Commit - Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

Working directory – the folder of the directory locally on the computer

Staging area – a file that stores information about what will go into the next commit

Branch – a lightweight movable pointer to any one commit

Origin vs local– origin/main isn't a local branch per se; it’s a *copy* of the remote server's main and gets updated when you do a fetch. You can't modify it the way you can a local branch, eg by committing onto it.

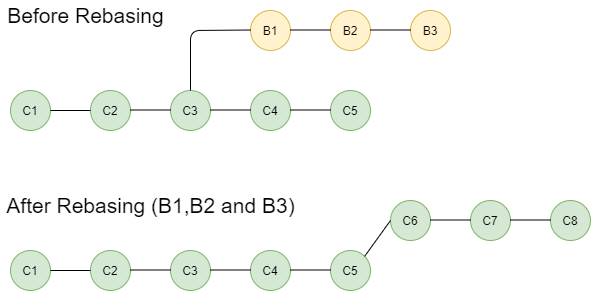
Fetch – updates the branches in your local repo by looking at the branches at the remote and brings them to the local **it doesn’t change your repository files.**

Pull – brings a local branch up to date with the remote branches it also update you branches with remote branches.

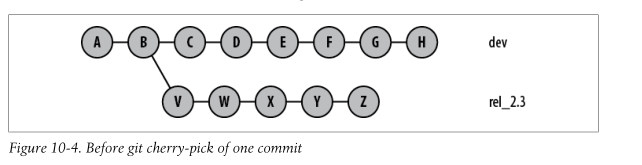
Diff – shows the changes between the working directory and staging area you can give it a branch name it. it will show the difference to it, you can give it a commit hash, it generally comperes between the working directory and what you give it but you can give it certien flags to compere with something else like –-staged.

Checkout – you can use checkout to also create branch using -b flag .

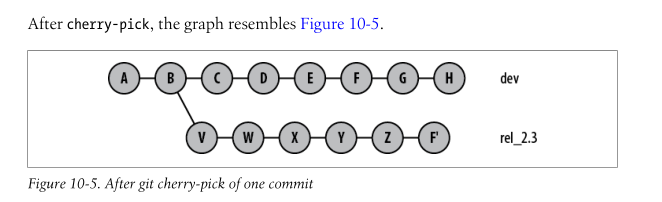
Git rebase – instead of merging it will transfer the commits to the branch listed.



Cherry pick – let you pick a commit to move to the current branch



Cherry picking from dev to rel commit f:



Stash – takes from the working directory the changes from the last commit and put them in the stash, the stash is a stack with the changes added to it

Stash list – lists the stashes saved in the stack uses stash@{0} as the first and increases the number to symbol lower items in the stash.

Apply – takes the first item in the stack and applies the changes to the working folder and keeps the stash as is.

Pop – does as apply but removes the item from the stack

Drop – removes from the stack

Clear – removes all from the stack

You can stash untracked files using -u flag

-a to stash all files including those ignored by gitignore

Stack – use cases: function calls- to where to jump in memory for the next line of code, its used in compiles to check syntax of code. Its being used often in cyber security because you crash or even invade programs by abusing the stack and causing an overflow, for example stack overflow and buffer overflow.

Bisect – uses binary search to try and find a bug you start by letting it now the bad commit usealy the head or current commit your working on and you give it the last commit you checked that was good then it starts asking you by binary searching the commits between them if they were good or bad.

Git reset soft – keeps the staging area and working directory unchanged but moves the head pointer to specified commit. Main use case if you want to combine commits together.

Hard – resets also the staging area and working directory. If you want to go back to a certain commit completely without an option to undo.

Changelog – between releases you put a change log that will describe what has changed in this release uses it shows the bugs fixed the new features added. You can add what ever you want to it because its also a MD file that you can change in what ever way you want.

Git attributes – it’s a file that has some file configurations for the repo for example EOL configuration, binary files ending, custom drivers for diff command and merge. Etc

Git commit hash – uses SHA-1 its based on the contents of the files, metadta, time etc of the commit.

