Creational

Factory

When:

1) a class needs to instantiate a subclass of another class but doesn't know which one
2) a class cannot anticipate the class of objects it must create
3) a class wants it subclasses to specify the objects it creates

Advantages:

1) decouples the implementation of the product from its use
2) able to add additional products or change an implementation of a product and it will not affect the creator. Can change design easier.

Abstract Factory (factory of factories)

When:

1) when we need to deal with multiple factories
2) when a family of related products is designed to be used together and you need to enforce this constraint

Advantages:

1) promotes consistency among products
2) makes exchanging product families easy

Singleton

Notes:

1) Use Bill Pugh method (inner static helper class)
2) Choose Singleton over Dependency Injection if you need functionality over multiple classes and multiple layers (otherwise you pass injector object to all of these multiple classes). DI is better for testing purposes, to avoid statics and for non-stable dependencies (external service, file system or database).

Builder

When:

1) When creating a complex object in terms of parameters
2) When you want to create different representations of the object
3) Works better then Factory patterns if the Object to be created has a lot of attributes

Advantages:

1) Hides the complex construction process and represents it as a simple process
2) Allows objects to be constructed in a multistep and varying process
3) Hides the internal representation of the product from the client
4) Product implementations can be swapped in and out
5) Construction process must allow different representatiosn for the object that is constructed

Prototype

When:

1) Used when creation of an object is costly and you have a similar object already existing. Uses java cloning to copy the object (shallow) or de-serialization when you need deep copies.
2) When instances of a class can have one of only a few different combinations of state

Notes:

1) Object which you are copying should provide the copying feature
2) Instead of cloning, you can try and create a copy constructor that returns a new instance of a class
3) Another alternative is to serialize an object and then immediately deserialize it. This is slower than cloning.

Advantages:

1) More efficient and cheaper to create in some cases

Structural

Adapter(Wrapper)

When:

1) Works as a bridge between two incompatible interfaces

2) Vendor has a new library but it has a different class library than the previous vendor. You don't want to change the existing code and you can't change the vendor code. So you write a new class that adapts the new vendor interface into your code

Bridge

When:

1) When you want to avoid a permanent binding between an abstraction and its implementation
2) When both the abstractions and their implementations should be extensible by subclassing
3) When changes in the implementation of an abstraction should have no impact on clients
4) When you want to hide the implementation of an abstraction completely from clients
5) When you have a ton of implementation classes

Advantages:

1) Decouples an implementation so that it is not bound permanently to an interface
2) Abstraction and implementation can be extended independently
3) Changes to the concrete abstraction classes do not affect the client

Notes:

1) Adapter is used after classes have been designed and Bridge is used up front in a design

Composite

When:

1) Allows you to manipulate a single instance of the object just as you would manipulate a group of them
2) Using multiple objects in the same way with nearly identical code to handle them

Decorator

When:

1) Add new functionality to an existing object without altering its structure
2) Add responsibilities to individual objects, not to an entire class
3) Add or remove responsibilities by simply attaching or detaching decorators
4) When extension by subclassing is impractical or not allowed (final)

Façade

When:

1) When you want to provide a simple interface to a complex subsystem
2) When there are many dependencies between clients and the implementation classes of an abstraction
3) When you want to layer your subsystems, use a façade to define an entry point to each subsystem level

Advantages:

1) Shields clients from subsystem components
2) Can eliminate complex or circular dependencies
3) Does not prevent applications from using subsystem classes if they need to

Flyweight

When:

1) When you need to reduce the number of objects created (memory usage). Crucial for low memory devices such as mobile or embedded.
2) When an application uses a large number of objects
3) When storage costs are high because of the sheer quantity of objects
4) When many groups of objects may be replaced by relatively few shared objects (once extrinsic state is removed)

Advantages:

1) Reduces the number of object instances at runtime (saves memory)
2) Centralizes state for many virtual objects into a single location

Proxy

When:

1) When we want to provide a controlled access of a functionality (might need to defer full cost of creation and initialization until we actually need to use it)
2) To provide a wrapper implementation for better performance

Chain of Responsibility

When:

1) When you want to decouple a request's sender and receiver
2) When multiple objects determined at runtime, are candidates to handle a request
3) When you do not want to specify handlers explicitly in your code
4) When you want to issue a request to one of several objects without specifying the receiver explicitly (you might not know who needs to handle it – chain determines who will handle it)

Advantages:

1) Frees an object from knowing which other object handles a request
2) Both the receiver and the sender have no explicit knowledge of each other
3) Simplifies your object (it does not have to know the chain's structure or keep direct references to its members. Only a single reference to the successor and it just goes down the chain)
4) Allows you to add or remove responsibilities dynamically by changing the members or order of the chain

Command

When:

1) Used for undo/redo operations (Command's execute method can store state for reversing effects in the command itself)
2) Useful when modeling transactions (which can be responsible for changes in data)
3) When you want to parameterize objects by an action to perform
4) When you want to specify, queue and execute requests at different times (A command object can have a lifetime independent of the original request)
5) When you want to support logging changes so that they can be reapplied in case of a system crash
6) When you want to implement a callback method

Interpreter

When:

1) When there is a language to interpret and when the language is simple (grammar does not have many rules)
2) Used for scripting and programming languages

Iterator

When:

1) When you need a way to access the elements (forwards, backwards , concurrently) of an aggregate object (container) sequentially without exposing its underlying representation
2) When you need multiple traversals of aggregate objects

Mediator

When:

1) When you need an intermediary who can track the communication between two objects
2) When you need to reduce communication complexity between multiple objects or classes
3) When you need a router between objects
4) When you need to centralize complex communications and control between related objects

Advantages:

1) Increases the reusability of the objects supported by the mediator by decoupling them from the system
2) Simplifies maintenance of the system by centralizing control logic
3) Simplifies and reduces the variety of messages sent between objects in the system
4) Allows you replace one object in the structure with a different on without affecting the classes and interfaces

Memento

When:

1) When you need to save the state of an object, so you can go back to the specified state in the future
2) When you need an undo

Advantages:

1) Easy to implement, but can be time consuming. Might have to consider serialization

Observer

When:

1) When a change to one object requires changing others and you do not know how many objects need to be changed
2) When multiple objects are dependent on the state of one object
3) When objects should be able to notify other objects without making assumptions about who these objects are.

State

When:

1) When you need an object where its behavior is the result of the function of its state
2) When you want to change behavior based on state without if/else logic or switch logic
3) When an object's behavior depends on its state and it must change its behavior at runtime depending on that state.
4) When operations have large multipart conditional statements that depend on the object's state (usually represented by one or more enumerated constants)

Advantages:

1) Puts all behavior associated with a state into one object
2) Allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement.
3) Helps avoid inconsistent states since state changes occur by rebinding one variable rather than several
4) Very easy to add more states for additional behavior. Code is more robust, maintainable and flexible

Strategy

When:

1) When you want to select the behavior of an algorithm dynamically at runtime
2) When many related classes differ only in their behavior, strategies provide a way to configure a class with one of many behaviors
3) When an algorithm uses data that clients should not know about

Advantages:

1) New algorithms complying with the same interface can be easily introduced
2) Applications can switch strategies at runtime (polymorphism)

Template Method

When:

1) When you need to support multiple algorithms that behave conceptually the same but have different implementations for each of their steps

Visitor

When:

1) When you need to define a new operation without changing the classes of the elements (of an object structure) on which it operates
2) When you need one or more operations to be applied to a set of objects at runtime
3) When an object structure (collection or list) contains many classes of objects with differing interfaces and you want to perform operations on these objects that depend on their concrete classes

4) When you want to add capabilities to a composite of objects and encapsulation is not important
5) When the object structure is shared by many applications, use the visitor pattern to put operations in just those applications that need them