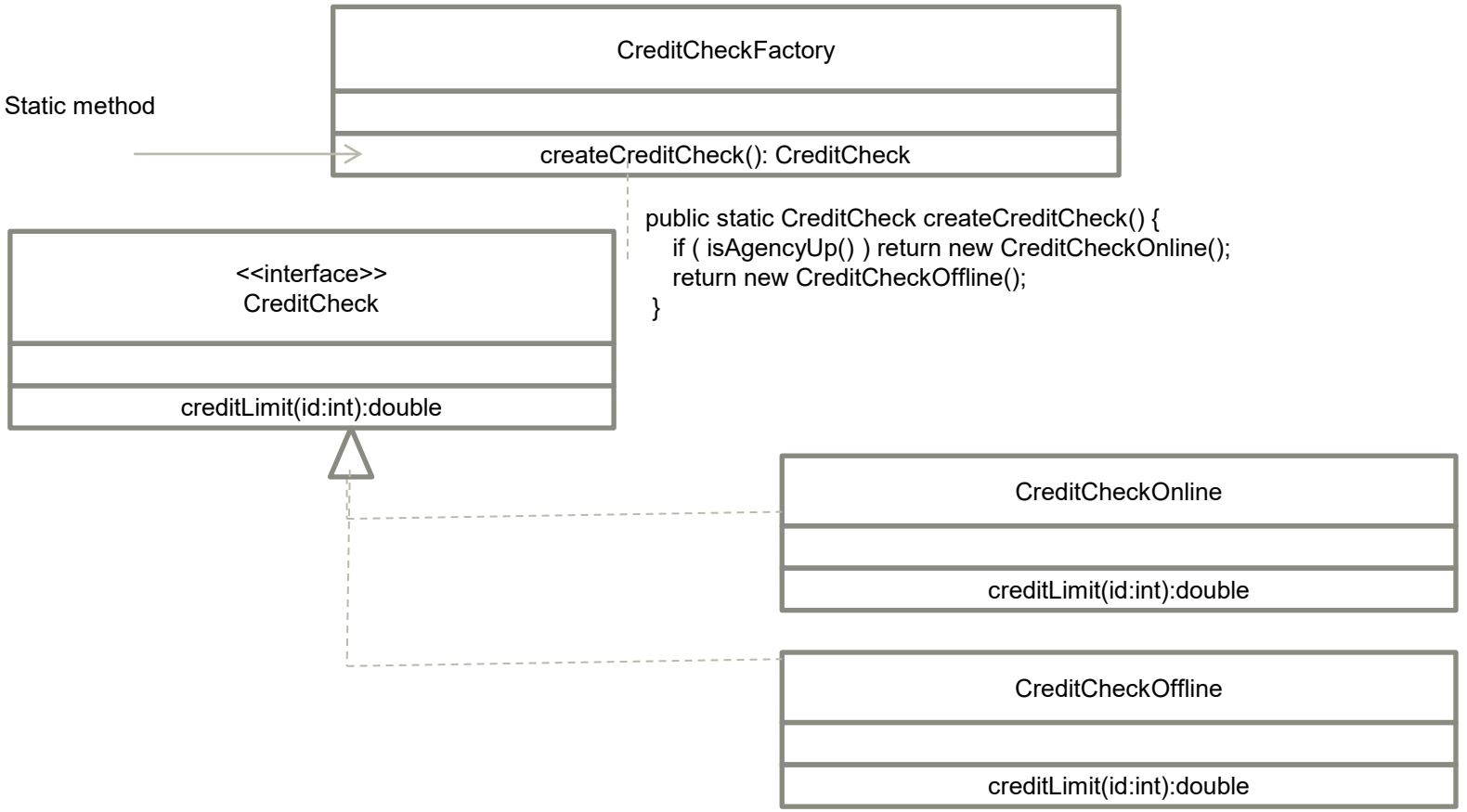
Factory Method

- A factory method creates and returns an object of some concrete type that implements a specific interface
- Not all creation methods are factory methods
  - For example if they return pre-specified concrete classes
- Java API provides factory methods in several classes e.g.
  - `java.util.Collection.iterator()` returns an `java.util.Iterator`
  - `java.util.Calendar.getInstance(…)`
  - `java.text.NumberFormat.getInstance(…)`
  - `java.util.logging.Logger`
  - constructors are protected
  - how do we know which concrete types these are?
  - why are factory methods useful in these classes (the last three are probably more obvious than the first)

Factory Method: Custom Example

- Suppose a company wants to check with the credit agency to determine the credit limit to be allowed for customers.
- It is possible the credit agency may be offline at times.
- The client does not have (and should not need to know) this information.
- In such cases, the credit limit is to be determined using a series of questions in a dialog.
- To handle these cases, the `CreditCheckFactory` class will create either `CreditCheckOnline` or `CreditCheckOffline` objects both of which implement the method `creditLimit()`
- The following diagram shows a simple design, you may wish to implement this as an exercise
- See also `gamma.maze.app.MazeGame.createMaze()` factory method

