Design Patterns notes.

General.
Patterns are classified by *purpose* and *scope*.
The purpose is defined as :

- creational
  Creational patterns deal with the creation of objects and help to make a system independent of how objects are *created*, *composed* and *represented*. They also enable flexibility in *what* gets created, *who* creates it, *how* it gets created and *when* it gets created.

- structural
  Structural patterns deal with how objects are arranged to form larger structures

- behavioural
  Behavioural patterns deal with how objects interact, the ownership of responsibility and factoring code in variant and non-variant components.

The scope is defined as :

- class - static relationships through class inheritance (*white-box* reuse)
- object - dynamic relationships through object composition (*black-box* reuse) or collaboration

Pattern summary.
There are ***5*** creational patterns, ***7*** structural patterns and ***11*** behavioural 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 Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Pros/cons of using Design Patterns.
Pros.

- quality, flexibility and re-use
  Design Patterns capture solutions to common computing problems and represent the time, effort and experience gained from applying these solutions over numerous domains/iterations. Generally systems that use Design Patterns are elegant, flexible and have more potential for reuse

- provide a common frame of reference for discussion of designs
- patterns can be combined to solve one or more common computing problems

- provide a common format for pattern specification
  Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences

Cons.

- complexity.
  Design Patterns require a reasonable degree of study and can be difficult for some designers to grasp. Junior designers/developers may not have encountered Design Patterns and have to learn them before they can be productive on a project.

Creational Patterns Summary.

N.B. in the following pattern descriptions, the terms abstract class and interface are equivalent

Pattern	***Description***	***Pros/Cons***
Abstract Factory *"Provide an interface for creating families of related or dependent objects without specifying their concrete classes"*	A "family" of abstract create methods (each of which returns a different *AbstractProduct*) are grouped in an *AbstractFactory* interface. *ConcreteFactory* implementations implement the abstract create methods to produce *ConcreteProducts*.	Pros. • shields clients from concrete classes • easy to switch product family at runtime – just change concrete factory • "keep it in the family" – enforces product family grouping Cons. • adding a new product means changing factory interface + all concrete factories
Builder *"Separate the construction of a complex object from it's representation so that the same construction process can create different representations"*	An appropriate *ConcreteBuilder* (implements *Builder)* is constructed and associated with a *Director*. The *Director* traverses an object graph and passes each object to the Builder. The Builder uses each object to build-up a complex *Product* over time. When the object graph has been fully traversed, the final Product can be retrieved from the Builder.	Pros. • separates complex construction from (re)presentation • shields the Director from the algorithm and internal structure used to build the Product • enables a consolidated Product to be built up over time – e.g. the Product requires info from multiple sources, info available at different times
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"*	An abstract *Creator* class defines an abstract create method (or provides a default create method) which returns an abstract *Product*. A *ConcreteCreator* class implements the abstract create method to return a *ConcreteProduct*. This enables the Creator to defer Product creation to a subclass. *N.B. Factory Method is often used in the Abstract Factory pattern to implement the create methods*	Pros. • shields clients from concrete classes • if a framework uses the Factory Method pattern, it enables third-party developers to plug-in new Products • the Creator create method can be coded to return a default Product Cons. • coding a new Product means writing *two* classes – one for the concrete Product and one for the concrete Creator • static – inheritance based
Prototype *"Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype"*	Objects implement the *clone ()* method of the *Prototype* interface by returning a copy of self. A client maintains a registry of Prototype instances. When a new instance is required, the client invokes clone ()	Pros. • shields clients from concrete classes • the object is the factory - i.e. Product and Creator combined (saves coding a Creator for every Product) • pre-configured object instances – instead of create/set member vars every time Cons. • every Prototype instance has to implement clone () which may not be easy – e.g. circular references, contained elements don't support copying, large number of classes to be retrofitted, etc.
Singleton *"Ensure a class has only one instance and provide a global point of access to it"*	A Singleton is defined with a static getInstance () method, a protected constructor and any other required instance methods. As the constructor is protected, the only way to obtain an instance is through the static getInstance () method. This serves to control the number of instances created/in use by clients.	Pros. • controls access to the instance(s) • controls the number of instances • more flexible than a static class - the instance(s) constructed in getInstance () can be a subclass so method overrides are allowed (can't use method overrides with a static class).

Structural Patterns Summary.