Git head – the pointer which points to the current branch you are working on.

Git reference is a way to reference commits relative to the head commit using mainly using ~ and ^

^- symbolises the perant commit meaning the commit you can use it to travel thru the perants meaning ^2 will go to the perant of the perant.

~ - symbolises the direct commit before it. ~2 two direct commits before it on the same **line (branch)**

They can be combined.

Tags – are being used to mark specific points in your repository history as being important. They are often used to mark release points

Environment – its not native for git but is added to the hosts of git, it allows you to config variables, create tasks that run on commit, etc to each environment, you can set a branch to environment that it will follow, usually you have an environment for at least prod and dev and you can create as much as you want.

There are two types of tags:

Lightweight – marking a commit these tags are just pointers to a specific commit.

Annotated – by using -a flag you create an annotated tag, they are stored as full objects, including checksum, tagger name, email, date, and a tagging message you can also sign it in order to authenticate its creator, you should generally use annotated because of its extra data that being stored, but if you want to create a temp tag you can use lightweight tags.

Semantic versioning – when giving a version the convention is MAJOR.MINOR.PATCH so when tagging you should use this convention for the tag name.

Release – being used adjacent to tags to mark specific points in the repository history.

**Clean code:**

meaningful names:  
intention-revealing names names should clearly express the intent of the variable function or class.

Avoid disinformation – names should not be misleading or ambiguous.

Prononcable names – you should use names that are easy to pronounce because it will be easy to discuss and remember.

Searchable names – names should be easy to find in the code.

Avoid encodings – do not use prefixes or Hungarian (\_name) notation keep names clean and simple.

Functions:

Small functions – should be small and preform a single task

Do one thing – each function should do one thing and do it well.

Depictive names – function names should describe what they do.

Avoid side effect – functions should not have hidden effects and variables that are not clear from the function name or signature.

Functions arguments – should be kept to a minimum.

Comments:

Use comments to explain why something is done and not what is done. Avoid redundant, misleading or obvious comments.

You should strive to write a code that Is clear enough to be understood without comments.

Formatting:

Consistent – pick a format and stick to it.

Vertical – group related lines of code together and use spacing to separate different logical sections.

Horizontal – keep lines short typically less than 80-100 characters. use proper indentations and spacing around operators.

Objects and data structures:

Objects hide their data behind methods, while structs expose their data and have no significant behavior.

DTOS – use DTOs to transfer data between components. For example web applications should not receive from the beckend the full user object but should receive a part of it instead of nulling values In the object create a new object to send.

Error handling:

Use exceptions and not error codes: exceptions are preferred over error codes.

Define exception classes – create specific exception classes for different error types.

Don’t return null - avoid returning null instead exceptions.

Don’t pass null – avoid passing null arguments to functions.

Boundaries:

Third party code – treat third party code as a black box and minimize dependencies.

Encapsulate boundaries: try to isolate external libraries to isolate the rest of the code from changes by using interfaces or adapters.

Unit testing:

Test driven development: write tests before writing code they test.

Clean tests: tests should be readable, maintainable and act as documentation for the code.

Fast, independent, repeatable: tests should run quickly, be independent of each other, and be deterministic (aka produce the same result).

Clases:

Small classes: classes should be small and focused on a single responsibility.

Single responsibility: a class should have only one reason to change. And have one thing they oversee.

Conhesion: methods within a class should be related and work together towards a common purpose.

Systems:

Separations of concerns: different aspects of the system should be managed by different parts of the system.

Dependency injection: use dependency injection to manage dependencies and improve testabily.

Keep the system under control: regularly refactor and clean the code to maintain control over the systems complexity.

Emergence:

Simple design: follow kent becks four rules of simple design: passes all tests reveals intentions no duplication minimal number of classes and methods.

Concurrency (multi thread, process etc):

Keep it simple: concurrency adds complexity, so keep concurrent code as simple as possible.

Limit the scope of data sharding: minimize the amount of data shared between threads.

