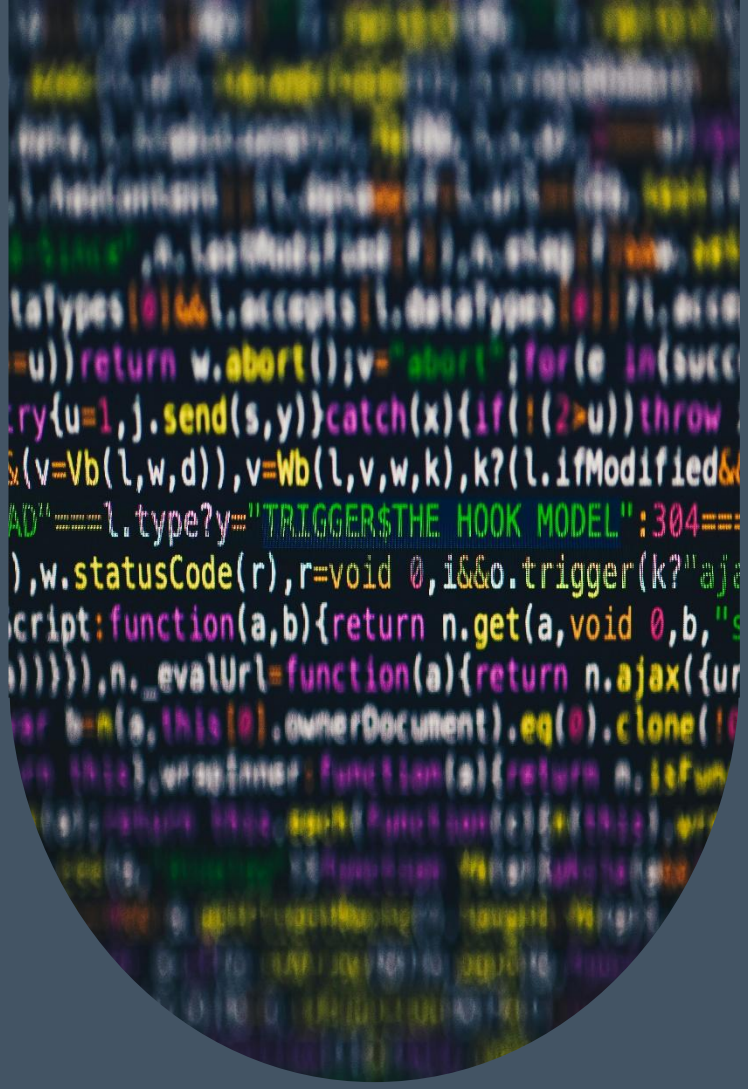
Patterns in Software Development

Architectural Patterns,
Design Patterns,
and Idioms



SETU
Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

setu.ie
INSPIRING FUTURES

Motivation

- Developing software is hard
- Designing reusable software is more challenging
- finding good objects and abstractions
- flexibility, modularity, elegance  reuse
- takes time for them to emerge, trial and error
- Successful designs do exist
- exhibit recurring class and object structures

Introduction

- Patterns occur in ever facet of software development, at every phase, and at every level of detail
- The ability to recognize patterns allow us to classify problem solutions in terms of problems and contexts, providing a common vocabulary for software developers to use

What are Patterns?

- Christopher Alexander says “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Great, so what is a Pattern?

- In general, a pattern has the following attributes:
- The Name: the pattern name is a handle we can use to refer to the pattern, thus describing the problem, solution, and consequences in a word or two
- The Problem: Describes when to apply the pattern, this is broken into both a description of the problem and its context.
- The Solution: describes the elements that make up the design, their relationships, responsibilities, collaborations. Often includes some example code or pseudo-code, but doesn't describe a particular implementation
- The Consequences: describes the results and trade-offs of using the pattern. All design is compromise, and make patterns provide both benefits and drawbacks that must be balanced

The Need for Patterns

- Designing object-oriented software is hard
- Designing reusable object-oriented software is even harder
- Have to find pertinent objects, factor them into classes at the right granularity, establish relationships between them, design a flexible interface
- Design should be specific enough for the problem at hand, but general enough to address future problems (scale up or down)

Multiplicity of Design

- The problem most inexperienced designers face isn't that there aren't any solutions to a problem, it's that there are too many
- It's hard to choose the proper solution from a large set of potential solutions without experience in what works and what doesn't work for a given problem (results, consequences, and trade-offs are hard to foresee)
- Experienced designers know which solutions have worked in the past and which haven't; they have experience in the consequences and trade-offs

Purpose of Patterns

- The purpose of patterns then can be generalized as providing the following:
- Giving a vocabulary to a problem, context, solution, consequences set
- Cataloging designs that have proved successful in past systems and formalizing their elements, relationships, interactions, etc.
- Provide enough information about trade-offs and consequences to allow an intelligent design decision to be made about applying a given solution

Types of Patterns

- There are generally 3 types of patterns:
- Architectural Patterns
- Design Patterns
- Idioms

Architectural Patterns

- Architectural Patterns express a fundamental structural organization for software systems
- It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules for establishing relationships between them

Architectural Patterns Examples

- 3-tier database systems (Database, Intermediate DB Layer, User-Application)
- Client/Server
- Component (Module) -Based Software Systems
- Feedback Systems
- Event-Driven Systems

Design Patterns

- Popularized by the 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also called the *Gang of Four Book* or *GOF)*
- Their work largely based on Alexander's *The Timeless Way of Building*

How Design Patterns Solve Problems

- OOP is made up of objects, where an object packages both data and operations that may be performed on that data
- Ideally, the only way to change the data of an object is via a request to that object (normally a method call) – we call this encapsulation; it's representation is invisible to the outside world
- The hard part of object-oriented design is decomposing a system into objects. Many factors come into play: granularity, encapsulation, dependency, flexibility, performance, reusability, and each affects the nature of the decomposition, often in conflicting ways

Object-Oriented Design Paradigms

- OO Design methodologies favor many paradigms for designing object-oriented systems:
- You can single out objects and “verbs” in your system and write corresponding classes and operations
- Focus on collaborations and responsibilities in the system (have objects for different sub-systems)
- Model the “real-world” and draw object design from their real world counterparts
- Object-Oriented design also tends to foster a “meet in the middle” style of design; where you first decompose a system into high-level abstractions, usually objects, and then design reusable primitives that allow you to build those high level systems

Object Granularity

- Objects can very tremendously in size and scope, from directly interfacing with low-level hardware all the way up to the application level
- Deciding what should be an object is an important task
- Too often classes are designed poorly and, instead of encapsulating a single concept or abstraction, end up as the anti-pattern “the blob”. The blob tends to grow well beyond modeling a single concept or encapsulating a single entity, it violates the “one-object, one-responsibility” notion of OOP

