Mobile Architecture & Security

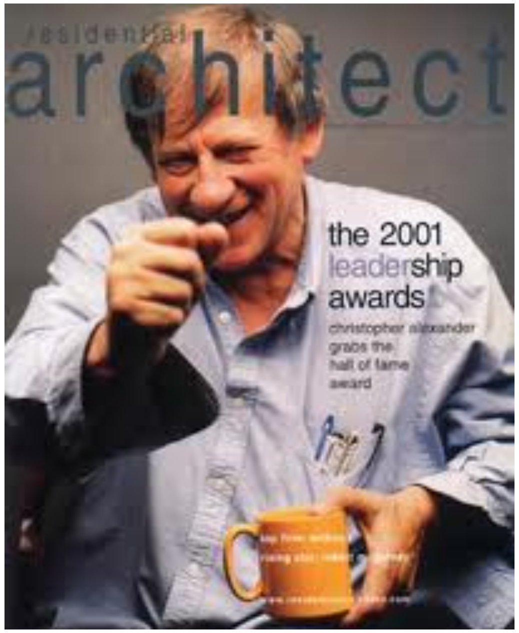
Design Patterns

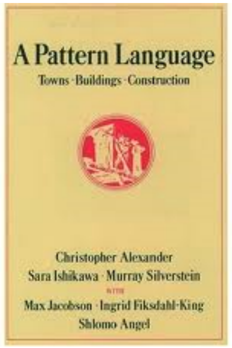
Mikhail.Timofeev@ncirl.ie

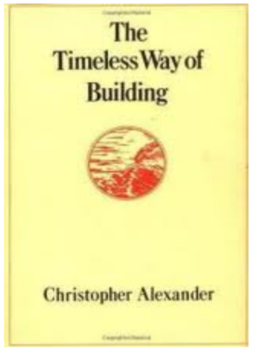
Overview

3.1 Introduction

- **Christopher Alexander**
  - Was a *real* architect
  - Originated the idea of patterns in relation to the design of buildings and towns in the late 1970's
  - *'Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.'*
    - The context – under what conditions can a pattern be applied
    - The problem – a 'system of forces'
    - The solution – what configuration can be put in place that balances the 'system of forces'

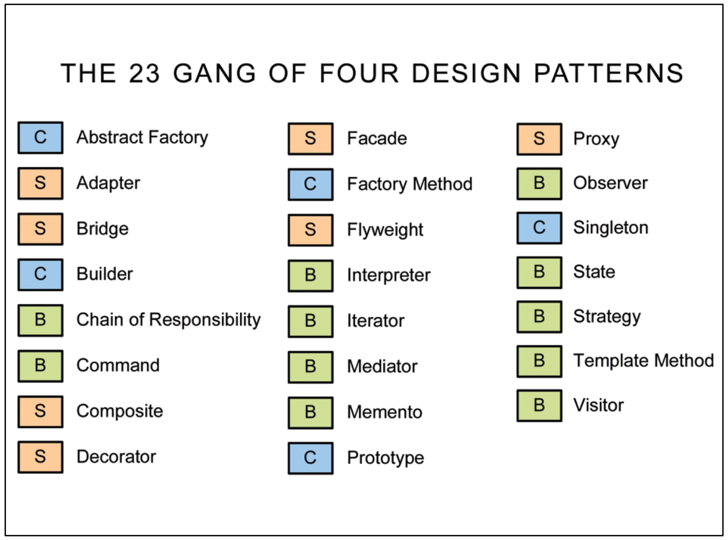



A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
with
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel



3.2 What are Design Patterns?

- A **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 (Christopher Alexander, 1977)



THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

- A **design pattern** is a general repeatable solution to a commonly occurring problem in software design (GoF, 1994)

3.2 What are Design Patterns?

- **Benefits**
  - Speed up the development process by proven solutions
  - End up having standard & maintainable codebase
  - Robust system design than ad-hoc system design
  - It's easy for developers to communicate since patterns are well defined
  - Correct use of OOP concepts
  - Well-tested & well-documented solutions for common problems
  - Easy to accommodate change
  - Nothing to innovative, Just reuse

3.2 What are Design Patterns?

- **Note**
  - Design patterns should never be interpreted dogmatically
  - Are not Superman and will never magically pop up to save a project in trouble
  - Are neither the dark nor the light side of the Force. They might be with you, but they won't provide you with any special extra power
- **Remember**
  - DPs are just helpful
  - You don't choose it, it emerges out of your factoring steps
  - The wrong way to deal with DPs is by going through a list of patterns and matching them to the problem. It works the other way around.