Smells and heuristics:  
code smells: be aware of common signs of poorly written code (eg., large classes lone methods and excessive comments)

Refactor regularly: continuously improve the code by refactoring to eliminate smells and improve design.

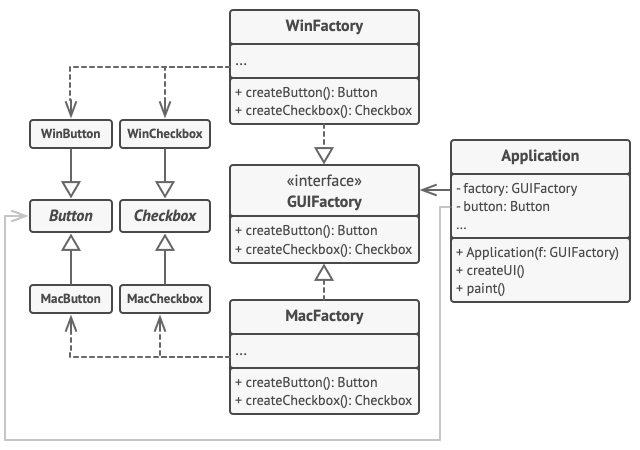
Continuous learning – stay updated with best practices and new techniques in software development.

**Design patterns**

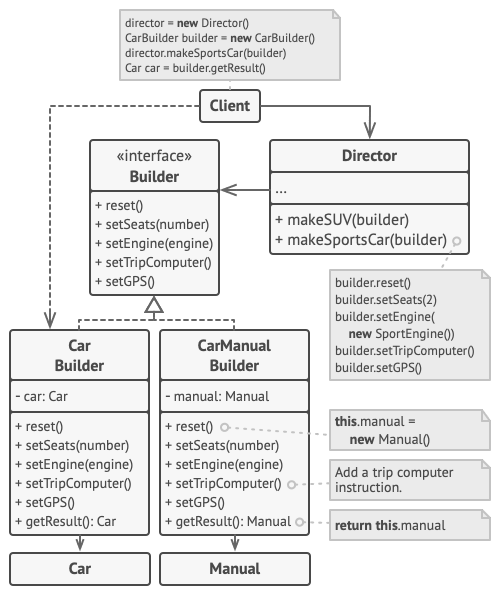
Factory –  The Factory Method separates product construction code from the code that actually uses the product. Therefore it’s easier to extend the product construction code independently from the rest of the code.



Abstract factory - is a way of organizing how you create groups of things that are related to each other.



Builder – when having an an object with a lot of options, you should create an interface with those attributes and the implementation will be in the builder, an engineer object will be in charge of creating object by getting the builder instance and having a method for creating the robot from the engineer object.

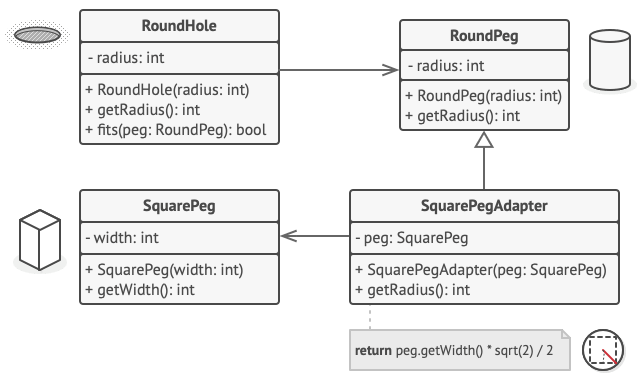


Prototype– by using the clonable interface you can easily clone objects.