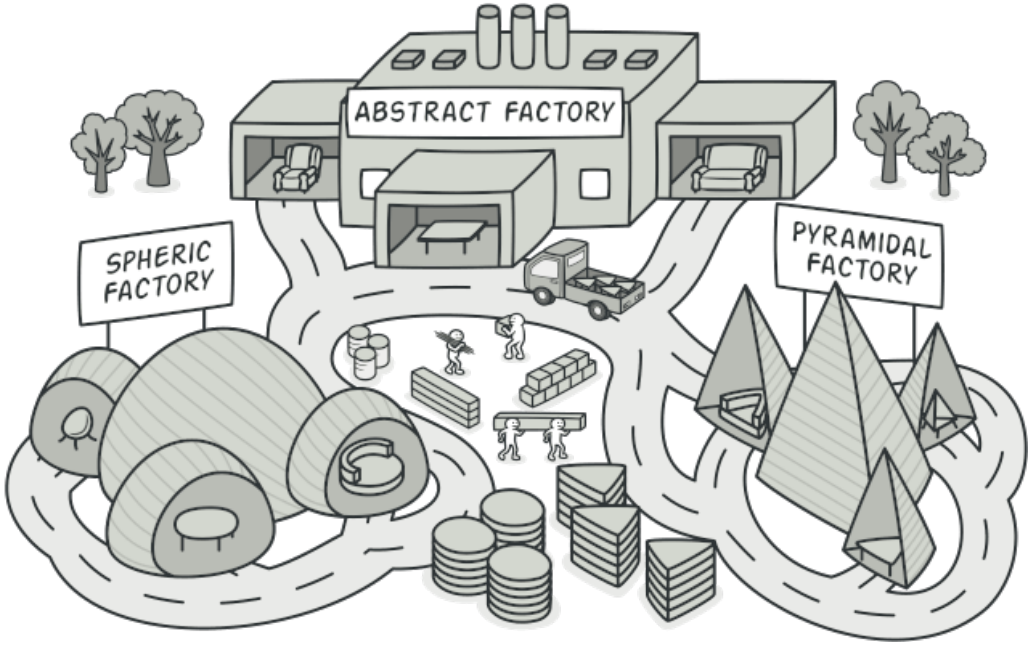
Example: Factory Method (in Factory Class)



From: Steven John Metsker & William C. Wake, Design Patterns in Java, Addison-Wesley, 2006