3.3 Elements of Pattern Design

- **The pattern name**
  - It is used to describe a design problem, its solutions and consequences in a word or two
- **The problem**
  - It describes when to apply the pattern
- **The solution**
  - It describes the elements that make up the design, their relationships, responsibilities and collaborations
- **The consequences**
  - The results an tradeoffs of applying the pattern

3.4 Object-Oriented Design

- The entire gist of **OOD** is contained in this sentence:
  – You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them
- Separate what varies from what stays the same
- Favor composition over inheritance
- Always program to interfaces not implementations
- Strive for loosely coupled design between objects
- Classes should be open to extension but closed for modification (open-close principle)

3.5 Classification of Design Patterns

- Purpose – ‘What the pattern does’
  - Creational Patterns
    - Concern the process of object creation
  - Structural Patterns
    - Deals with the composition of classes and objects
  - Behavioural Patterns
    - Deals with the interaction of classes and objects
- Scope – ‘What the pattern applies to’
  - Class Patterns
    - Focuses on the relationships between classes and their subclasses
    - Involves reuse through inheritance
  - Object Patterns
    - Focuses on the relationships between objects
    - Involves reuse through composition

3.5 Classification of Design Patterns

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

3.6 Structural Patterns

- **Introduction**
  - Structural Patterns are concerned with how classes and objects come together to form larger structures.
    - Examples:
      - How can we add or remove functionality from existing objects in a dynamic manner?
      - How can we control access to an object?
      - How can we simplify interfacing with a complex subsystem?
      - How can we treat single objects and composite objects in a like manner?

- **Structural Patterns**

Decorator	**Proxy**	**Composite**	**Bridge**
Flyweight	**Adapter**	**Facade**	

3.6 Structural Patterns

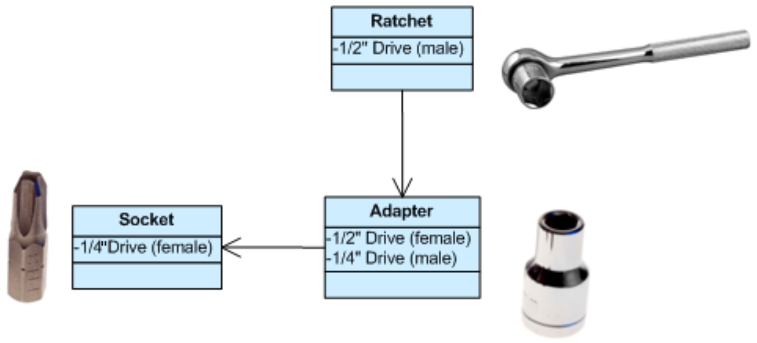
- **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- **Facade** provides a simplified interface to a large body of code.
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

3.6 Structural Patterns: Adapter

- **Intent**
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system
- **Problem**
  - An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.
  - It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary

3.6 Structural Patterns: Adapter

- **Example**
  - The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.
  - Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same.
  - Typical drive sizes in the United States are 1/2" and 1/4".
  - Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.

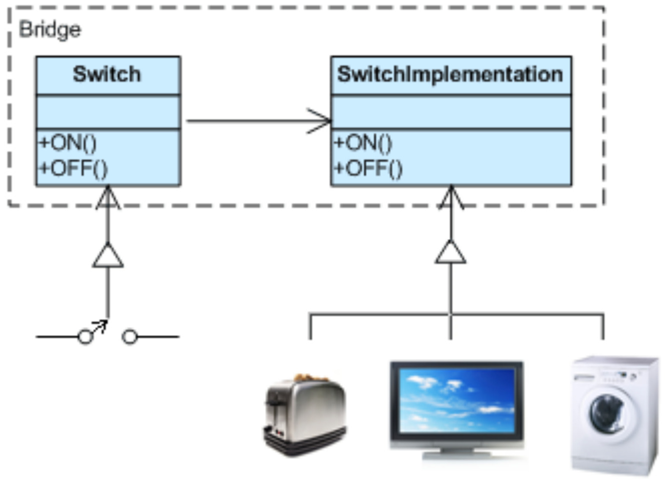


3.6 Structural Patterns: Bridge

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
  - Beyond encapsulation, to insulation
- **Problem**
  - "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