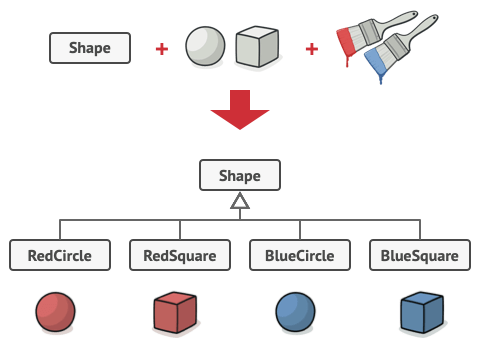
Singleton – if you want one instance of an object you can use it, by creating a static class that will have one instance of that object and have a function to return that instance if it exists else create and return it, its really useful when multi-threading.

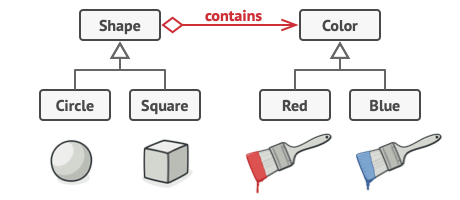


Adapter – when having different functions in different objects that do the same thing or even with different units you can create an adapter to one of the object into another for example to robots that attack one another but one has different functions signatures so you create an adapter in order for them to communicate easily.



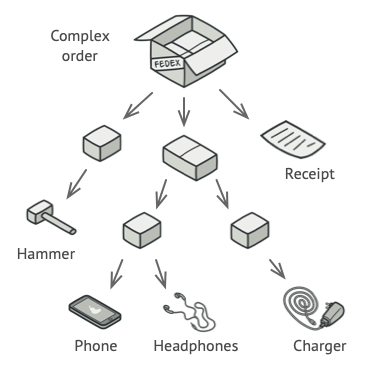
Bridge – sperates an objects interface from its implemantations allowing the two to vary indepndendly, for example colored shapes instead of having multiple shapes that are named with different colors move the color to a class and have it as an attribute to the shape.

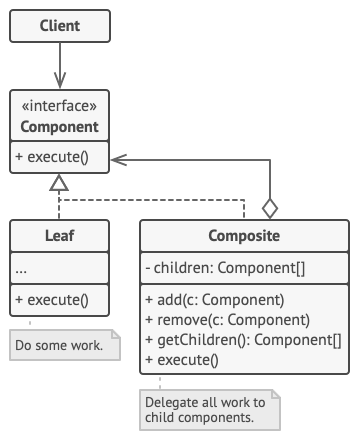




Composite – when wanting to structure a box that can have multiple boxes and items in it you need to define it in that way you can do it by creating a common interface for it and later storing that common interface as an array that can be unpacked later.

An example an interface of a song component you implement it into a song class and you have a song playlist the playlist by itself is a song component.



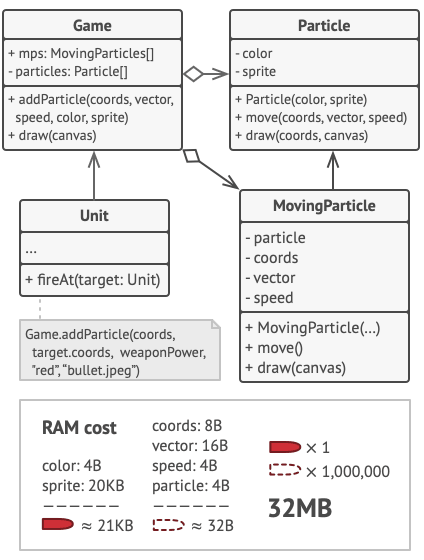


Decorator – when wanting to have an unknown amount of decorators on an object for example pizza you can do it by unwrapping layer by layer from the main object by creating an abstract class that will function as the main object example pizza and having a decorator class that inherits the pizza you can create decorators on to of one another therefore making it easy to calculate a price of the pizza by recursively delayering from the pizza and seeing what is it its very similar to a linked list in that sense because in practice it is a linked list.