The Blob

*the failure of the do-it-all interface

- An Anti-Pattern that is often used to describe classes that encompass many concepts and abstractions
- Usually a large behemoth class with little or no encapsulation
- The problem with large, behemoth classes is that they tend to widen the gap between what is *syntactically* valid and what is *semantically* valid in a system. That is, you end up with constructs that are syntactically correct, but violate the semantics of the program.
- In general, if X must be done in a system before Y, the structure of your code and design should enforce this constraint as much as possilble

Types and Interfaces

- A type is a name used to describe a particular interface (this does not necessarily have to refer to a concrete class)
- Objects declare a set of operations that it supports, each of these operations has a particular signature, that is, it's name, the objects it takes as parameters and its return type. The set of all operations that an object supports defines an interface to that object
- Nothing is said of member functions, etc. In C++ and other languages, free-functions can also be considered part of an objects interface if they define a valid operation on a given object

Inheritance vs. Composition

- There are two common techniques for facilitating object reuse in systems: class inheritance and object composition. Object composition refers to assembling (*composing*) multiple objects together to get more complex functionality
- Each has their own advantages and disadvantages, but in general, one of the tenets of object-oriented programming is to *Favor object composition over class inheritence*

Why Composition is Better ?

- Usually, inheritance is defined at compile-time, meaning you can't change implementations inherited from parent classes at run-time
- Parents classes usually define at least part of their subclasses' physical representation. For this reason, it's often said that "*inheritance breaks encapsulation*". Normally, the subclass is implemented in terms of the parent's implementation, leading to a cascade of changes should a parent class be modified
- Composition is defined at run-time by objects acquiring references to other objects. Composition requires objects to respect interfaces, which require objects to be designed so they don't stop you from using an object with many others

More on Composition

- Because objects are accessed solely through their interfaces, we do not break encapsulation, and any object may be swapped out for another so long as their type (that is, their interface) remains the same
- Composition also allows for each class to be kept isolated, small, and well encapsulated. Inheritance hierarchies will tend to remain small and will be less likely to grow into unmanageable and unmaintainable monsters

Back to Patterns

- Axiom: Patterns help to facilitate good design by defining good object granularities, interfaces, responsibilities, relationships, and abstractions
- Patterns are not solutions to new problems, they are solutions to established and studied problems that crop up again and again in object-oriented systems
- Patterns are not specific to any code, language, or implementation, they describe general solutions to problems

Categories of Design Patterns

- Design Patterns are generally broken into three groups:
- Creational Patterns deal with object creation at run-time
- Structural Patterns deal with the composition of objects and classes
- Behavioral Patterns deal with the way in which classes or objects interact and distribute responsibility

Design Pattern Catalog

Creational Patterns

Abstract Factory

Builder

Factory Method

Prototype

Singleton

Structural Patterns

Adapter

Bridge

Composite

Container

Decorator

Façade

Flyweight

Proxy

Behavioral Patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Multiple Dispatch

