Enterprise Design Patterns:

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. It is a description or template for how to solve a problem that can be used in many different situations.

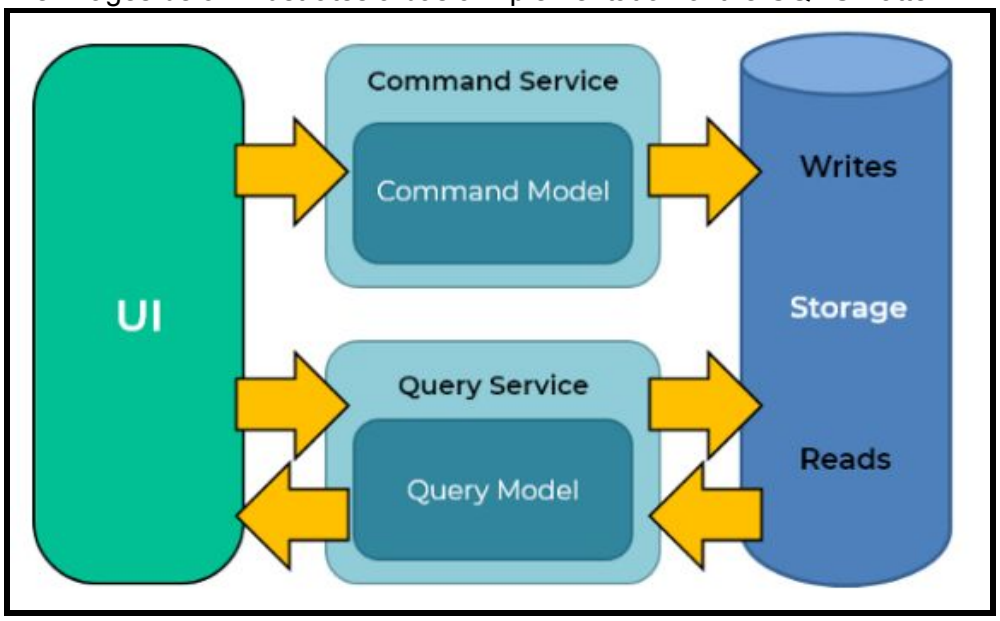
Chain of Responsibility Design Pattern:

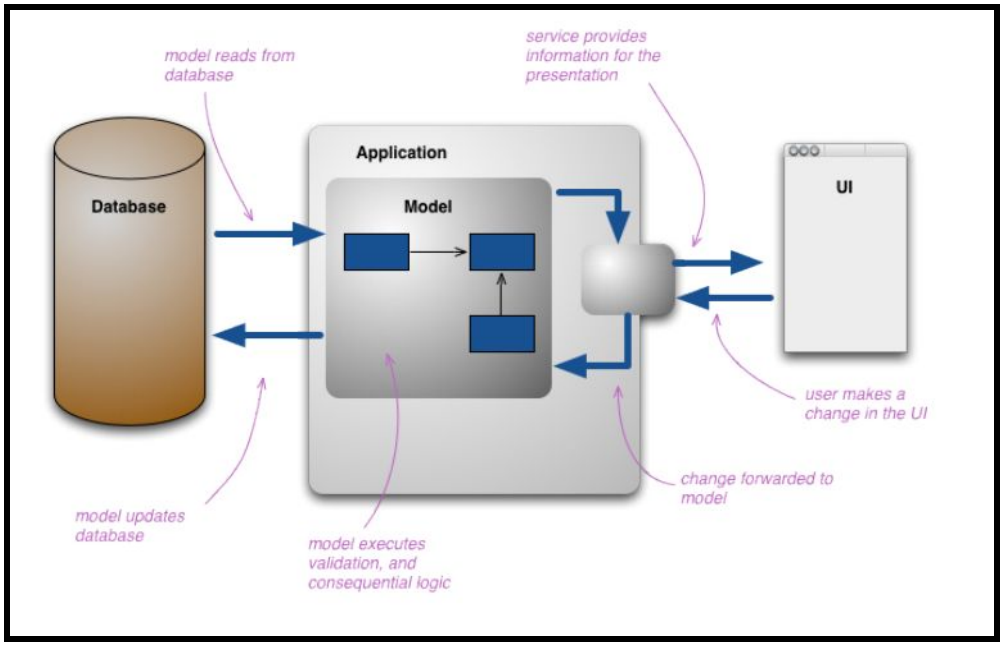
- **Chain of responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- The chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
- Motivation:
    - In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one.
    - Moreover, the object which needs to handle the event might be private/protected.
- Solution:
    - Allow an object to send a command without knowing what object will receive and handle it.
    - The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both.
- Intent:
    - It avoids attaching the sender of a request to its receiver, so other objects can handle it too.
    - The request is sent from one object to another across the chain until one of the objects will handle it.
- Here are a few situations when using chain of responsibility is more effective than other design patterns:
    - More than one object can handle a command.
    - The handler is not known in advance.
    - The handler should be determined automatically.
    - It's wished that the request is addressed to a group of objects without explicitly specifying its receiver.
    - The group of objects that may handle the command must be specified in a dynamic way.