Pattern	***Description***	***Pros/Cons***
Adapter *"Convert the interface of one class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces"*	A concrete *Adapter* class implements methods defined in a *Target* interface by wrapping calls to methods in a concrete *Adaptee* (and also provides equivalent functionality for required Target methods if they don't exist in Adaptee). A *Class Adapter* uses multiple inheritance of Target and Adaptee. With an *Object Adapter*, Adapter contains an Adaptee and forwards requests.	Pros. • enables interoperability – especially useful when using one or more third-party class libraries in your code • highlights the Target "contract" – e.g. if shipping reusable components, include a default adapter for use by clients • *object adapter* – a single Adapter can adapt many Adaptees (including subclasses) • *class adapter* – automatically inherit Adaptee methods; inherited methods can be overridden Cons. • type adapted – to the outside world, the Adaptee looks like an Adapter (can't pass to Adaptee methods unless a two-way adapter is implemented) • *object adapter* – need to write tedious method mapping/delegation code • *class adapter* – need to provide an adapter for each subclass • *class adapter* – multiple-inheritance of potentially similar interfaces (risk of method name collisions)
Bridge *"Decouple an abstraction from its implementation so that the two can vary independently"*	*Abstraction* (an abstract base class) provides core functionality for it's subclasses by aggregating *primitive* methods from an *Implementor* (an abstract class/interface) into high-level methods. *ConcreteImplementor* classes provide specific implementations of the primitive methods. This facilitates a clean separation between elements that are common (e.g. Window.draw ()) and elements that are specific (e.g. XWindow.draw ())	Pros. • decouples abstraction from implementation – clean separation between common aspects and specific differences • extensible – abstraction and implementation can evolve independently • shields clients from concrete classes – a change in the implementation doesn't require the client to be updated • implementation can be swapped at runtime
Composite *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly"*	*Component* (an abstract base class) is sub-classed into either a *Leaf* or a *Composite*. A Composite contains one or more Components – i.e. a Leaf or another Composite. This enables a client to view a single item or a group of items as one type – a Component.	Pros. • facilitates uniform view - clients are shielded from details of whether a Component is a Leaf or Composite • easy to add new components – everything referenced by Component Cons. • referring to either as Component makes it too general – can't control what Components make up a Composite without explicitly checking

Structural Patterns Summary (2).

Pattern	***Description***	***Pros/Cons***
Decorator "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality"	*ConcreteDecorator* (subclass of *Decorator*) classes wrap *ConcreteComponent* (subclass of *Component*) classes to transparently extend their functionality. This is achieved by added functionality before/after dispatching method calls to the Component. Transparency is achieved as the Decorator interface matches the Component interface	Pros. • more flexible than inheritance - functionality can be extended on an instance basis, at runtime, etc. • promotes reuse – a Decorator can enhance anything that implements *Component* • enables recursive composition – can construct a chain of Decorators Cons. • too transparent – a Decorator looks just like the original Component • difficult to conceptualise – lots of fine-grained Decorators connected in lots of different ways
Facade *"Provided a unified interface to a set of interfaces in a sub-system. Facade defines a higher-level interface that makes the sub-system easier to use"*	A *Facade* provides a simplified view of a complex object model by aggregating methods from multiple *subsystem classes* into a few high-level methods. Communication is one-way – the Facade knows about the subsystem classes but the subsystem don't have any knowledge of the Facade.	Pros. • shields the client from the complexity of the subsystem • decouples the client from the subsystem – relationship management is externalised to the Facade • performance - batch several method calls into one • control – provides a central point to exercise control
Flyweight *"Use sharing to support large numbers of fine-grained objects efficiently"*	A pool of common objects (*Flyweights)* are shared by splitting the object state into static (*intrinsic*) and instance specific (*extrinsic*) components. When invoking methods on the Flyweight, the client must pass the extrinsic state in the method. The pool is managed by a *FlyweightFactory* which ensures that objects are added to the pool on first request and retrieved from the pool thereafter.	Pros. • support a large number of clients using a relatively small pool Cons. • overhead – have to pass in extrinsic state each time
Proxy *"Provide a surrogate or placeholder for another object to control access to it"*	A common *Subject* interface is defined and implemented by a *RealSubject* class and a *Proxy* class. The Proxy acts as a middle-man between the client and the RealSubject. As far as the client is concerned, the Proxy looks identical to the RealSubject (it's transparent).	Pros. • provides a layer-of-indirection between the client and the RealSubject which can be used to implement a variety of useful features (load-on-demand, location transparency, access control, reference counting) Cons. • too transparent – as the Proxy is transparent, the client isn't aware of how the Proxy should be used (e.g. with location transparency, every method call is a remote call)

Behavioural Patterns Summary.