Observer

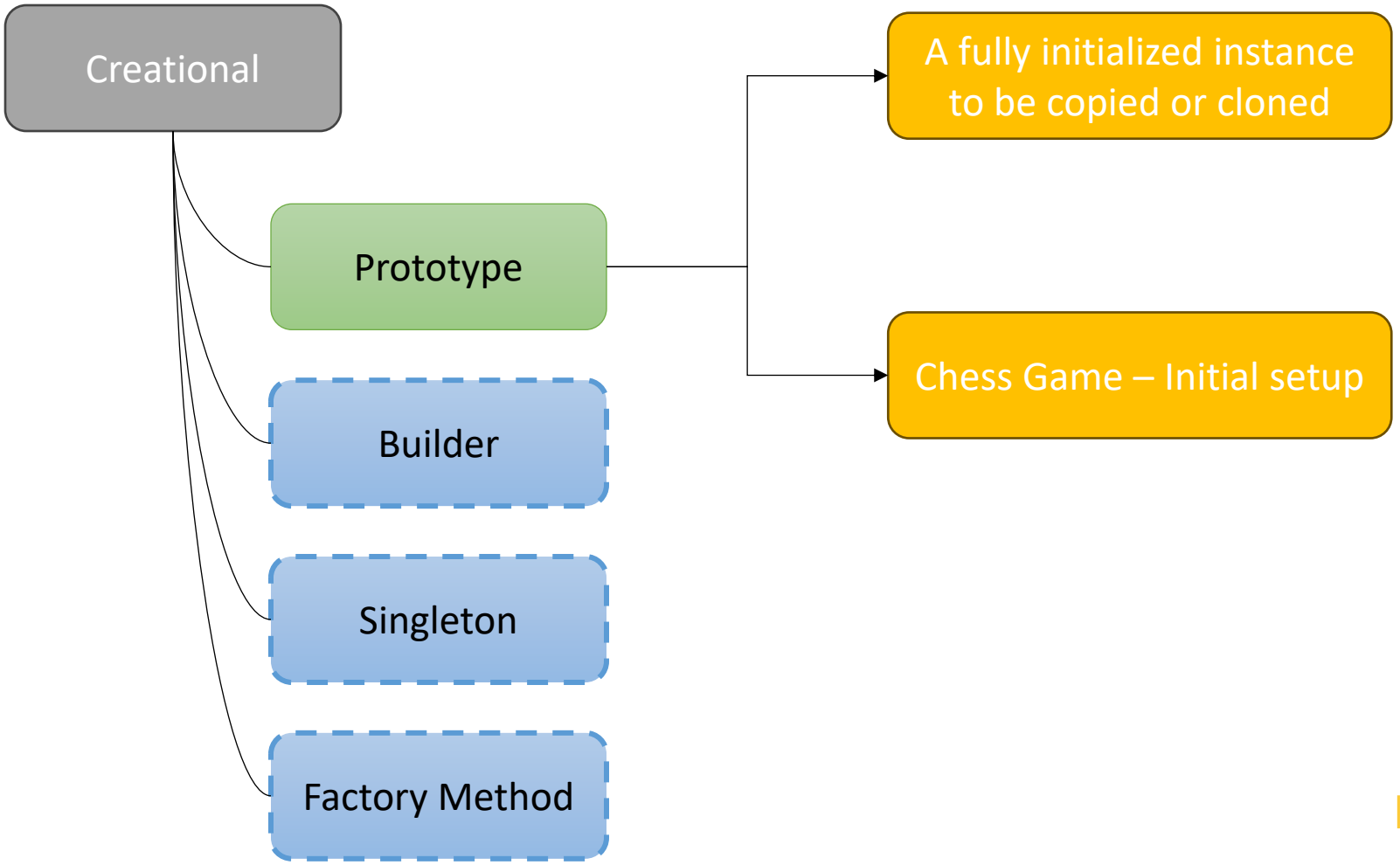
State

Strategy

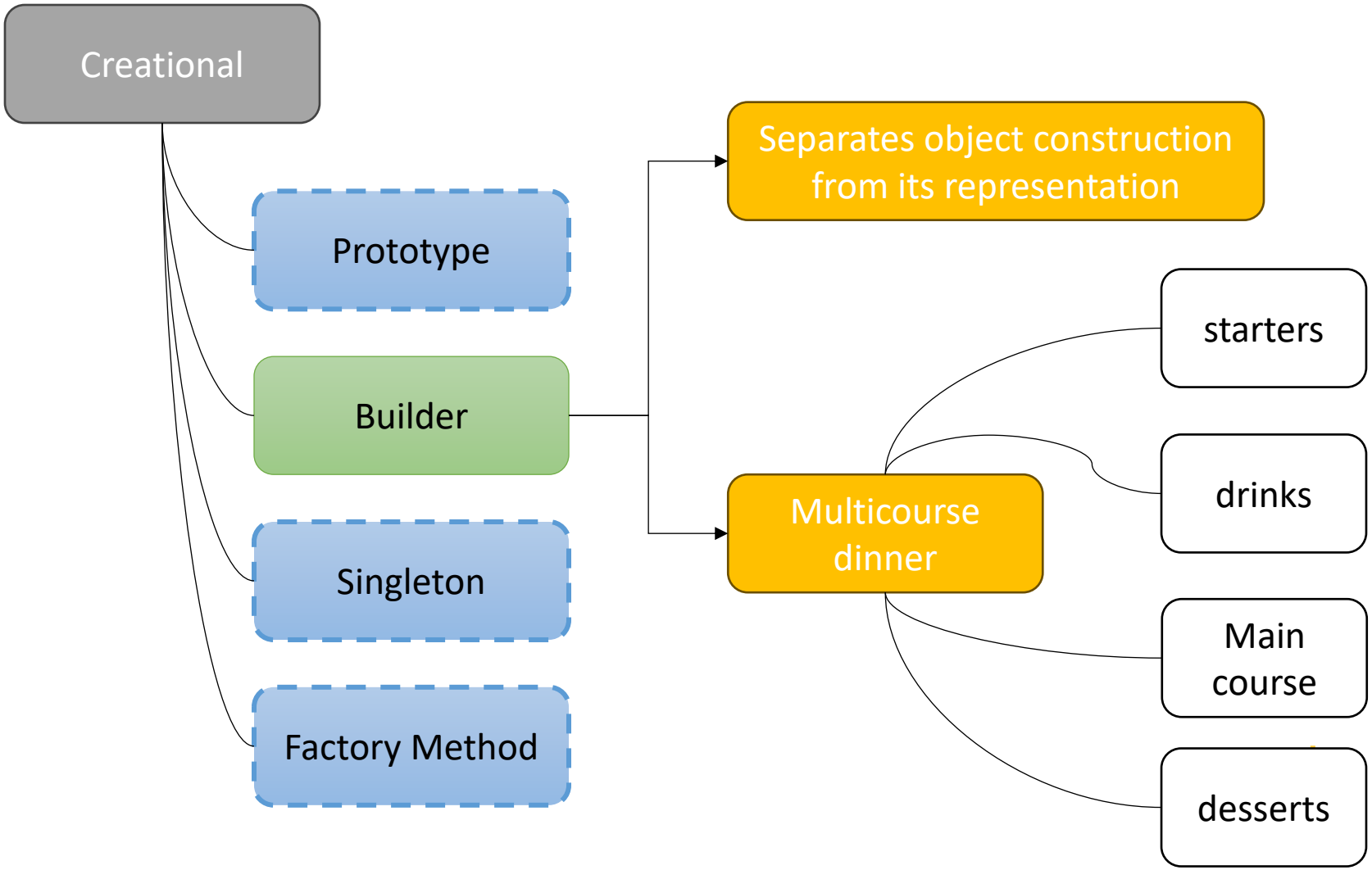
Template Method

Visitor

Creational

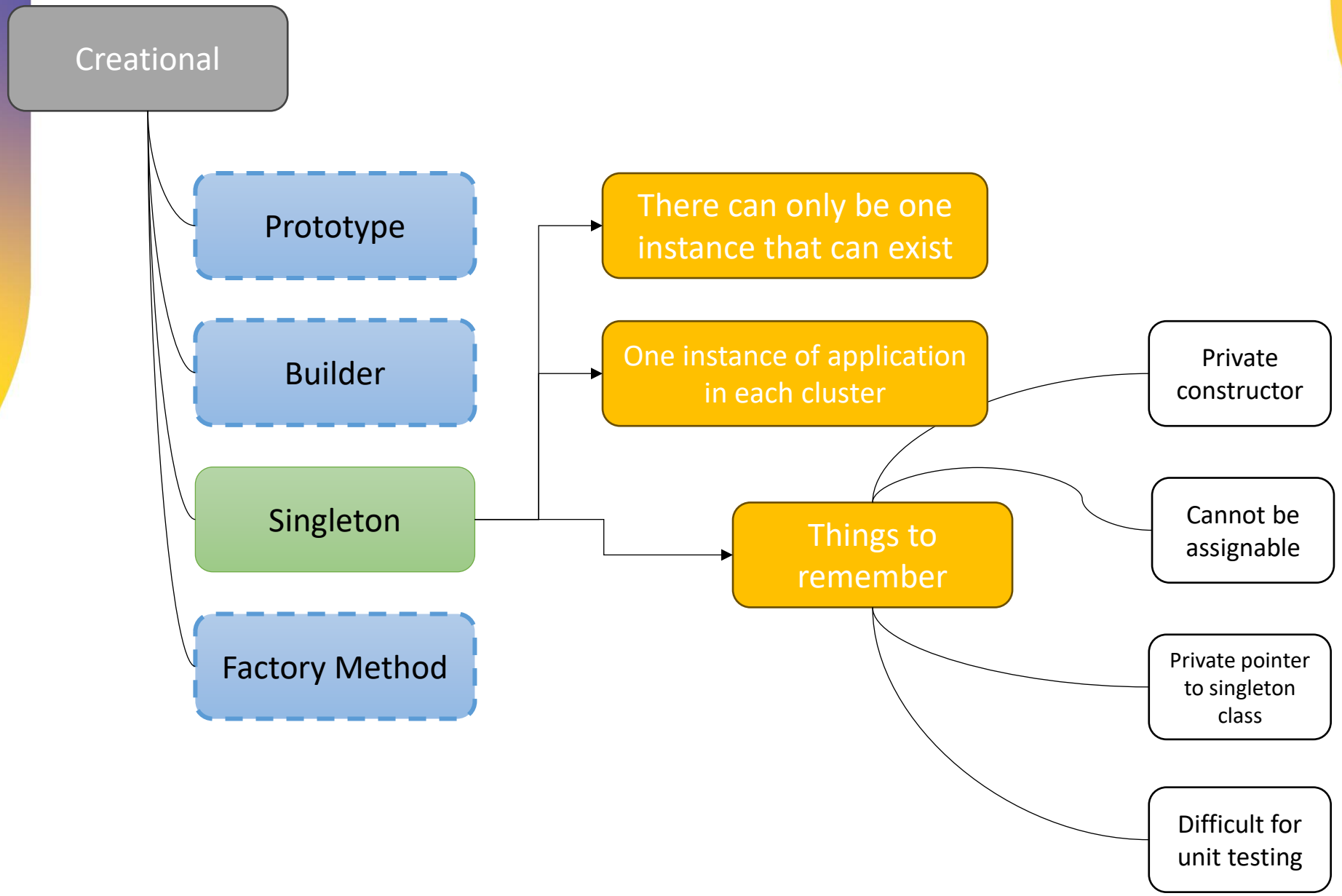

Creational
Prototype
Builder
Singleton
Factory Method
A fully initialized instance to be copied or cloned
Chess Game – Initial setup

Creational


Creational
Prototype
Builder
Singleton
Factory Method
Separates object construction from its representation
Multicourse dinner
starters
drinks
Main course
desserts

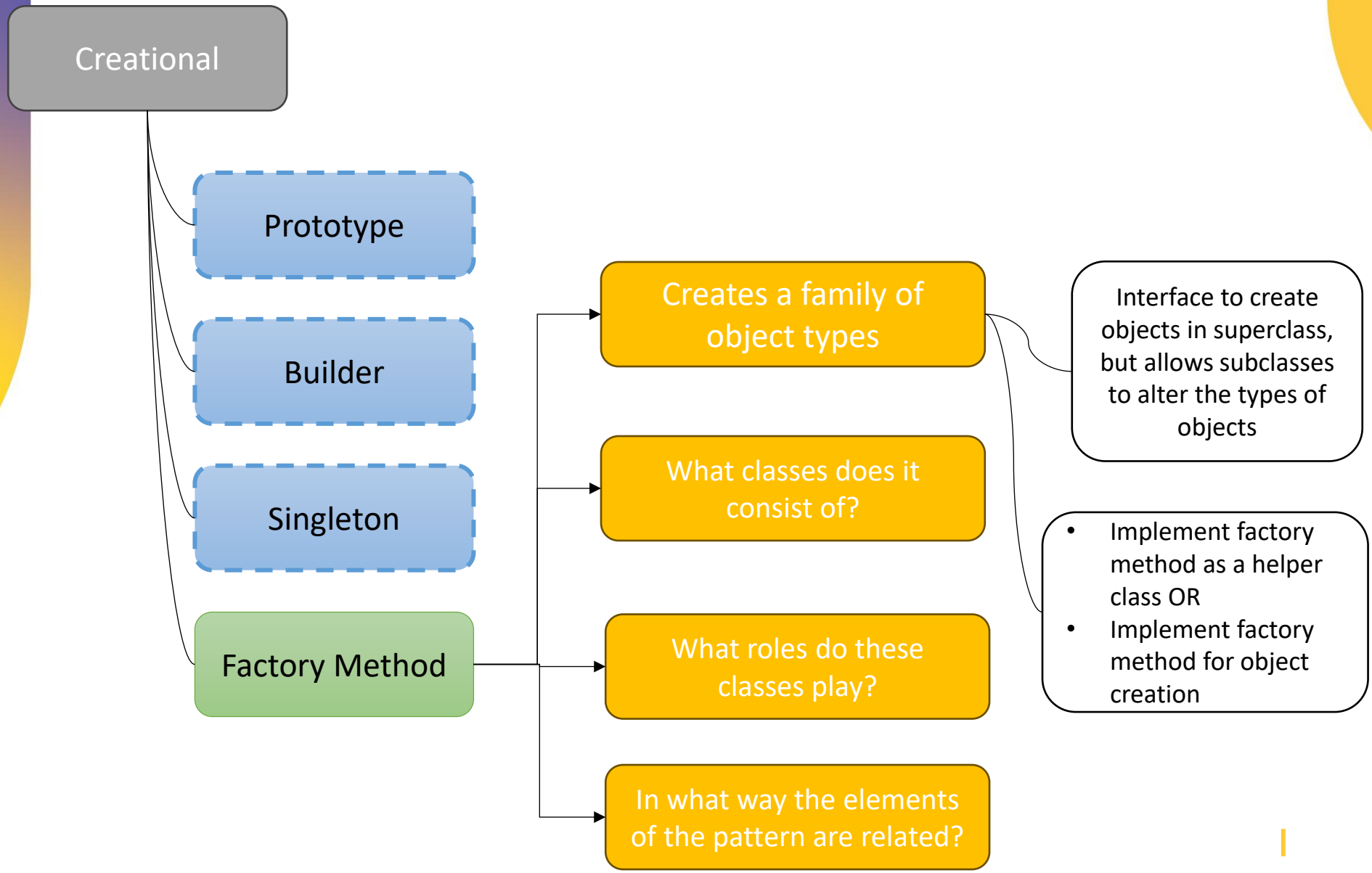
```
public class BuilderPattern {

	static class Coffee {
		private Coffee(Builder builder) {
			this.type = builder.type;
			this.sugar = builder.sugar;
			this.milk = builder.milk;
			this.size = builder.size;
		}

		private String type;    //Should be enum
		private boolean sugar;
		private boolean milk;
		private String size;    //Should be enum

		public static class Builder {
			private String type;   //Should be enum
			private boolean sugar;
			private boolean milk;
			private String size;    //Should be  enum

			public Builder(String type) {
				this.type = type;
			}

			public Builder sugar(boolean value) {
				sugar = value;
				return this;
			}

			public Builder milk(boolean value) {
				milk = value;
				return this;
			}

			public Builder size(String value) {
				size = value;
				return this;
			}

			public Coffee build() {
				return new Coffee(this);
			}
		}

		@Override
		public String toString() {
			return String
					.format("Coffee [type=%s, sugar=%s, milk=%s, size=%s]",
							type, sugar, milk, size);
		}

	}

	public static void main(String[] args) {
		Coffee coffee = new Coffee.Builder("Mocha").milk(
				true).sugar(false).size("Large").build();
		System.out.println(coffee);

		//Simplifies Creation
		//More Readable Code
		//Values cannot be modified
	}
}
```