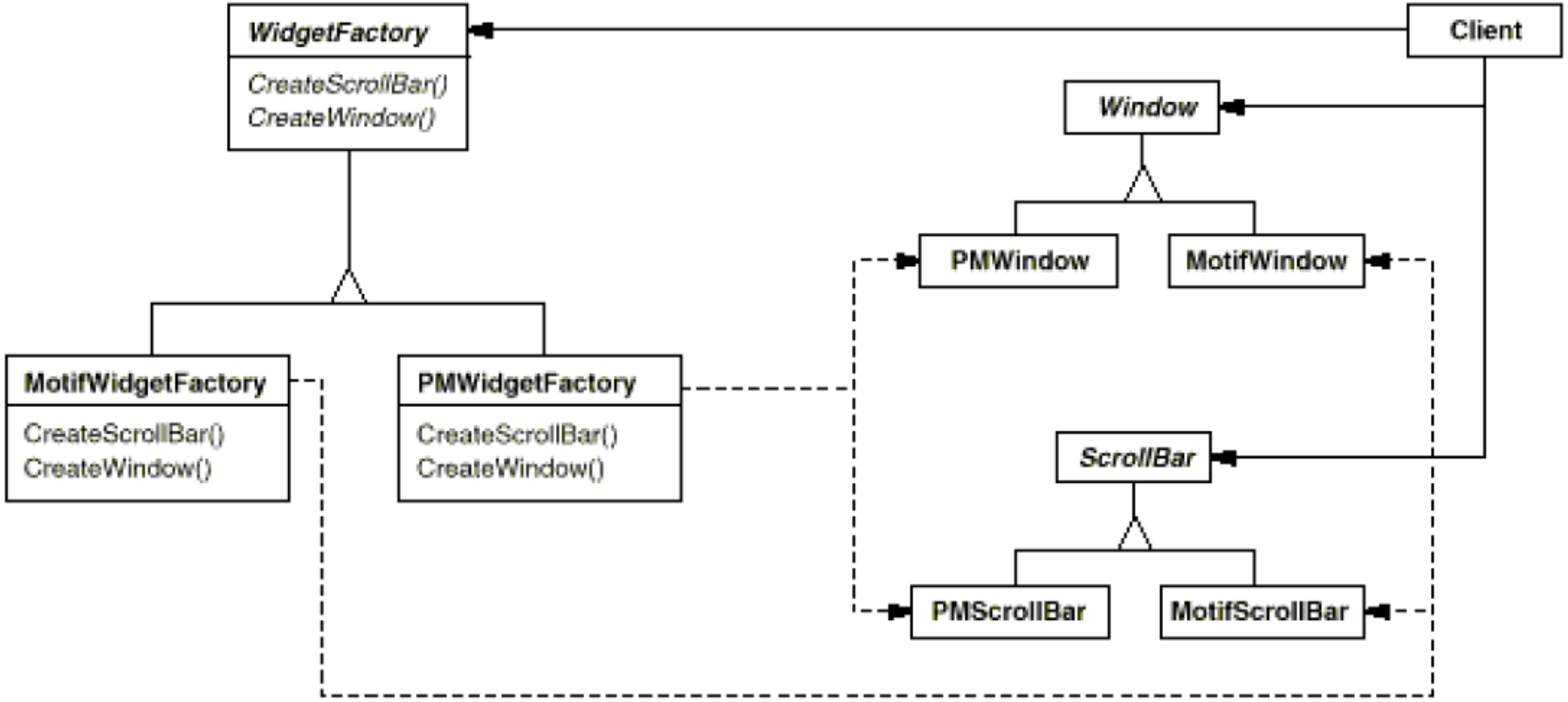
Abstract Factory Pattern

- **Problem –** You have families of objects that can be specialised but objects of one specialised family must always be used alongside objects of the same family.
- **Solution**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes to ensure that the family (implementation) can change but the correct (compatible objects) are always created.
  - Represent each object type as AbstractProducts (there may be many). An AbstractFactory declares methods for creating each type of AbstractProduct. ConcreteFactory creates products of the same family (i.e. related ConreteProduct types)
- **Advantage** – Helps to reduce coupling between client and object implementations and reduces errors with mismatched products. Also facilitates extensibility by making the creation of new product types independent of the client.



https://refactoring.guru/design-patterns/abstract-factory

Abstract Factory Pattern (example case)



From: Erich Gamma et al., *Design patterns : elements of reusable object-oriented software* , Addison-Wesley, 1995

Abstract Factory Pattern: Consequences

- Simple example

```
factory.gui.*;
```

- See **Maze** example from Gamma textbook

```
gamma.factory.*;
```

- It helps to isolate concrete classes. Client creates instances through the factory and manipulates them through their abstract interfaces.
- It makes exchanging product families easy – as the concrete factory is instantiated only once.
- It promotes consistency among products – product objects in a family are designed to work together.

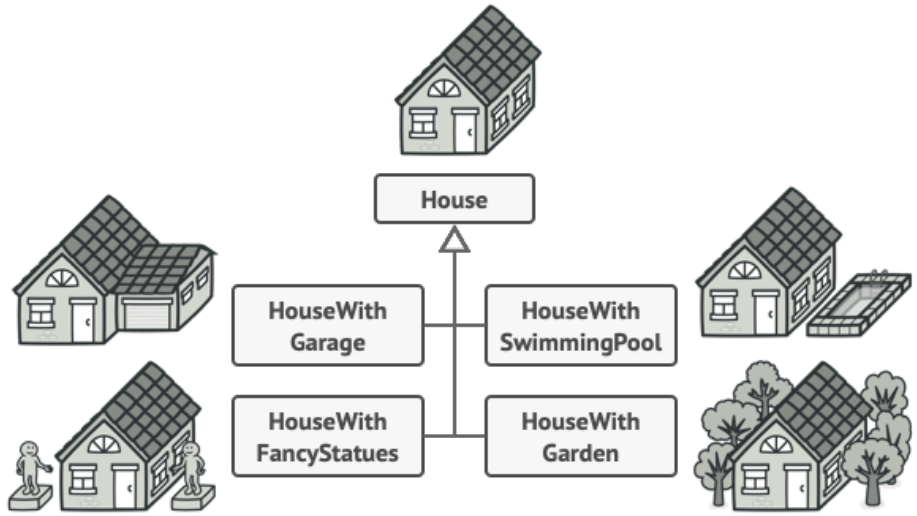
Builder

Purpose:

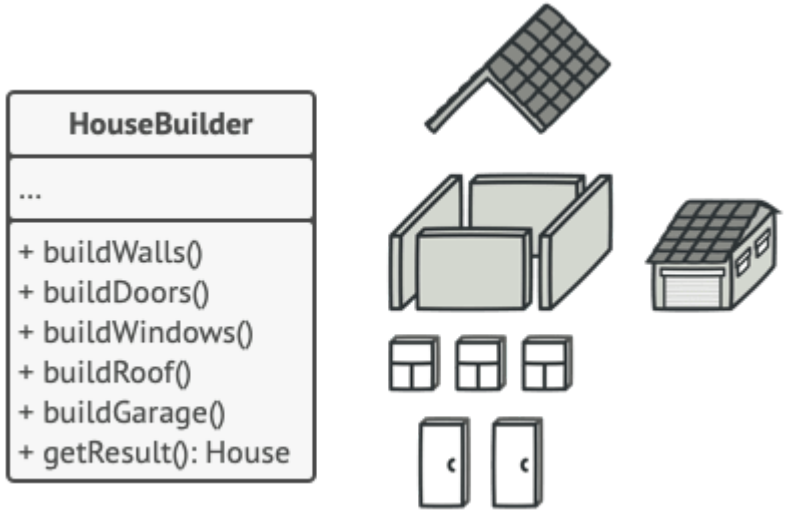
- Separate the construction of a complex object from its representation so that the same construction process can create different representations

Application:

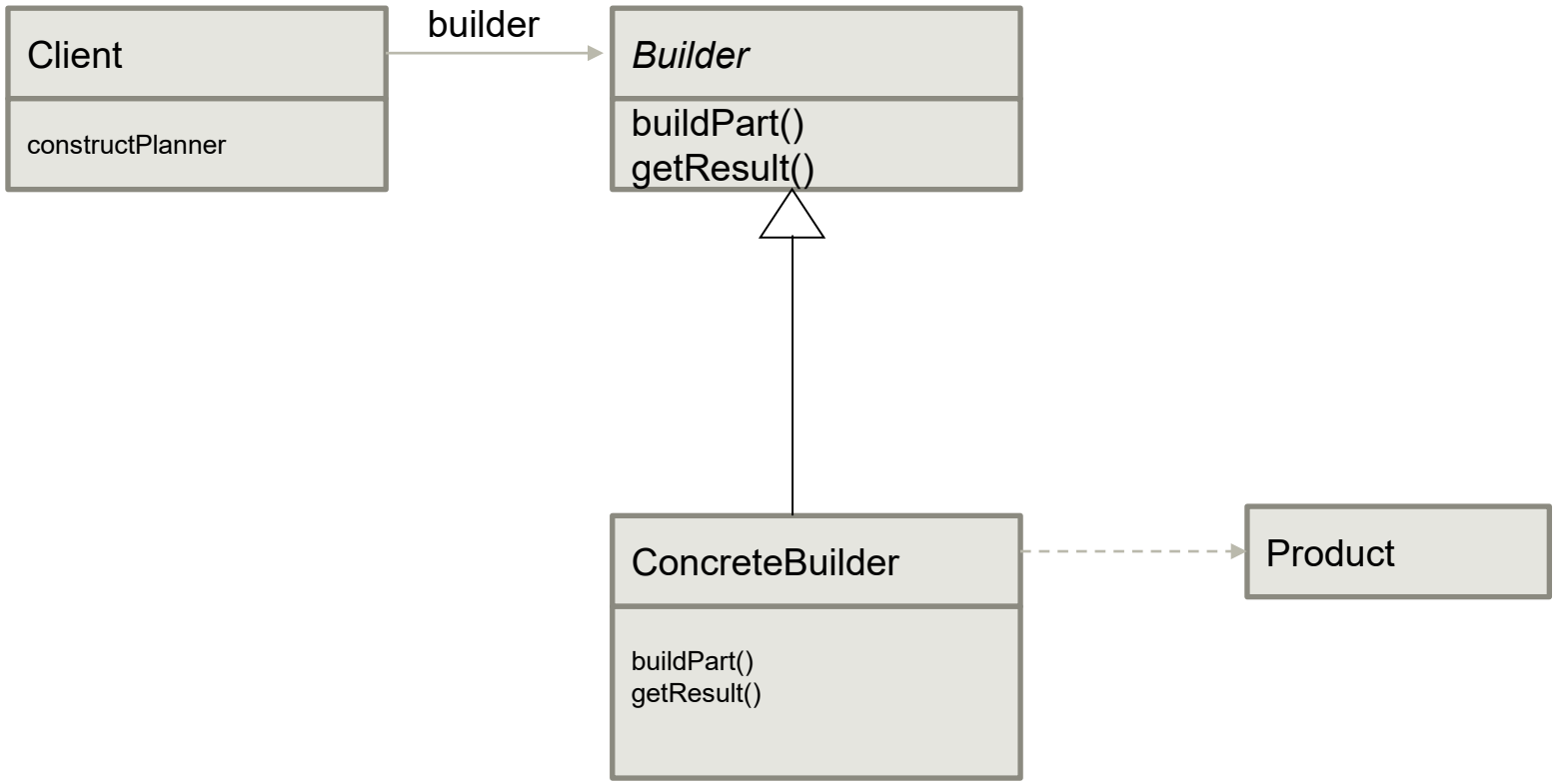
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations of the object concerned
- This pattern is similar to AbstractFactory except that it concentrates on Building an object *piece by piece* rather than families of related objects
- `java.lang.StringBuilder` provides a Java API example of the builder pattern where methods such as `append()` build the string piece by piece and `toString()` returns the final `String`
  - see override method `gamma.maze.site.Room.toString()` for an example



https://refactoring.guru/design-patterns/builder



Structure


Client
constructPlanner
builder
Builder
buildPart()
getResult()
ConcreteBuilder
buildPart()
getResult()
Product

Builder: Example

A Scenario

You've been asked to build a vacation planner for GetAways Theme park. Guests can choose a hotel and various types of admission tickets, restaurant reservations and special events.

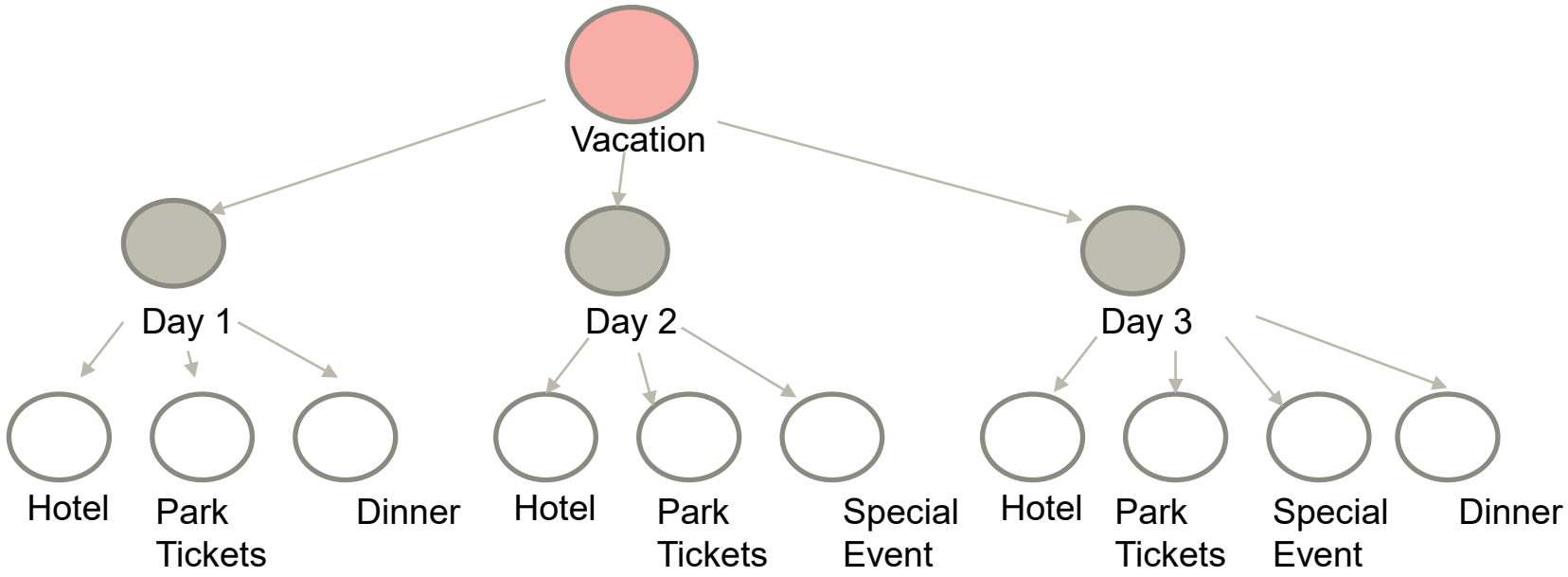
The following slides show a basic design (you may wish to implement it as an exercise!)