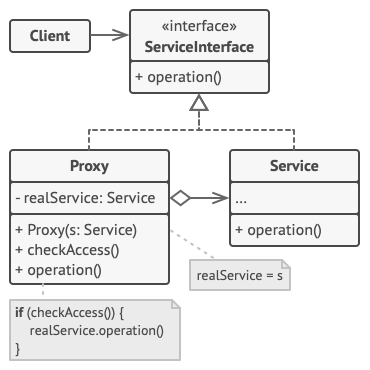
Façade – when calling a shop for an order instead of going thru all of the options an operator in this case the façade is the one who knows were to move you too, the same can be said when you want to convert from certain file types you might need to run different process to do so but instead of needing to go to them one by one you can have a façade and use it for that.

It basically hides the algorithm in a class.

Flyweight – when having a lot of objects that are the same instead of creaeting millions of copys of them you can share those objects instead, lets say you want to paint a million rectangles in different color instead of creating a million rectangles you can use the same rectangle but paint it in different colros, therefore saving a lot of memory and processing time.

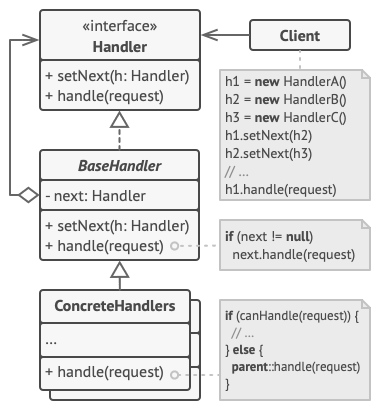


Proxy – when wanting to hide certain states or functions from the users for example an atm cache amount. So you create a class that acts as a proxy.



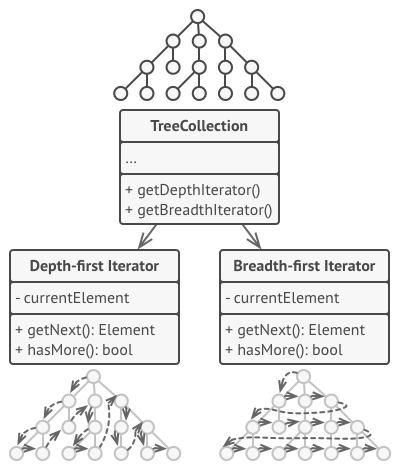
Chian of responsibility – it allows you to send a command without knowing who is responsible to handle the request, the request is passed along by the handlers until one handles it. For example routes in express/fastapi.

It creates a graph of handlers.



Command – instead of having to call a function and limiting Extensibility you can create commands and excuter to execute this commands by creating a common interface to have an excute function that will be called on each command. It allows you to reuse these commands and combine them into more complicated commands.

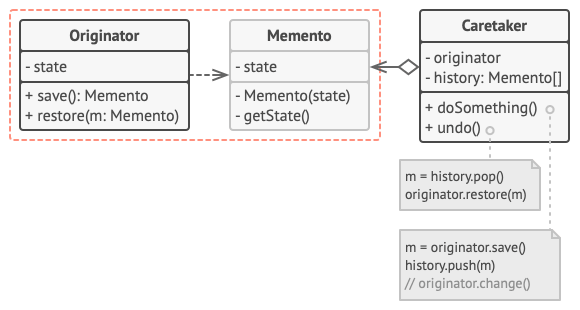
Iterator – when you want to traverse through elements without needing to understand the underlying data structure, and easing the use of the data structure, you do it by litigating the traveling thru the data structure to a dedicated class.



Mediator – in order to reduce coupling between components that communicate between them is by passing their communications thru a mediator. For example when brokers want to trade stocks instead of communicating between them they should use a mediator that will collect the buys and sell offers and see if there is an acceptable offer.

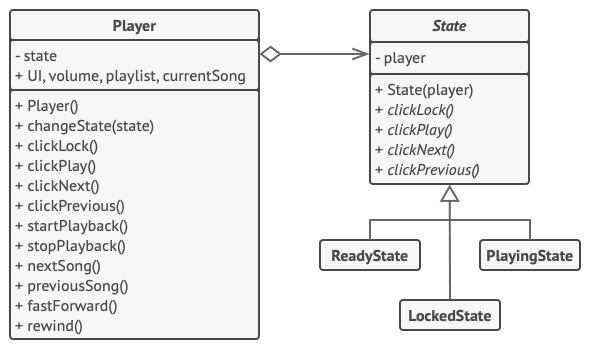
Memento - used to capture and restore an object's internal state without violating encapsulation, allowing you to revert an object to a previous state. By saving an instance of the state as a memento you can store and use the state of said object later on with ease.