Command Query Responsibility Segregation Pattern:

- **Command and Query Responsibility Segregation (CQRS)** is an architectural pattern where the main focus is to separate the way of reading and writing data. This pattern uses two separate models:
    1. **Queries:** Which are responsible for reading data.
    2. **Commands:** Which are responsible for updating data.
- The CQRS pattern separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security.
- Commands represent the intention of changing the state of an entity. They execute operations like Insert, Update, Delete. Commands objects alter state and do not return data.
- Queries are used to get data from the database. Queries objects only return data and do not make any changes.
- CQRS was first described by Greg Young. It consists of separating the model to update information from the model you use to read information.

- The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore. However, as our needs become more sophisticated we steadily move away from that model. We may want to look at the information in a different way, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places.
- The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively.
- The images below illustrates a basic implementation of the CQRS Pattern:



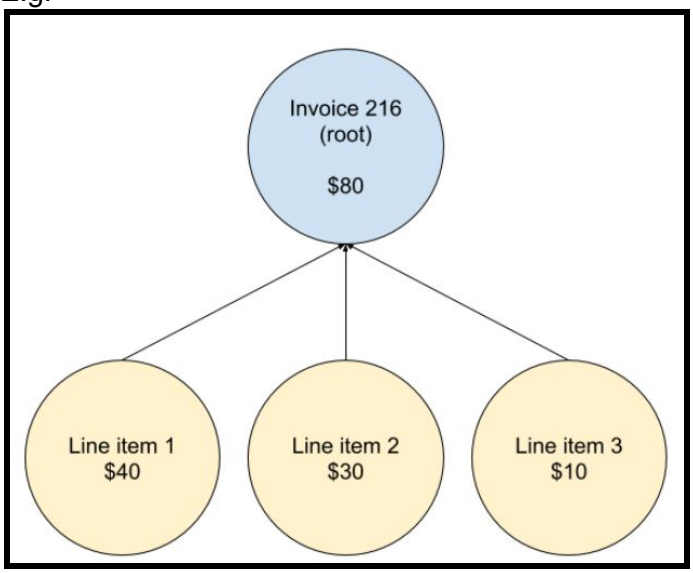


Domain Driven Design Aggregate Pattern:

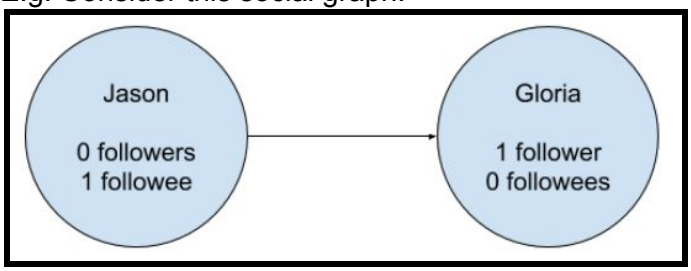
- An **aggregate** is a pattern in Domain-Driven Design (DDD).
- Aggregates consist of one or more entities/objects that change together. We need to treat them as a unit for data changes, and we need to consider the entire aggregate

consistency before we apply changes. Every aggregate must have an **aggregate root** that is the parent of all members of aggregate. Note that it is possible to have an aggregate that consists of one object. In this case, that object would still be the aggregate root.