Creational


Creational
Prototype
Builder
Singleton
Factory Method
There can only be one instance that can exist
One instance of application in each cluster
Things to remember
Private constructor
Cannot be assignable
Private pointer to singleton class
Difficult for unit testing

```cpp
#include <iostream>
#include <mutex>
using namespace std;

class Singleton {
private:
    // Member variables
    string name, loves;

    // Static pointer to the Singleton instance
    static Singleton* instancePtr;

    // Mutex to ensure thread safety
    static mutex mtx;

    // Private Constructor
    Singleton() {}

public:
    // Deleting the copy constructor to prevent copies
    Singleton(const Singleton& obj) = delete;

    // Static method to get the Singleton instance
    static Singleton* getInstance() {
        if (instancePtr == nullptr) {
            lock_guard<mutex> lock(mtx);
            if (instancePtr == nullptr) {
                instancePtr = new Singleton();
            }
        }
        return instancePtr;
    }
    // Method to set values
    void setValues(const string& name, const string& loves) {
        this->name = name;
        this->loves = loves;
    }
    // Method to print values
    void print() const {
        cout << name << " Loves " << loves << "." << endl;
    }
};

// Initialize static members
Singleton* Singleton::instancePtr = nullptr;
mutex Singleton::mtx;

// Driver code
int main() {
    Singleton* s1 = Singleton::getInstance();
    s1->setValues("StudentA", "GamesENGII");
    s1->print();

    Singleton* s2 = Singleton::getInstance();
    s2->setValues("StudentB", "GamesENGII");
    s2->print();

    return 0;
}
```

Creational


Creational
Prototype
Builder
Singleton
Factory Method
Creates a family of object types
What classes does it consist of?
What roles do these classes play?
In what way the elements of the pattern are related?
Interface to create objects in superclass, but allows subclasses to alter the types of objects
• Implement factory method as a helper class OR
• Implement factory method for object creation

Factory Method – Implementation

Suppose you have a Battleship superclass that extends into different types of subclasses like Destroyer, Carrier… etc.

Let’s first implement in a conventional method that just inherits parent class

Next, let’s use factory method implementation.

```cpp
class Battleship {
public:
    Battleship() {
        std::cout << "Battleship Created" << std::endl;
    }
    virtual void Fire() = 0;
    virtual void Steer() = 0;
};

class Destroyer : public Battleship {
public:
    Destroyer(){
        std::cout << "Destroyer Created" << std::endl;
    }
    void Fire() override {
        std::cout << "Destoryer Fire" << std::endl;
    }
    void Steer() override {
        std::cout << "Destroyer Steer" << std::endl;
    }
};

class Carrier : public Battleship {
public:
    Carrier(){
        std::cout << "Carrier Created" << std::endl;
    }
    void Fire() override {
        std::cout << "Carrier Fire" << std::endl;
    }
    void Steer() override {
        std::cout << "Carrier Steer" << std::endl;
    }
};
```

```cpp
int main() {
    Destroyer d;
    Carrier c;
    d.Fire();
    c.Fire();
    return 0;
}
```

Factory Method

```
enum class ShipType{
    Destroyer,
    Carrier,
};

Battleship* CreateShipFactory(ShipType type){
    Battleship* pShip = nullptr;
    if(type == ShipType::Carrier) {
        pShip = new Carrier();
    }
    else if(type == ShipType::Destroyer) {
        pShip = new Destroyer();
    }
    return pShip;
}
```

```
int main() {
    Battleship* d = CreateShipFactory(ShipType::Destroyer);
    Battleship* c = CreateShipFactory(ShipType::Carrier);
    d->Fire();
    c->Fire();
    return 0;
}
```

Patterns Explored

- We'll explore a small subset of the given patterns in detail, going over the concept, the problem/context pair, solution, and a little about implementation
- Attempt to focus on patterns that aren't obvious or common sense
- Patterns to explore
- Factory Method
- Composite
- Iterator
- Visitor

Factory Method

- **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
- **Problem**
- Frameworks use abstract classes to define and maintain relationships between objects. A framework is also often responsible for creating these objects
- Sometimes it is impossible at compile-time to determine which objects need instantiated. The framework knows when an object is needed, but not what kind
- **Applicability**
- You have a certain family of objects that are all inter-related, but each entity in the family performs a different task
- You cannot determine at compile-time which object to instantiate, and the object that's to be created may change at run-time

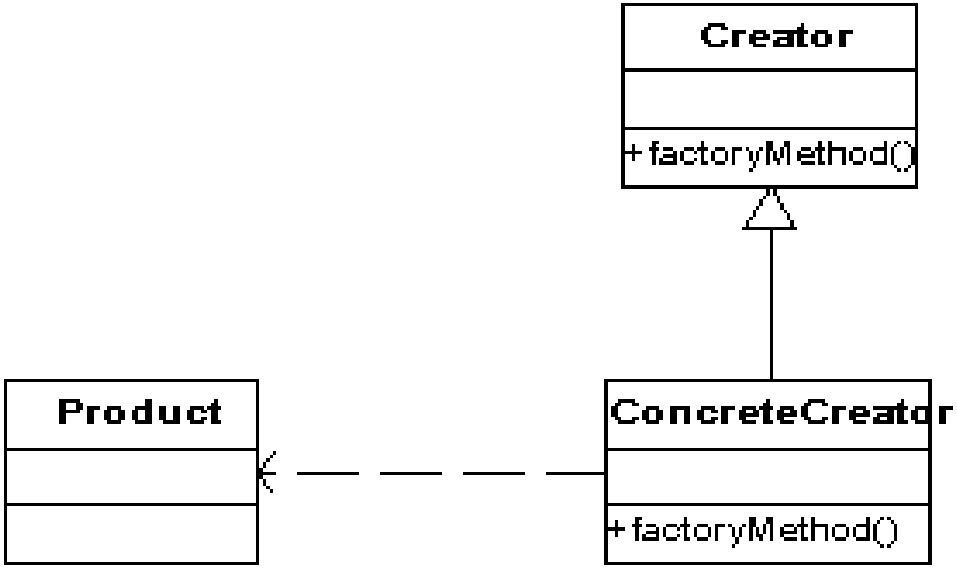
Motivating Example

- You're designing an application that supports opening of many different types of documents, for the sake of an example, let's say a program like Photoshop, so you can open .png, .jpg, .gif, etc.
- At some level of granularity, you need to create different objects to handle these different types of documents, it could be at a high-level, defining a different Document class for each type of file, or at a low-level, defining a different DocReader class for each file
- You don't know at compile-time what kinds of files are going to be open, and what's more, the types of files supported by the system might expand in the future