Observer – when having objects watching for one object changes that one object should store and notify the other objects on his changes.

State – instead of having a lot of switch cases to swap between states you can have an interface and classes to describe the states and what states should follow when something specific is done.



Strategy – when you have multiple different algorithms to do the same thing you can create an interface that will gather under the same function signature therefore making it easier to use them without a lot of complications.



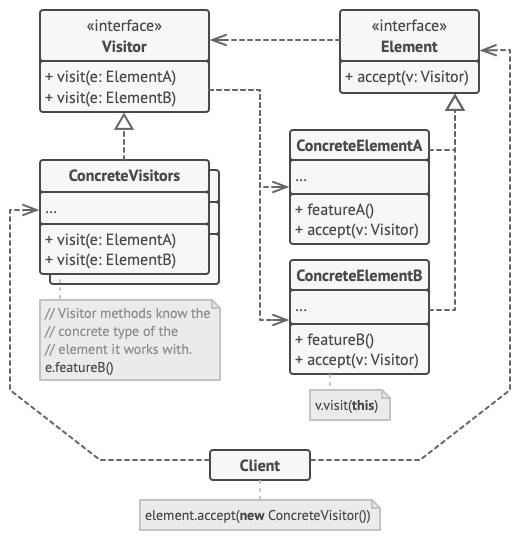
Template - The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method.* The steps may either be abstract, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).



Visitor – imagine a seasoned insurance agent who’s eager to get new customers. He can visit every building in a neighborhood, trying to sell insurance to everyone he meets. Depending on the type of organization that occupies the building, he can offer specialized insurance policies:

* If it’s a residential building, he sells medical insurance.
* If it’s a bank, he sells theft insurance.
* If it’s a coffee shop, he sells fire and flood insurance.

The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes. The original object that had to perform the behavior is now passed to one of the visitor’s methods as an argument, providing the method access to all necessary data contained within the object.



Database design:

**1. Database Design Fundamentals:**

* **Schemas:** Schemas define the logical structure of a database, specifying how data is organized and how relationships among data are maintained. For instance, a schema for a library database might include tables for books, authors, and borrowers, with relationships between these entities defined.
* **Normalization:** Normalization is the process of organizing data to reduce redundancy and improve data integrity. This involves dividing a database into two or more tables and defining relationships between the tables. The primary goals of normalization are to minimize duplicate data, ensure logical data storage, and support efficient data retrieval.
* **Views and Access Control:** Views are virtual tables created by querying data from one or more underlying tables. They can simplify complex queries, enhance security by restricting access to specific data, and provide a level of abstraction. Access control involves setting permissions for different users or roles, ensuring that sensitive data is only accessible to authorized individuals.
* **DBMS Selection:** Choosing a Database Management System (DBMS) depends on factors like data volume, transaction speed requirements, scalability, and specific use cases. SQL databases (like MySQL, PostgreSQL) are suited for structured data and complex queries, whereas NoSQL databases (like MongoDB, Cassandra) are ideal for unstructured or semi-structured data and provide horizontal scalability.