- A DDD aggregate is a cluster of domain objects that can be treated as a single unit.
- An aggregate will have one of its component objects be the aggregate root.
- The only public member of an aggregate is the aggregate root.
- The root can thus ensure the integrity of the aggregate as a whole.
- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Furthermore, transactions should not cross aggregate boundaries.
- E.g.



- E.g. Consider this social graph:



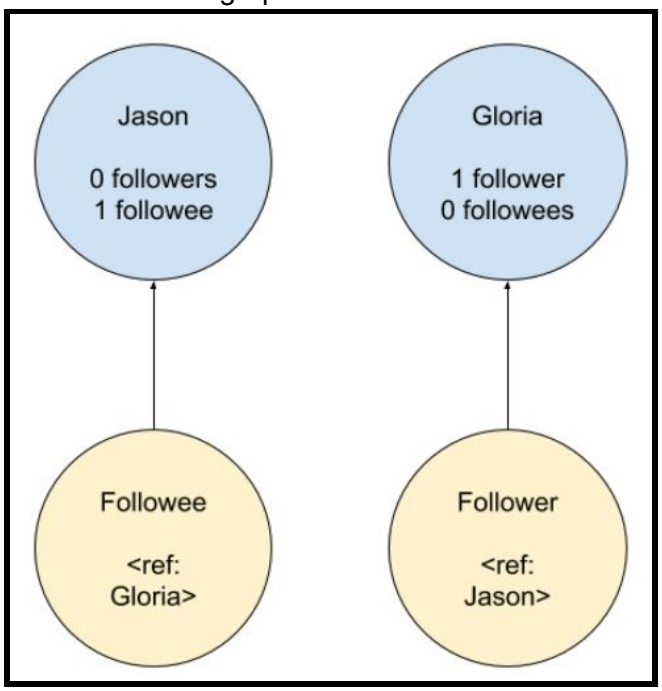
When Jason follows Gloria, we create a Follow relationship between nodes.
Note that, in this design, each person is not an aggregate.
A person doesn't have a consistency boundary. One transaction modifies two persons.

We can model the social graph with aggregates by separating followership from followeeship.
Now, Jason's and Gloria's person objects are both aggregates. They contain private follower and followee objects that contain references to other person objects.
But because there are two different aggregates, we need two different database transactions when Jason follows Gloria:

1. We add a followee, referencing Gloria, to Jason and update Jason's followee count.

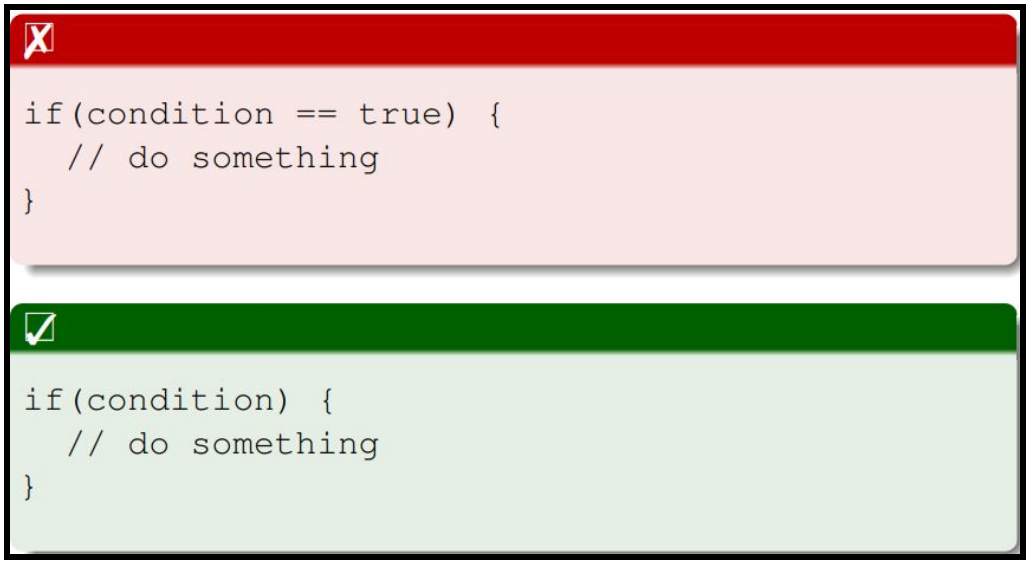
2. We add a follower, referencing Jason, to Gloria and update Gloria's follower count.