See `gamma.builder.*;` for a provided source code example of Builder

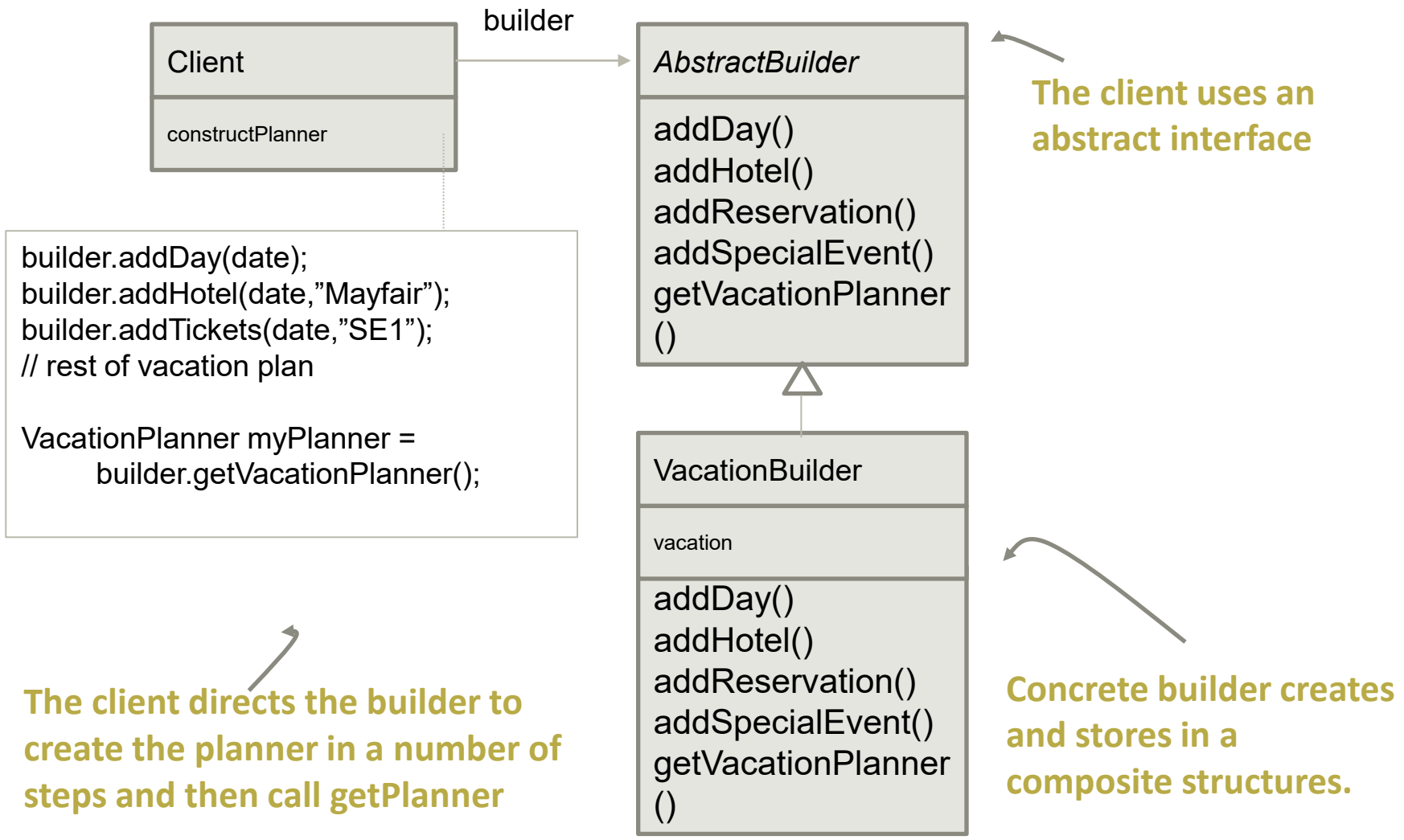
Builder: Problem

You need a flexible structure that represent all variations (different number of days, activities and events).