3.6 Structural Patterns: Bridge

- **Example**
  - The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.
  - A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off.
  - The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

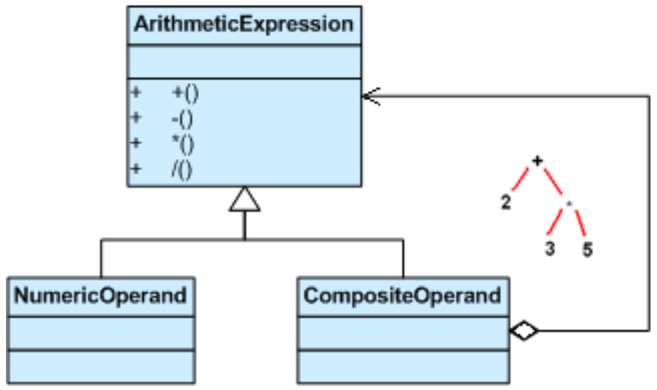


3.6 Structural Patterns: Composite

- **Intent**
  - Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - Recursive composition
  - "Directories contain entries, each of which could be a directory."
  - 1-to-many "has a" up the "is a" hierarchy
- **Problem**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

3.6 Structural Patterns: Composite

- **Example**
  - The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly.
  - Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand.
  - The operand can be a number, or another arithmetic expression.
  - Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.

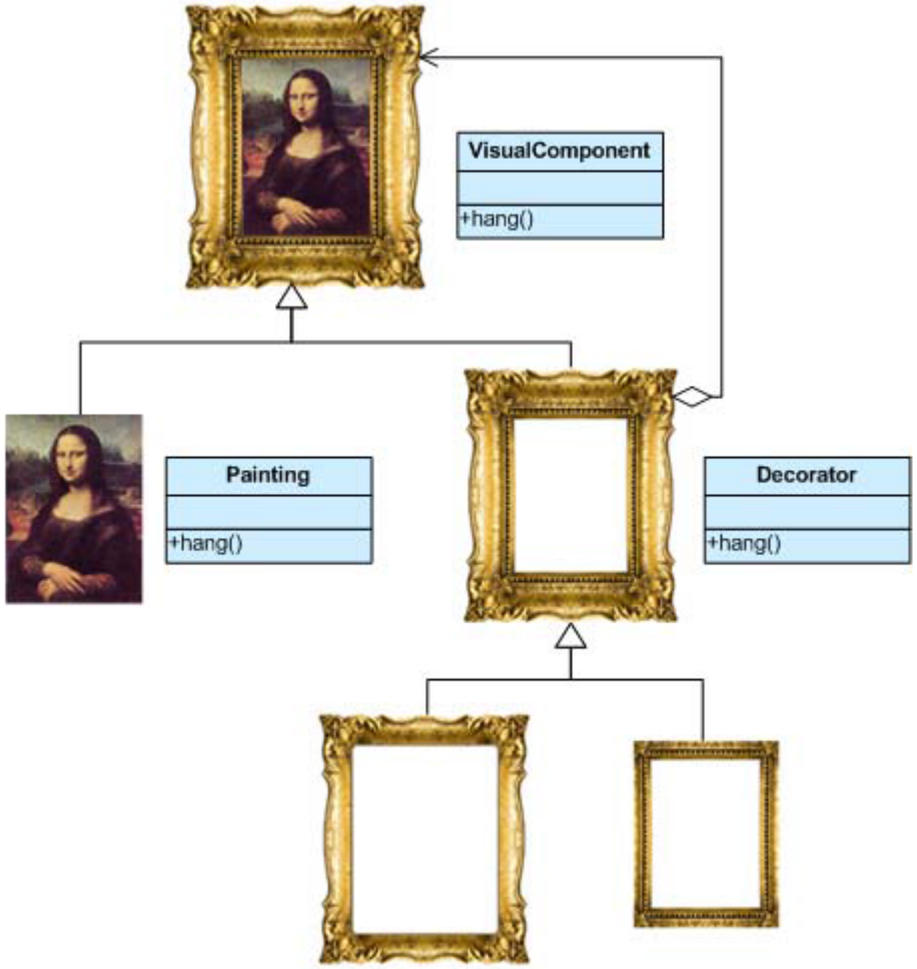


3.6 Structural Patterns: Decorator

- **Intent**
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client-specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.
- **Problem**
  - You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