Here is what the graph looks like now:



Code Quality:

- **High code quality means that it is:**
  - Correct and efficient
  - Easy to read and understand by other people
  - Easy to test
  - Easy to deploy
  - Easy to extend and maintain
- Good code makes the design clearer.
- Good design makes it easier to code.
- Note that producing code does not equal maintaining code.
- **Maintenance is more important because:**
  - We produce once but need to maintain for a while.
  - Code constantly needs to change and it needs to change without errors or negative effects on existing users.
- **Low quality code has a price:**
  - Low productivity slows down business growth.
  - Degrades customer experience.
  - Makes it hard for the company to attract talent, which leads to even lower quality code.
- The problem is that measuring quality is hard. Quality is an abstract concept making it hard to quantify.
- **Craftsmanship:**
  - Attention to details
  - Continuous refactoring
  - Commitment to quality and consistency
  - Experience

- **Tools:**
  - Automate as much as possible.
- **Communication:**
  - Peer review.
- **Basic Rules:**
  1. Be consistent and follow same style and conventions that are already in the codebase
  2. Be predictable.
     - Follow industry standards.
     - Dont surprise other developers.
  3. Avoid duplication.
     - If you see duplicate code, try to refactor it.
     - If you can't at least don't add to the mess.
- **Comments:**
  - Are "reader's notes" for your code.
  - They communicate extra info with other developers working on the code. E.g. When borrowing code from StackOverflow or other sites include a link in order to provide more context.
  - Avoid comments that are:
    - Wrong
    - Obsolete
    - Redundant
    - Poorly written
  - Avoid commented out code. Other developers will not remove it, and it makes the code messy and hard to read.
  - Sometimes, bad comments may mislead other people.
- **Coding Style:**
  1. 

2.

✗

```
if(someCondition) {
    return true;
}
else {
   return false;
}
```

✓

```
return someCondition;
```

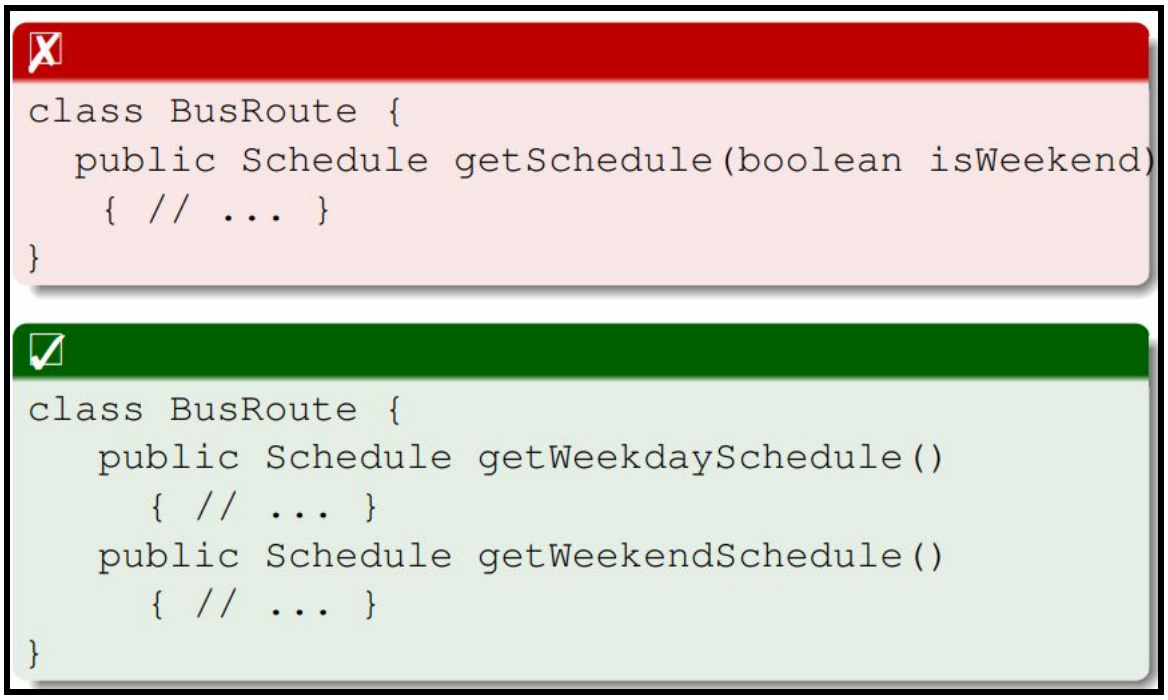
3.