How can a simple way be provided for a client to create such complex structures?



A Solution using the Builder Pattern



Consequences

- It let's you vary the product's internal representation.
- It isolates code for construction from representation
- Unlike other creational patterns which constructs the product in one shot, the Builder pattern constructs the product step by step
- It removes the need to have large numbers of overridden constructors for many different combinations
- This gives the client a lot of flexibility in terms of which steps are necessary, the order of steps etc.
- Product is retrieved only when construction is complete i.e. all creation steps have been performed

Prototype Pattern

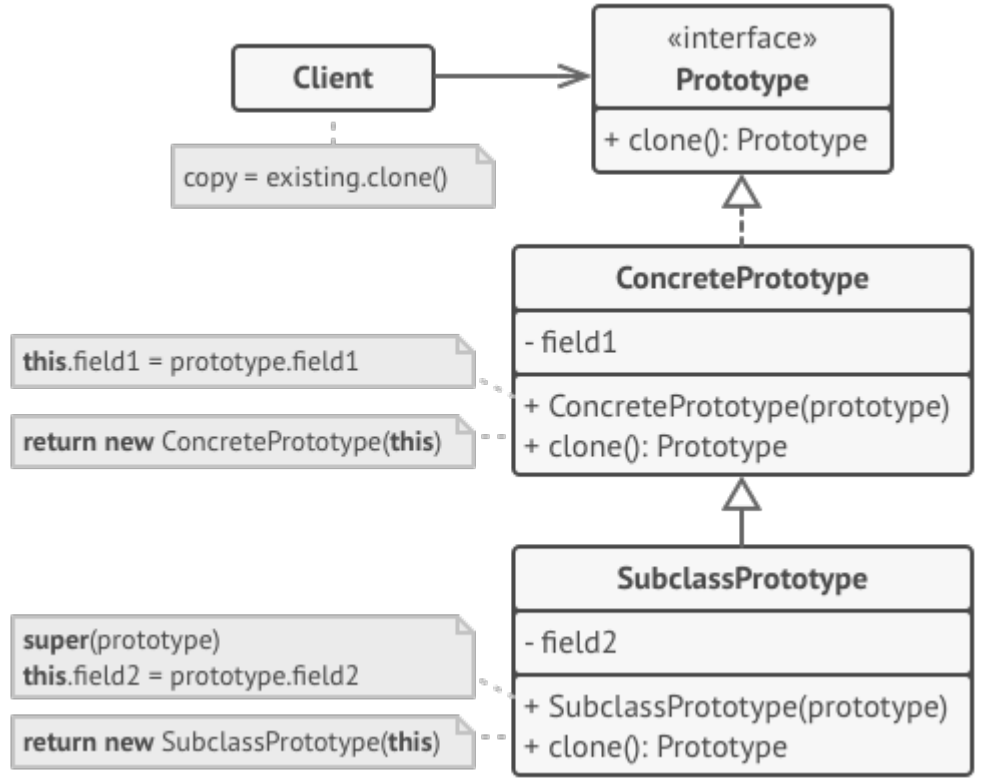
Purpose

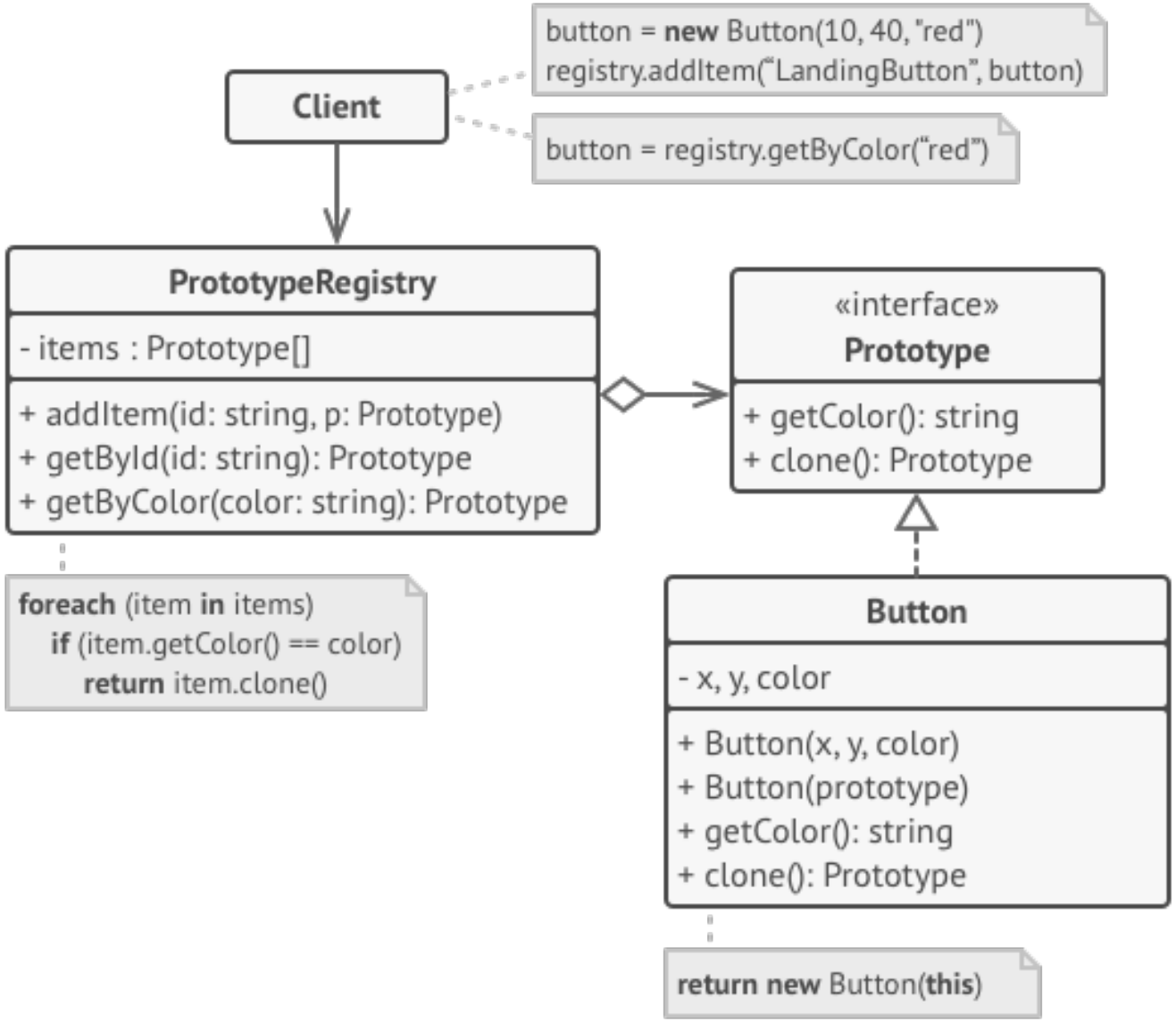
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Application