3.6 Structural Patterns: Decorator

- **Example**
  - The Decorator attaches additional responsibilities to an object dynamically.
  - The ornaments that are added to pine or fir trees are examples of Decorators.
  - Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look.
  - The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used.
  - As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.
  - Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall.
  - Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.

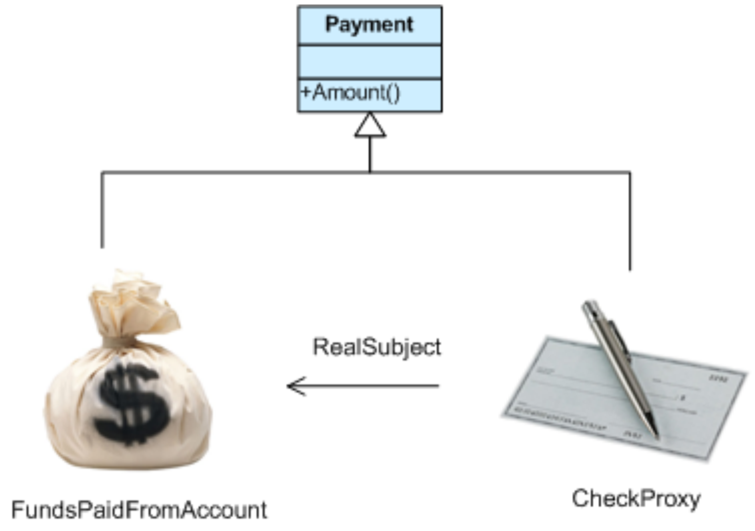


3.6 Structural Patterns: Proxy

- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.
- **Problem**
  - You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

3.6 Structural Patterns: Proxy

- **Example**
  - The Proxy provides a surrogate or place holder to provide access to an object.
  - A check or bank draft is a proxy for funds in an account.
  - A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



3.6 Structural Patterns: Proxy

- **Types of Proxy**
  - Virtual Proxies – hands the creation of an object over to another object. This can be useful if the creation process might be slow or might prove unnecessary
  - Authentication Proxies – checks that the access permissions for a request are OK
  - Remote Proxies – encodes requests and sends them across a network
  - Smart Proxies – add to or change requests before sending them on / enforce locking on real object prior to accessing request operation

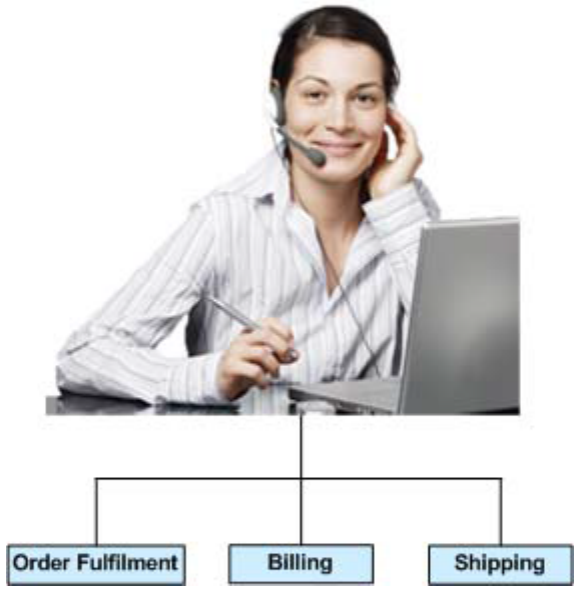
- **Uses**
  - Act as frontend to classes that have sensitive data or slow operations
  - Used in graphical applications – proxies can act as a placeholder for image objects
  - Can be used to initiate buffering of data streams (e.g., video)