**2. OLTP vs. OLAP:**

* **OLTP (Online Transaction Processing):**
  + **Purpose:** Supports day-to-day transactional applications like order processing, banking transactions, and retail operations.
  + **Design:** Optimized for quick, real-time transaction processing and high data integrity.
  + **Data Characteristics:** Data is up-to-date, typically stored in a normalized format to reduce redundancy. The database size is usually smaller, often measured in gigabytes.
  + **Users:** Used by a large number of users, including employees and customers, who require real-time access to transactional data.
  + **Queries:** Involves simple, frequent queries and updates, such as inserting, updating, or deleting records.
* **OLAP (Online Analytical Processing):**
  + **Purpose:** Supports complex analysis and decision-making processes, such as market research, sales forecasting, and business intelligence.
  + **Design:** Optimized for read-heavy operations and complex queries, often using a star or snowflake schema.
  + **Data Characteristics:** Data is historical, aggregated, and often stored in a denormalized format for faster querying. The database size is larger, often measured in terabytes.
  + **Users:** Primarily used by data analysts, business intelligence professionals, and executives for strategic analysis.
  + **Queries:** Involves complex, infrequent queries that aggregate and summarize data, such as calculating averages, totals, or trends over time.

**3. Examples of Use Cases:**

* **OLTP:**
  + **Bookstore Example:** Keeping track of book prices, customer transactions, and employee hours.
  + **Retail Example:** Processing sales transactions, updating inventory levels, and managing customer loyalty points.
  + **Banking Example:** Recording deposits and withdrawals, updating account balances, and processing payments.
* **OLAP:**
  + **Bookstore Example:** Analyzing which books have the highest profit margins, identifying the most loyal customers, and determining the employee of the month.
  + **Retail Example:** Analyzing sales trends over different periods, understanding customer buying patterns, and evaluating the effectiveness of marketing campaigns.
  + **Banking Example:** Performing risk analysis, detecting fraudulent activities, and generating financial reports.

**4. Interdependence of OLTP and OLAP:**

OLTP and OLAP systems complement each other and often work in tandem within an organization:

* **Data Flow:** OLTP systems capture and store transactional data, which is periodically extracted, transformed, and loaded (ETL) into OLAP systems.
* **Analysis Feedback:** Insights derived from OLAP systems inform business strategies and operational improvements, which are then implemented in OLTP systems.

**5. Design Considerations:**

* **Understanding Business Requirements:**
  + Before designing a database, it's essential to understand the specific needs and goals of the business. This includes determining what kind of data will be stored, how it will be accessed, and what kind of queries will be performed.
  + Different applications might require different approaches. For example, an e-commerce platform might need both OLTP for transaction processing and OLAP for sales analysis.

**6. Storage Solutions Beyond Traditional Databases:**

* **Data Warehouses:**
  + Optimized for OLAP, data warehouses store large volumes of historical data from various sources. They are designed for query efficiency and analytical performance.
  + Data warehouses often use a denormalized schema to speed up read operations and support complex queries.
* **Data Lakes:**
  + Data lakes can store all types of data, whether structured, semi-structured, or unstructured, at a lower cost.
  + They use a schema-on-read approach, allowing data to be stored in its raw form and interpreted as needed.
  + Data lakes are ideal for big data analytics and support various data processing frameworks like Apache Spark and Hadoop.

**7. Data Modeling:**

* **Conceptual Data Model:**
  + Describes the high-level entities, relationships, and attributes within a database. Tools like entity-relationship diagrams (ERDs) and Unified Modeling Language (UML) diagrams are often used.
* **Logical Data Model:**
  + Defines the structure of the database in terms of tables, columns, and relationships, translating the conceptual model into a format that can be implemented in a DBMS.
* **Physical Data Model:**
  + Describes how the data will be physically stored in the database, including considerations for partitions, indexing, and performance optimization.

**8. Dimensional Modeling for Data Warehouses:**

* **Star Schema:**
  + A simple and commonly used schema in data warehousing, where a central fact table is connected to multiple dimension tables. This schema is easy to understand and perform queries on.
* **Fact Tables:**
  + Contain quantitative data for analysis and are typically connected to multiple dimension tables. Fact tables store metrics and foreign keys linking to dimension tables.
* **Dimension Tables:**
  + Store descriptive attributes related to the facts, such as time, location, or product details. These tables are less frequently updated compared to fact tables.

By understanding these concepts, you can make informed decisions about how to design and implement a database that meets your business needs, ensuring efficient data storage, retrieval, and analysis.