- Allows objects to be created even when class name is not known
  e.g. a generic framework that can use previously unknown client supplied concrete classes
- Allows the addition of new classes at runtime using reflection
- When instances of a class have only a few different states it may be convenient to pre-create prototypes and clone them rather than instantiating them manually in the calling code each time with the appropriate state.
- To avoid building a class hierarchy of factories that parallels the class hierarchy of products
- As with other creational patterns Builder, AbstractFactory and Factory method, Prototype conceals object creation from clients

https://refactoring.guru/design-patterns/prototype





https://refactoring.guru/design-patterns/prototype

Some Notes on Cloning

- When cloning we must consider the implication of deep copy versus shallow copy
- Generally we need a deep copy so that all referenced objects of the object being cloned are also duplicated (otherwise the objects are not fully independent)
- If a class contains only primitive fields or references to immutable objects then a shallow copy is sufficient
- Java provides a `clone()` method in `java.lang.Object` which by default provides basic shallow copy
- classes must implement `Cloneable` interface to use the clone() method or exception is thrown
- Java clone implementation is considered limited so generally safer/better to write your own copy semantics (see http://www.artima.com/intv/bloch13.html)
- java Serialization provides an easy deep copy mechanism (but not very efficient)
- see `gamma.source.MazePrototypeFactory` example for basic use of `clone()`
- see `gamma.source.ReflectiveMazePrototypeFactory` example for basic use of Java reflection

Singleton

Purpose:

- Ensure a class has only one instance

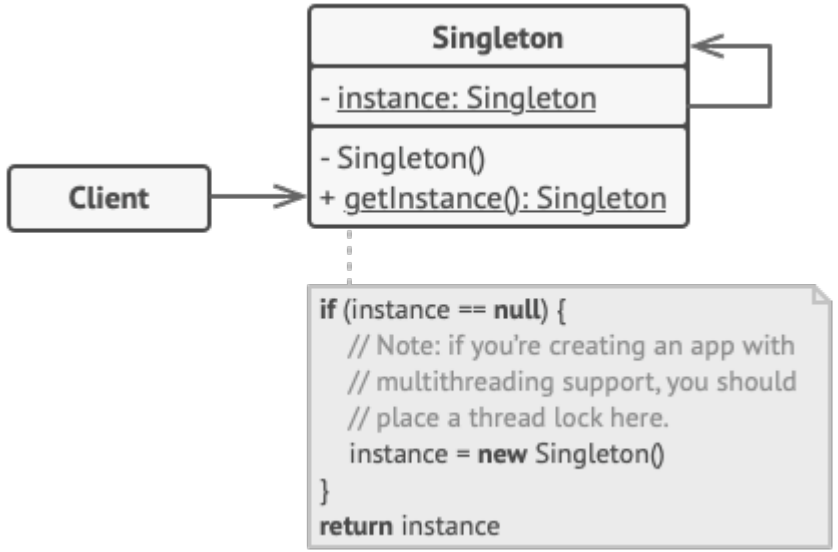
Application:

- Important for some classes to have only one instance (print spooler, file system, window manager, main application Frame, application model etc.)
- It ensures that class has only one instance and that the instance is easily accessible

Disadvantages:

Coupling becomes implicit and is not as easy to see which classes use the singleton (other than doing a search on the Project)

Can make testing harder since you an no longer create multiple instances to batch different properties and tests (need to run each test separately)



https://refactoring.guru/design-patterns/singleton

Singleton Quiz

```
class Singleton {
   private static Singleton uniqueInstance;
   static int count = 0;
   private Singleton() { count++; }
   public static Singleton getInstance()
   {  if (uniqueInstance == null)
         uniqueInstance = new Singleton();
      return uniqueInstance;
   }
}
```

```
public class TestSingleton {
   public static void main(String args[])
   {  Singleton s1 = Singleton.getInstance();
      System.out.println("Number of Singletons " + Singleton.count);
      Singleton s2 = Singleton.getInstance();
      System.out.println("Number of Singletons " + Singleton.count);
   }
}
```

What will be the output?