Factory Method

- Participants
- **Product**
- Defines the interface of objects the factory method creates
- **ConcreteProduct**
- Implements the product interface, defines a concrete type
- **Creator**
- Declares the factory method, which returns an object of type Product
- **ConcreteCreator**
- Provides a concrete factory method that returns an instance of a ConcreteProduct

Factory Method



Factory Method – Implementation

```
template <class Key,
          class Product,
          class Creator = Product* (*)() >
class Factory {
public:
    void Register(const Key& id, Creator creator);
    Product Create(const Key& id);
private:
    std::map<Key, Creator> creators_;
};
```

Implementation

- The Factory classes used templates to achieve generality. In the case of our example, let's see how it would be used:

```
struct DocReaderCreator {
    DocumentReader* operator()();
};

class Factory<string, DocumentReader, DocReaderCreator*> factory;

struct JPGReaderCreator : public DocReaderCreator {
    JPGReader* operator ()() { return new JPGReader(); }
};
```

Implementation

```
factory.Register("JPG", new JPGReaderCreator);

// Later:
DocReader* reader = factory.Create(reader_type);
```

- When a reader is created, there are no details visible about the type of reader created other than an identifier that signifies the object type, which in our case is just a string. This allows new products to easily be added to the factory, and avoids using a giant switch to determine what kind of object to create

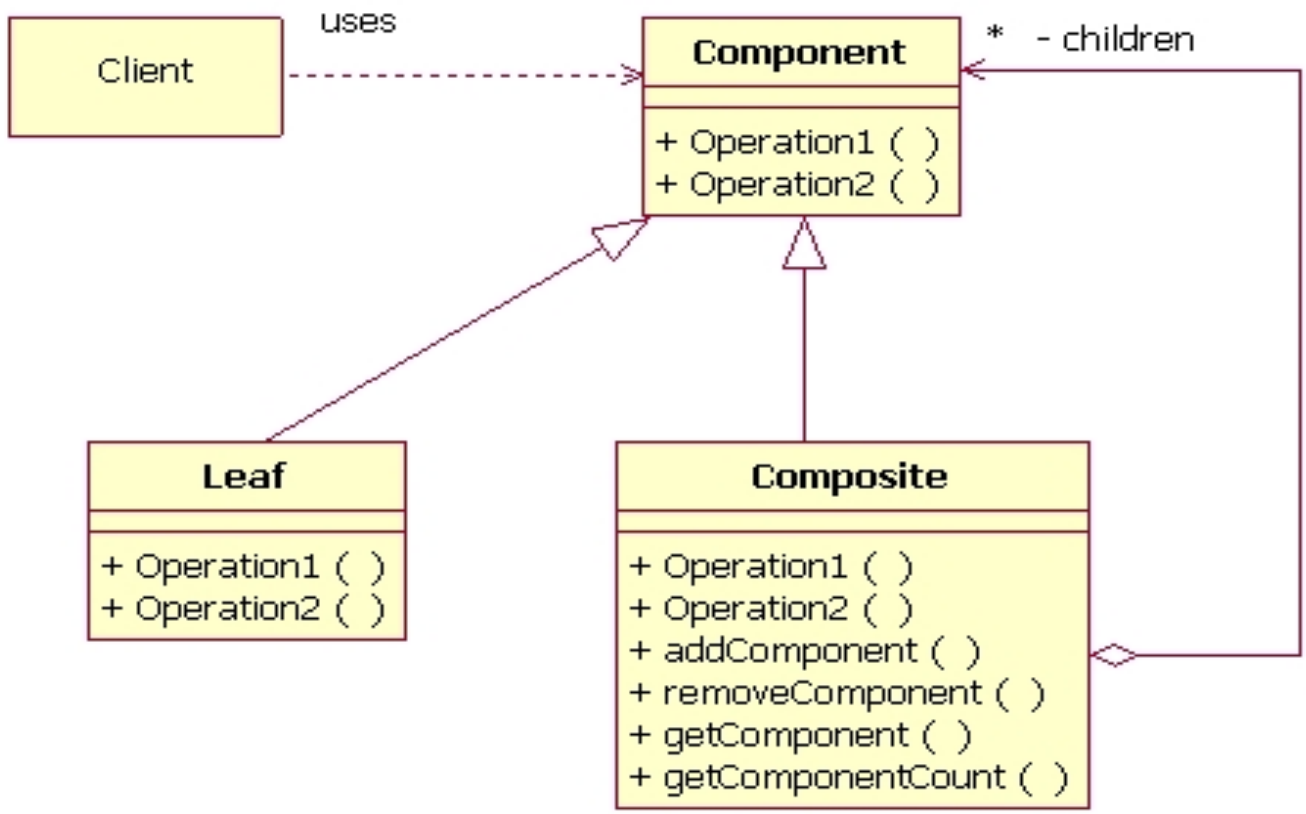
Composite

- **Intent**
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- **Problem**
- You want to build more complex components out of simpler components, combining them together at run-time to create complex objects
- However, you don't want to treat the objects and their containers differently; manipulation should be uniform across all related objects
- **Applicability**
- Use when you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

Composite

- **Participants**
- **Component**
- Declares the interface for objects in the composition
- Implements default behavior for the interface common to all classes, as appropriate
- **Leaf**
- Represents leaf objects in the composition
- Defines primitives in the system
- **Composite**
- Defines behavior for components having children
- Stores children components

Composite


Client
uses
Component
+ Operation1 ( )
+ Operation2 ( )
* - children
Leaf
+ Operation1 ( )
+ Operation2 ( )
Composite
+ Operation1 ( )
+ Operation2 ( )
+ addComponent ( )
+ removeComponent ( )
+ getComponent ( )
+ getComponentCount ( )

Composite

- Consequences
- Defines class hierarchies consisting of primitives and compositions uniformly; wherever a client expects a primitive object, it can also take a composition
- Makes the client simple, as the client can treat composite structures and single objects uniformly
- Makes it easy to add new kinds of components
- Disadvantage is that it can make your design overly general; it’s harder to restrict the components of a composite, often have to rely on run-time rather than compile-time checks to enforce contraints