3.6 Structural Patterns: Façade

- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem.
  - Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.
- **Problem**
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

3.6 Structural Patterns: Façade

- **Example**
  - The Facade defines a unified, higher level interface to a subsystem that makes it easier to use.
  - Consumers encounter a Facade when ordering from a catalog.
  - The consumer calls one number and speaks with a customer service representative.
  - The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



3.7 Creational Patterns

- **Introduction**
  - Creational Patterns are concerned with how objects are composed, created, and represented. These patterns promote loose-coupling between the constituent parts of a system. They encapsulate knowledge about the classes that are used within a system but hide the details of how objects are created and put together.
    - Examples:
      - How can we create objects without knowing the exact subclasses being used?
      - How can we restrict the number of instances of a class that can be created?

- **Creational Patterns**

Abstract Factory	**Prototype**	**Singleton**
Factory Method	**Builder**	

3.7 Creational Patterns

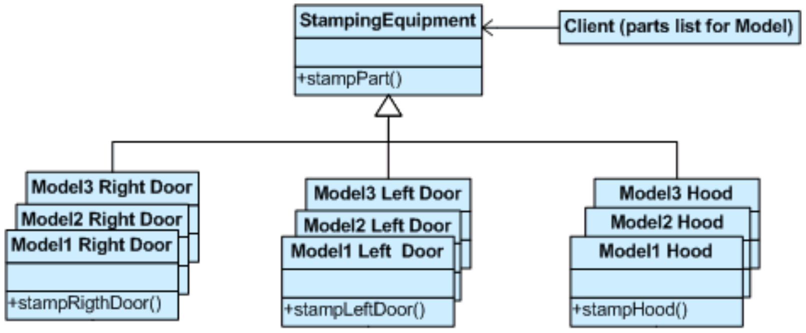
- **Abstract Factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory Method** creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.

3.7 Creational Patterns: Abstract Factory

- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products"
- **Problem**
  - If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of *#ifdef* case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code

3.7 Creational Patterns: Abstract Factory

- **Example**
  - The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.
  - This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles.
  - The stamping equipment is an Abstract Factory which creates auto body parts.
  - The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars.
  - Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.

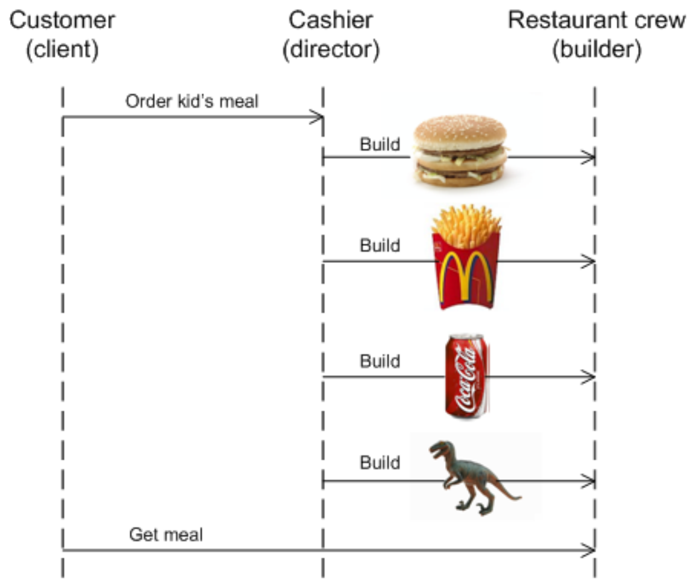


3.7 Creational Patterns: Builder

- **Intent**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Parse a complex representation, create one of several targets.
- **Problem**
  - An application needs to create the elements of a complex aggregate.
  - The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

3.7 Creational Patterns: Builder

- **Example**
  - The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.
  - This pattern is used by fast food restaurants to construct children's meals.
  - Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur).
  - Note that there can be variation in the content of the children's meal, but the construction process is the same.
  - Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same.
  - The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag.
  - The drink is placed in a cup and remains outside of the bag.
  - This same process is used at competing restaurants.