Pattern	***Description***	***Pros/Cons***
Chain of Responsibility *"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."*	Decouples the sender of a request from the "ultimate" receiver. The request is passed along a chain of potential *Handler*s until one of them deals with it. If a handler doesn't wish to deal with the request, it passes the request to it's *successor*	Pros. • reduced coupling • flexible responsibility – handling the request is optional Cons. • the request may get handler by the default handler which may not know what to do with it
Command *"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."*	The purpose of the Command pattern is to decouple an event generator (the *Invoker*) from the event handler (the *Receiver*). A *ConcreteCommand* class (sub-classed from *Command*) defines an *execute ()* method which calls the appropriate method on the Receiver (the *action* method). The client is responsible for associating the Receiver with the Command and then the Command with an Invoker. *N.B. 1:1:1 mapping between Invoker, Command and Receiver.*	Pros. • decouples Invoker from Receiver – makes Receiver more re-usable as it doesn't manage the relationship with the Invoker • Command encapsulate a request – requests can be stored so they can be undone, processed at a later time, etc. • extensible – easy to add new Commands • macros – commands can be grouped into *macros* so that multiple commands can be run at once • dynamic – e.g. different Commands, multiple Invokers, decide at runtime, etc. Cons. • can't centralise related action methods in one Command class - only one method is used (execute ())
Interpreter *"Given a language, define a representation for its grammar along with an interpreter that use the representation to interpret sentences in the language."*	***Don't care***	

Behavioural Patterns Summary (2).

Pattern	***Description***	***Pros/Cons***
Iterator *"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."*	A common OO requirement is traversal of an aggregate structure. The implementation of the traversal is factored out of the *Aggregate* class into an *Iterator*. The Factory Method pattern is used by a *ConcreteAggregate* to create a *ConcreteIterator*. The ConcreteIterator keeps track of the "current" position. The same interface is used to iterate regardless of the underlying aggregate structure.	Pros. • shields the client from the aggregate's internal representation • the aggregate can be iterated in many different ways (i.e. multiple ConcreteIterators) • more than one iterator can be active – the iterator stores the current state so each is self contained • simplifies the ConcreteAggregate code – iterator is in a separate class Cons. • uses Abstract Factory so have to define a ConcreteAggregate in addition to the ConcreteIterator • if the underlying aggregate is updated while using an Iterator, the operation of the Iterator may be undefined.
Mediator *"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."*	A collection of related classes called *Colleague*s (sub-classed as *ConcreteColleague*) need to inform each other when an event occurs. Rather than couple every colleague to every one of it's peers, each Colleague publishes the event to a *Mediator* (sub-classed as *ConcreteMediator)*. The Mediator then republishes the event to the other Colleagues. Communication is therefore two-way – the Mediator knows about the Colleagues and vice-versa.	Pros. • promotes a loose coupling between the Colleagues – instead of a Many:Many publish, it's a Many:1 (Colleagues to Mediator) followed by a 1:Many (Mediator to Colleagues) • promotes reuse – Colleagues aren't bogged down with relationship management code so can be reused in other circumstances • centralizes relationship management in the Mediator Cons. • the Mediator can become very complex and difficult to maintain
Memento *"Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."*	Uses an *Originator* (managed a contained Memento obj), *Memento* (snapshot of originator state, preserves encapsulation) and a *Caretaker* (manages Memento objects)	Pros. • preserves encapsulation • state can be stored and reloaded later on
Observer *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*	One or more *Observer*s (sub-classed as *ConcreteObserver)* can be registered with a *Subject* (sub-classed as *ConcreteSubject*). When the state of the Subject changes, all registered Observers are notified. Two notification models are available : *push* (the state change is sent with the notification) and *pull* (the notification is the event only, if the Observer wants to see the state change it requests it from the Subject)	Pros. • abstract coupling of Subject and Observer – Subject doesn't care what an Observer does with the event, just notifies it • supports broadcast – in theory, any number of Observers can be supported (doesn't work in practice) Cons. • the client to the Subject works in isolation – isn't aware that setting the state could cause a cascade of event notifications

Behavioural Patterns Summary (3).