Motivating Example

- Compilers are used to generate abstract syntax trees of languages so analysis can be performed and code generated
- But, part of an AST should be treated uniformly, consider expressions:
- Each expression can be made up of multiple expressions, but we should be able to treat all expressions uniformly
- Expressions could be made up of algebraic operators, variable accesses, etc
- We'd like to treat all expressions in a uniform manner for when we later need manipulate the syntax tree

Implementation

```
struct Expression {
    virtual int evaluate() = 0;
};

class Plus : public Expression {
public:
    Plus(Expression* lhs, Expression* rhs);
    int evaluate() { return lhs.evaluate() + rhs.evaluate(); }
};
```

Implementation

```
class Identifier {
public:
    int evaluate() { return val_lookup(id); }
private:
    string id;
};

struct IntLiteral {
public:
    IntLiteral(int n);
    int evaluate() { return val; }
};
```

Implementation

```
// Usage, for expression 10 + 5 + a
Expression* e = new Plus(
    new IntLiteral(10),
    new Plus(
        new IntLiteral(5),
        new Identifier("a")
    )
);
```

Iterator

- **Intent**
- Provide a means to access the elements of an aggregate type without exposing its underlying implementation
- **Problem**
- An aggregate object such as a list should give a means to iterate across all elements in the list without exposing its internal structure
- But, you don’t want to clutter the list interface with messy details about traversals, particularly if it precludes you from having multiple traversals occurring at once
- You might also want to be able to swap the list container out with a different container later on, but keep the code that uses the container the same

Iterator

- **Applicability**
  - Use an iterator to access an aggregate object's contents without exposing its underlying implementation
  - To support multiple traversals of a aggregate objects
  - To decouple an aggregate object from the algorithms that act upon that object

- **The last point is significant, and provides the motivation for our example**

Iterator

- Participants
- **Iterator**
- defines an interface for accessing and traversing elements
- **ConcreteIterator**
- Particular iterator implementation
- **Aggregate**
- defines an interface for creating an iterator object
- **ConcreteAggregate**
- creates concrete iterators

Motivating Example

- We'd like to design a set of containers, iterators, and algorithms that can all work together to provide a library of data structures
- The algorithms should, whenever possible, be decoupled from the containers they operate on. This allows us to define new algorithms that work with existing containers, and new containers that work with existing algorithms
- Fortunately for us, if we're using C++, this problem has already been solved

C++ Iterators

- **The concept of iterators is very important to the C++ Standard Library. They provide an abstraction between the representation of data stored in containers, and the algorithms that act upon that data**
- **Sample:**

Say we have a function, print, that we want to use to print the contents of a container. We'd like to generalize this algorithm as much as possible, hopefully so we can use it for multiple containers.

```
template <class FwdIterator>
void print(FwdIterator begin, FwdIterator end)
{
    while (begin != end)
    {
        std::cout << *begin << std::endl;
        ++begin;
    }
}
```

C++ Iterators

- Again, we use C++ templates to generalize the function. Our type, FwdIterator, is an iterator concept, that is, it provides the operations increment (++), and dereference (*). In this way, Iterators form a subset of pointer syntax and semantics. The type of iterator passed to print is inconsequential, so long as it supports the iterator concept:

```
std::vector<int> v;       // assume populated
print(v.begin(), v.end());

int array[20];            // assume populated
print(array, array + 20);

std::list<int> l;         // assume populated
print(l.begin(), l.end());
```

Visitor

- **Intent**
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
- **Motivation**
- Sub-classing allows us to easily add new class types, and these new classes can define different behavior for a family of objects
- However, it is relatively hard to add new operations to a given set of classes; the new operation must be added to each class in the hierarchy. Distributing all of these new operations across all the classes also makes the classes harder to understand and is frequently undersirable

Visitor

- **Applicability**

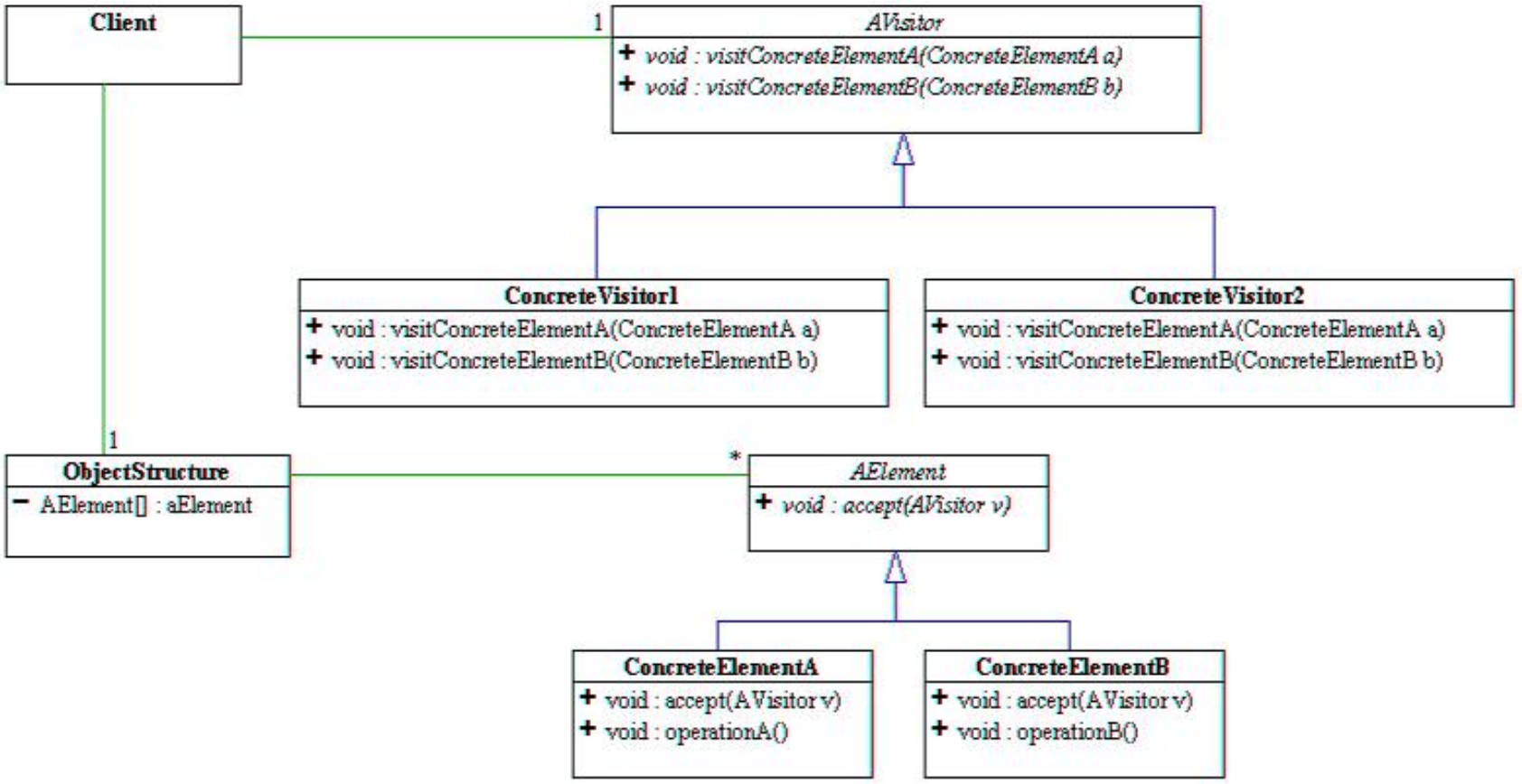
Use visitor when:

- An object structure contains many classes of objects with differing interfaces and you want to perform operations on these objects depending on their concrete class types
- Many distinct, unrelated operations need to be performed on objects and you want to avoid polluting the classes with these operations. Visitor lets you keep related operations together
- The class hierarchy is relatively stable, that is, classes are rarely added or removed from the existing class hierarchy, but you often want to add new operations, Changing the hierarchy requires changing the interfaces to all the visitors

Visitor

- **Participants**
- **Visitor**
- Declares a Visit method for each class in the object hierarchy that you want to be visitable
- **ConcreteVisitor**
- Defines each Visit method for a particular operation; a ConcreteVisitor defines a single operation to be performed on the class hierarchy
- **Element**
- Declares an Accept method that takes a visitor as an argument
- **ConcreateElement**
- Implements the Accept method
- **Object Structure**
- Can enumerate it’s elements, may provide a way to apply a visitor over a range of elements, such as over a list

Visitor


Client
1
AVisitor
+ void : visitConcreteElementA(ConcreteElementA a)
+ void : visitConcreteElementB(ConcreteElementB b)
ConcreteVisitor1
+ void : visitConcreteElementA(ConcreteElementA a)
+ void : visitConcreteElementB(ConcreteElementB b)
ConcreteVisitor2
+ void : visitConcreteElementA(ConcreteElementA a)
+ void : visitConcreteElementB(ConcreteElementB b)
1
ObjectStructure
- AElement[] : aElement
*
AElement
+ void : accept(AVisitor v)
ConcreteElementA
+ void : accept(AVisitor v)
+ void : operationA()
ConcreteElementB
+ void : accept(AVisitor v)
+ void : operationB()

Motivating Example

- Back to our example about a compiler and building abstract-syntax trees
- In general, the class hierarchy of an abstract syntax tree doesn't change much once it has been defined, unless a language change is made, in which case, a lot of things need to be rewritten anyway
- However, we frequently want to perform many different, unrelated operations on a syntax tree, such as type checking, program restructuring, code generation, interpreting, etc

Implementation

```
abstract public class ASTNode {
    public void Accept(Visitor vis);
}

public class PlusNode {
    public void Accept(Visitor vis) {
        vis.Visit(lhs);
        vis.Visit(rhs);
        vis.Visit(this);
    }
}
```

Implementation

```
public class Visitor {
    public void Visit(PlusNode plus);
    public void Visit(IntLiteral intLiteral);
    public void Visit(IdNode id);
}

public class Interpreter {
    public void Visit(PlusNode plus) {
        stack.push(stack.pop() + stack.pop());
    }
    public void Visit(IntLiteral i) {
        stack.push(i.value());
    }
}
```

Visitor

- The primary disadvantage of visitor is that it's difficult to add new classes in the future, you need to be sure your class hierarchy is relatively stable and that a change to it represents a major change to the system
- The visitor pattern roughly implements a notion known as double dispatch; that is, instead of dynamically dispatching on a single type, such as in Java or C++ (the object type determines the function that is called), double dispatch uses two types to determine which function to call
- This is NOT function overloading; function overloading takes place entirely at compile-time, double dispatch takes place at run-time

Discussion

- Patterns do not exist as isolated elements of a system, frequently patterns work together to achieve design goals. We've already seen how composite can be used in conjunction with Visitor for class hierarchies.
- The Command pattern, which encapsulates a request in the system as an argument for delayed evaluation, can frequently be used with the Factory Method pattern; Commands are created at run-time by a Factory depending on user action

Idioms

- Idioms are lower level patterns than general design patterns; they are often specific to a programming language, and usually only encompass a few lines of code
- Idioms however are important to understand, and it’s often crucial to pick up the particular idioms of a language to use it properly

ScopeGuard

- **Intent**
- Scope guard is a pattern that occurs in languages with deterministic destruction designed to facilitate an undo operation, particularly in the case of exceptions being thrown
- **Motivation**
- Writing exception safe code is hard, particularly when you want to enforce certain exception guarantees
- The usual exception handling techniques, namely try…catch doesn't scale particularly well, and clutters code with additional control flow statements

ScopeGuard

- Example:
- Consider the following function

```
void CompoundOperation()
{
    ComplexOperation1();
    ComplexOperation2();
}
```

- **Assume that both ComplexOperation1 and ComplexOperation2 might throw an exception. Also assume that we want this function to give the strong exception guarantee; that is, either all of its operations succeed, or any changes made are rolled back in the face of failure**

ScopeGuard

- First approach might be this:

```
void CompoundOperation()
{
    ComplexOperation1();
    try
    {
        ComplexOperation2();
    }
    catch (exception e)
    {
        UndoOp1();
        throw;
    }
}
```

- This works, but now our 2 line function has become 6 or 7 lines, what's more, the control flow is interrupted by try…catch blocks that don't add any semantic meaning to the normal control flow, they're only there in case of errors

ScopeGuard

- What's more, this solution doesn't scale well. Consider what happens if we have 3 operations instead of 2; now we're faced with the prospect of nested try…catch blocks, all to handle an exception that should rarely occur; the resulting function spends more code dealing with error handling than actual program semantics
- We need a way to alleviate the burden of writing all the try….catch blocks that still allows us to write exception safe code, in this case, undoing the first operation should the second fail

ScopeGuard

- **ScopeGuard's Task**
- Encapsulate an undo request in an object such that, should an exception be thrown, the guard will perform the undo operation

```
void CompoundOperation()
{
    ComplexOperation1();
    ScopeGuard guard(&UndoOp1);
    ComplexOperation2();
    guard.dismiss();
}
```

- This seems much nicer than our previous solution. We've added 2 lines of code, yes, but they are less intrusive than the try catch blocks. What's more, this solution scales well, if we have 3 operations, we can simply add another guard that performs the undo of the second operation.

Design Patterns

Questions?