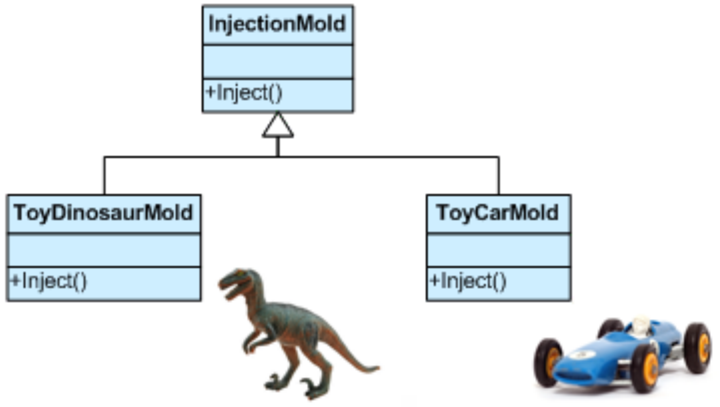


3.7 Creational Patterns: Factory Method

- **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
  - Defining a "virtual" constructor
  - The new operator considered harmful
- **Problem**
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation

3.7 Creational Patterns: Factory Method

- **Example**
  - The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate
  - Injection molding presses demonstrate this pattern
  - Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes
  - The class of toy (car, action figure, etc.) is determined by the mold

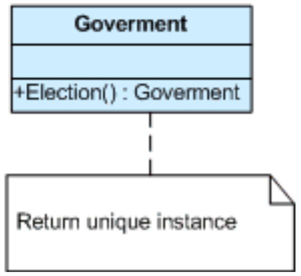


3.7 Creational Patterns: Singleton

- **Intent**
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated "just-in-time initialization" or "initialization on first use".
- **Problem**
  - Application needs one, and only one, instance of an object.
  - Additionally, lazy initialization and global access are necessary.

3.7 Creational Patterns: Singleton

- **Example**
  - The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element.
  - The office of the President of Ireland is a Singleton.
  - The Irish Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession.
  - As a result, there can be at most one active president at any given time.
  - Regardless of the personal identity of the active president, the title, "The President of Ireland" is a global point of access that identifies the person in the office.



3.8 Behavioural Patterns

- **Introduction**
  - Behavioural Patterns
    - Behavioural Patterns are concerned with algorithms and the assignment of responsibilities between them. The operations contributing to a single algorithm may be split up between different classes. The behavioural patterns capture ways of expressing the division of operations between classes and optimise how the communication should be handled.
    - Examples:
      - How can we choose which algorithm to use from a family of algorithms?
      - How can we encapsulate behaviour in an object and delegate requests to it?
- **Behavioural Patterns**

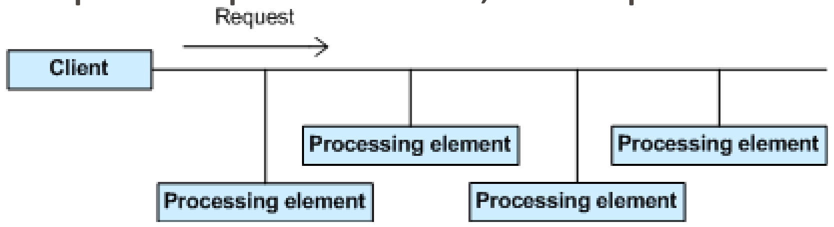
Command	**Interpreter**	**Iterator**
Mediator	**Memento**	**Observer**
State	**Strategy**	**Template Method**
Visitor	**Chain of Responsibility**	

3.8 Behavioural Patterns

- **Chain of responsibility** delegates commands to a chain of processing objects.
- **Command** creates objects which encapsulate actions and parameters.
- **Interpreter** implements a specialized language.
- **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento** provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State** allows an object to alter its behavior when its internal state changes.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