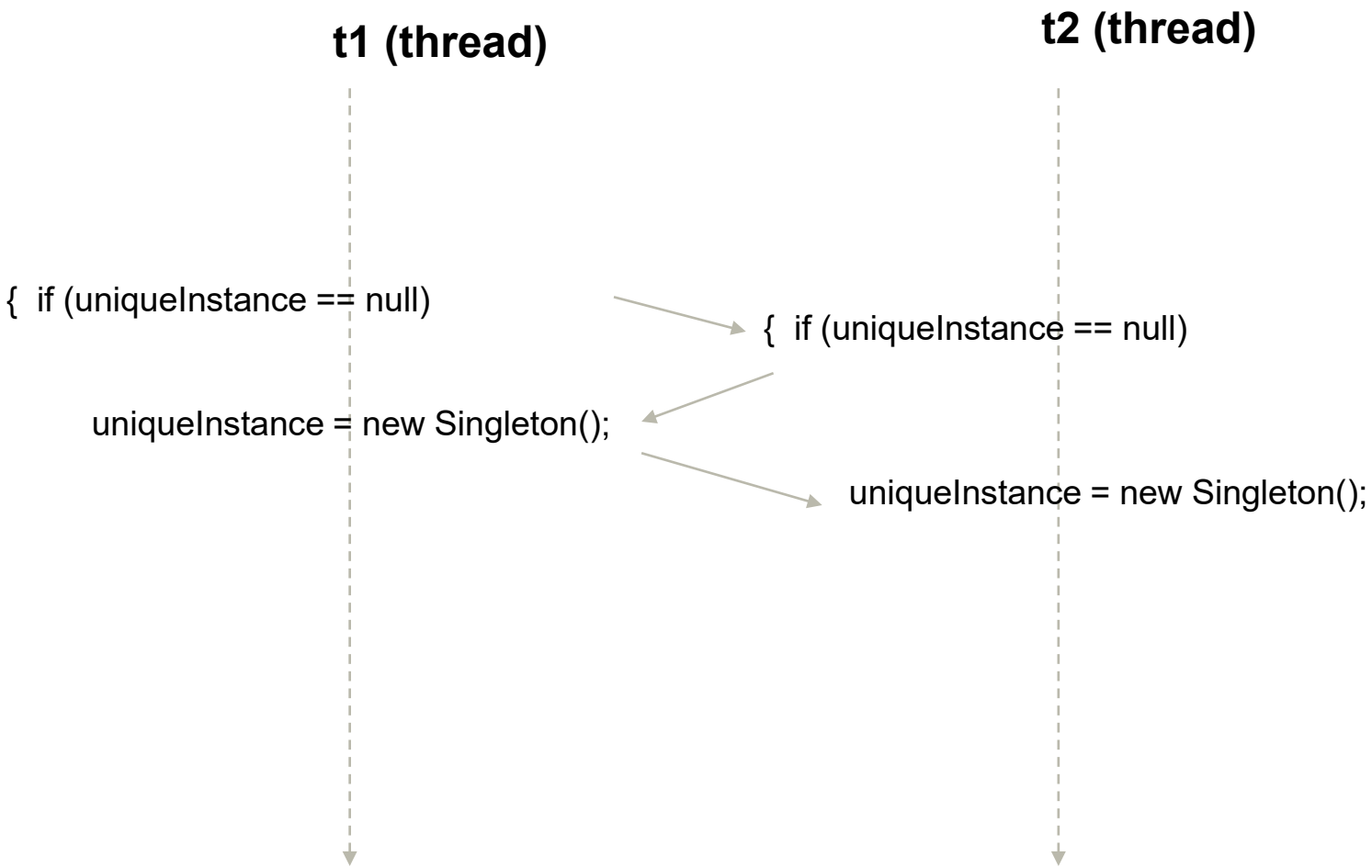
Singletons and Threads

```
public class ThreadFailUsingSingleton
{
  public static void main(String[] args)
  {
    // quick and dirty test using anonymous inner classes
    new Thread()
    {
      public void run()
      {
        Singleton.getInstance();
        System.out.println("Number of Singletons " +
              Singleton.count);
      }
    }.start();

    new Thread()
    {
      public void run()
      {
        Singleton.getInstance();
        System.out.println("Number of Singletons " +
              Singleton.count);
      }
    }.start();
  }
}
```

This program using two threads may end up creating two Singleton objects . Explain why?

Singletons and Threads



Thread Safe Singleton

```
public class ThreadSafeSingleton
{
   private static ThreadSafeSingleton uniqueInstance;
   // other useful instance variable here
   private ThreadSafeSingleton() {  }
   public synchronized static Singleton getInstance()
   {
      if (uniqueInstance == null)
      {        uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }
   // other useful methods here
}
```

For a more efficient approach that does not require synchronization see

http://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom

Ensures no two threads run this method at the same time

Creational Patterns: Summary

- Creational design process abstract the instantiation process – it decouples the clients from the way objects are created composed and represented.
- Creational patterns gives the flexibility in what get created, who creates it, how it is created and when it is created.
- It simplifies object creation for clients.
- Builds upon GRASP Creator and information expert since you still need to access the factory and pass it the appropriate information.