Pattern	***Description***	***Pros/Cons***
State *"Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class."*	A common *State* class is subclassed for all possible states. Each subclass restricts the operation of common methods based on it's state. Current state is stored in a *Context*; next state is return by the current State subclass when *handle ()* is called	Pros. • collects actions + transitions into state specific classes Cons. • doesn't scale – i.e. if large num of states/actions
Strategy *"Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."*	Algorithms are defined as *Strategy* classes. Related algorithms are grouped into a family of Strategy classes. A *StrategyContext* class contains all required info for the algorithm defined in the Strategy and the two classes work in conjunction to execute the algorithm.	Pros. • a family of Strategy classes is available – pick the most suitable or as directed by an external decision making process • simplified/cleaner code – instead of lots of "if" statements or subclasses each implementing an algorithm, one "dispatcher" class can provide all relevant Strategy's on request Cons. • clients must know the classes available in the family - clients instantiate Strategy instances when the StrategyContext is created
Template Method *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changes the algorithms structure"*	Capture the *invariant* behaviour of an algorithm in an abstract base class using high-level methods. Define the *variant* behaviour as *abstract primitive methods* so that concrete sub-classes can provide implementations. The high-level methods are defined using a combination of primitive methods and methods defined in the abstract base class. *N.B. basically a behavioural version of the Factory Method*	Pros. • shields the client from the details of the variant behaviour • quality & productivity – only the variant behaviour needs to be implemented
Visitor "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"	A client traverses an object graph and for each element invokes *accept (Visitor v)* which in turn calls back on the visitor with itself – i.e. *v.visit (this).* Consequently the Visitor gets notified when an object is traversed and what type the traversed object is. Each Visitor subclass has to support every type of object that will be traversed	Pros. • cleaner code – factors out type specific event handling from classes and centralises it in a Visitor • easy to add a new "operation" for all *Visitable* classes – an operation is implemented as a Visitor subclass with a handler method for each Visitable object type • can traverse multiple object types in the same traversal – unlike Iterator which can only traverse one type at a time • useful for running a variety of reports – without Visitor every class that you'd want to report on would have to have a custom method per report Cons. • if a new Visitable class is added, all Visitor subclasses have to be extended to support it • might break encapsulation - the Visitor needs access to the elements details

Comparison of patterns

Adapter

- Adapter vs. Bridge
  Bridge is used to support interoperability at design time – i.e. to support current implementations and future variations thereof.
  Adapter is used to support interoperability after design time – with existing classes that potentially cannot be modified (e.g. third-party libraries); to support unknown or unplanned interoperability in the future.

- Adapter vs. Proxy
  A Proxy is a surrogate for the Target so the interface is identical.
  In contrast, an Adapter changes the interface (the Adaptee's).

- Adapter vs. Facade
  Facade defines a new, simplified interface.
  In contrast, an Adapter reuses an existing interface (the Adapter's)

Decorator

- Decorator vs. Strategy
  Decorator changes the "skin", Strategy changes the "guts"

- Decorator vs. Adapter
  Decorator extends behaviour while maintaining the interface
  Adapter appears change the interface.

Visitor

- Visitor vs. Iterator
  Visitor can traverse different object types within the same traversal.
  Iterator only traverses one object type per traversal.

- Visitor vs. Chain of Responsibility
  C.O.R. works up the handler hierarchy – from specialized to generic
  A Visitor subclass is totally specific to that "operation".

Patterns in J2EE

General.

- Bridge
  Anything that provides a generic interface to a vendor specific product – JDBC, JMS, JNDI, JavaMail.

- Facade.
  Anything that hides the complexity from the client – InitialContext, Connection, DataSource

Servlets / JSPs

- Decorator, Chain Of Responsibility – Servlet 2.3 Filter
- Singleton - ServletContext, only one per application

JDBC

- Iterator – ResultSet

JNDI

- Iterator - NamingEnumeration, etc.
- Observer - EventContext.addNamingListener (.., NamingListener);

EJB

- Proxy, Facade - Remote interface
  The stub and skeleton combined act as a proxy to a remote EJB object.
  The stub and skeleton combined act as a facade – hides the networking details from the client.

- Abstract Factory - EJBHome
  A client gets a reference to a home object which implements the EJBHome interface (analogous to AbstractFactory). The client uses the home object to create an EJB object which implements EJBHome (analogous to AbstractProduct).
  N.B. with EJB the create methods only return a single Product so only one in the family

  It's tempting to say that the FactoryMethod is used in EJBHome but on balance it doesn't quite match : the container implementation of the EJBObject may defer creation to a subclass but it's unlikely (create methods are different for every EJB); the EJBObject adapts the bean class interface

- Decorator , Adapter - Remote interface
  The container provides implementations for EJBHome and EJBObject. The implementations apply transactions/security to methods before delegating the request to the bean class – e.g. create () checks if the role is allowed to execute the method, if so bean.ejbCreate () is executed

- Facade - Session Facade (a single session bean method manipulates multiple entity beans)
- Memento - Value Objects, SFSB activation/passivation
- Command - transaction logging

- Flyweight - Instance pooling
  A SLSB may contain intrinsic state (e.g. a socket connection). When a client uses a SLSB they pass in extrinsic state (e.g. the parameters to the method call). Consequently, a small pool of objects can support a large number of clients.

- Observer - EJB 2.0 MessageDrivenBean
- Interpreter - EJB 2.0 QL

- Template Method - EJB 2.0 CMP
  Persistent fields / relationships declared as abstract methods in the bean class. The bean uses the abstract/primitive methods in other methods. The container subclasses the bean class and implements the abstract methods. When a client references the EJB object, they're using the container sub-classed version.

JMS

- Mediator, Observer - Publish/Subscriber
  The MOM acts as a Mediator between Colleagues, which in this case are the Subject(s) and Observer(s)