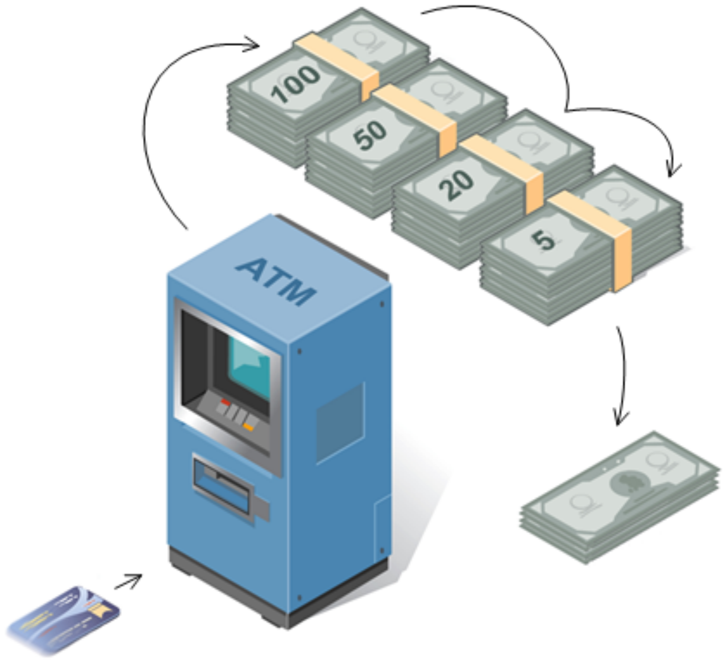
3.8 Behavioural Patterns: Chain of Responsibility

- **Intent**
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
  - An object-oriented linked list with recursive traversal.
- **Problem**
  - There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.



3.8 Behavioural Patterns: Chain of Responsibility

- **Example**
  - The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request.
  - ATM use the Chain of Responsibility in money giving mechanism.

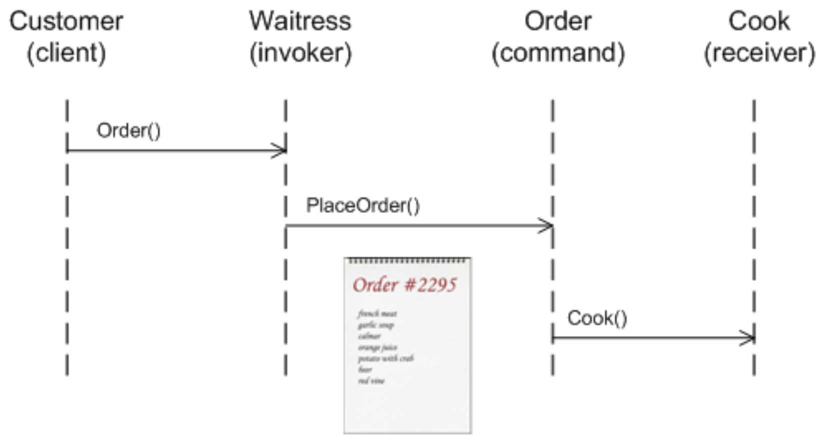


3.8 Behavioural Patterns: Command

- **Intent**
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - Promote "invocation of a method on an object" to full object status
  - An object-oriented callback
- **Problem**
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

3.8 Behavioural Patterns: Command

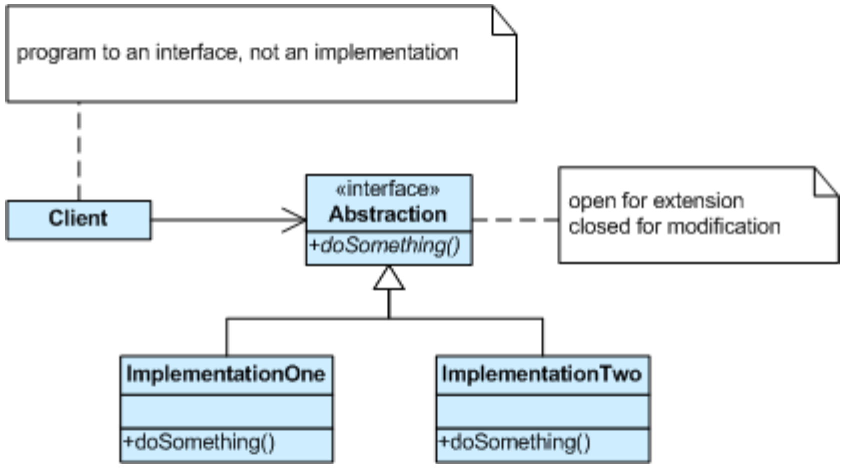
- **Example**
  - The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.
  - The "check" at a diner is an example of a Command pattern.
  - The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check.
  - The order is then queued for a short order cook.
  - Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



3.8 Behavioural Patterns: Strategy

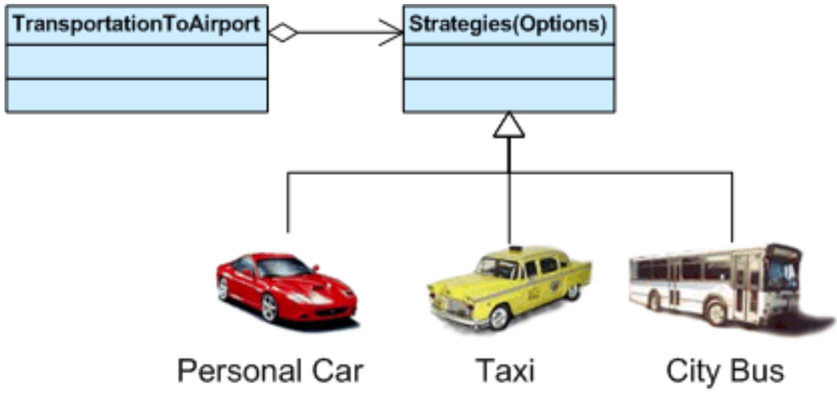
- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.

- **Problem**
  - One of the dominant strategies of object-oriented design is the "open-closed principle"
  - Figure demonstrates how this is routinely achieved – encapsulate interface details in a base class, and bury implementation details in derived classes
  - Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes



3.8 Behavioural Patterns: Strategy

- **Example**
  - A Strategy defines a set of algorithms that can be used interchangeably.
  - Modes of transportation to an airport is an example of a Strategy.
  - Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.
  - For some airports, subways and helicopters are also available as a mode of transportation to the airport.
  - Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.
  - The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.

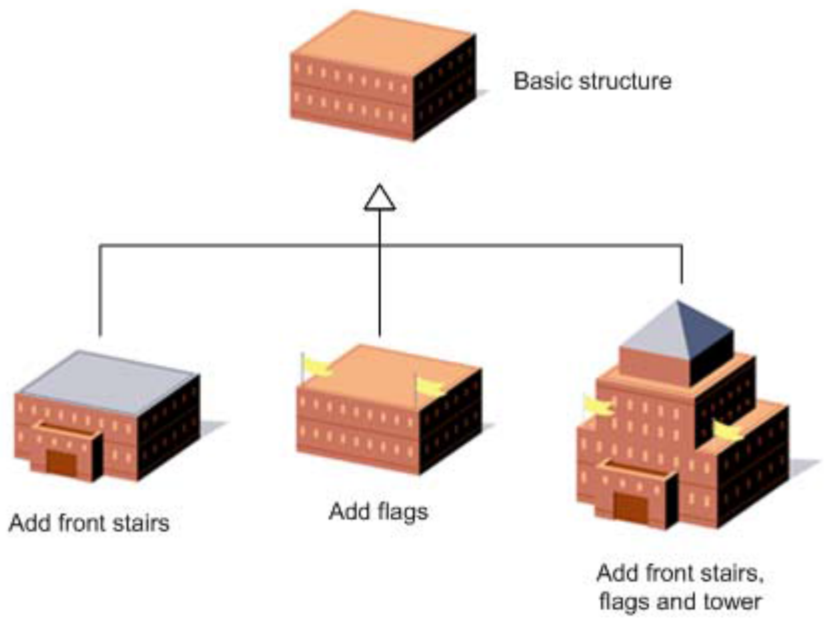


3.8 Behavioural Patterns: Template Method

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - Base class declares algorithm 'placeholders', and derived classes implement the placeholders.
- **Problem**
  - Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

3.8 Behavioural Patterns: Template Method

- **Example**
  - The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.
  - Home builders use the Template Method when developing a new subdivision.
  - A typical subdivision consists of a limited number of floor plans with different variations available for each.
  - Variation is introduced in the later stages of construction to produce a wider variety of models.



National College of Ireland

Questions?

Mikhail Timofeev
Office 3.18
MTimofeev@ncirl.ie