✗

```
boolean done = false;
while(! done) {
    // Do stuff ...
   if(finishedDoingStuff) {
      done = true;
  }
}
```

Instead, define an explicit stopping condition for the while loop. This will make the code cleaner and clearer and avoid unnecessary flag variables.

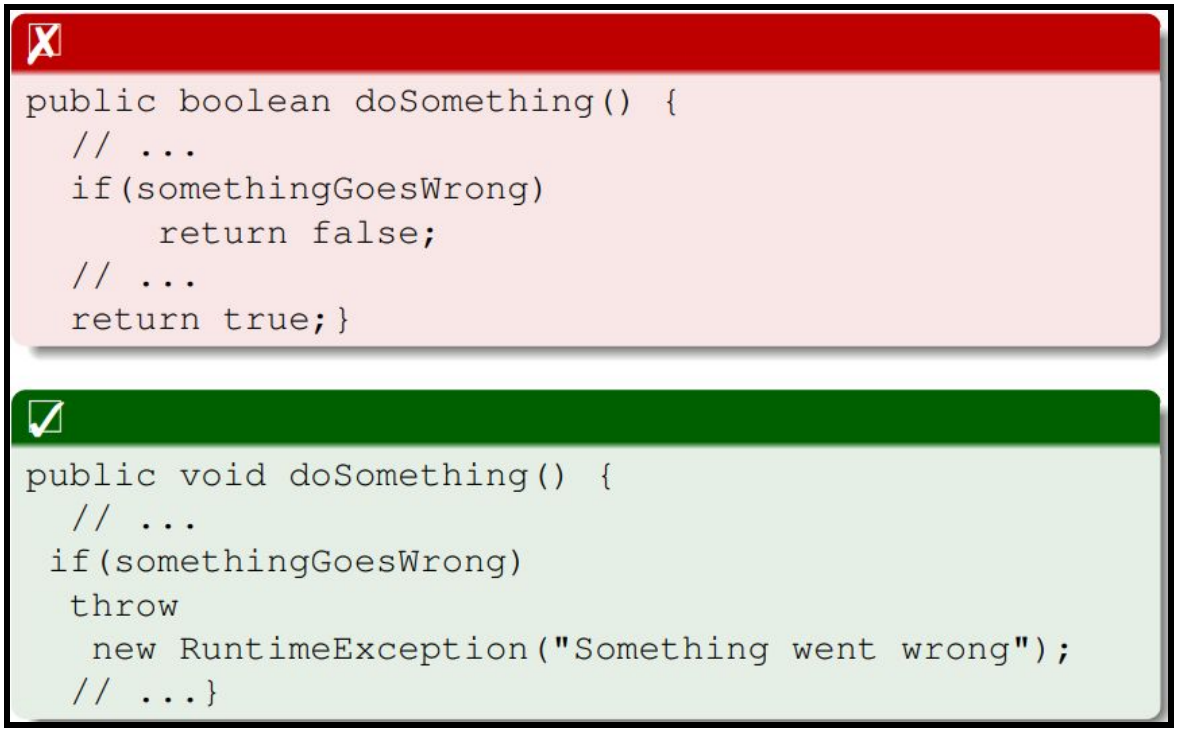
4.



```
class BusRoute {
  public Schedule getSchedule(boolean isWeekend)
   { // ... }
}
```

```
class BusRoute {
   public Schedule getWeekdaySchedule()
     { // ... }
   public Schedule getWeekendSchedule()
     { // ... }
}
```

5.



```
public boolean doSomething() {
  // ...
  if(somethingGoesWrong)
      return false;
  // ...
  return true;}
```

```
public void doSomething() {
  // ...
 if(somethingGoesWrong)
  throw
   new RuntimeException("Something went wrong");
  // ...}
```

- **Variable Names:**
- Use good variable names.
- E.g.

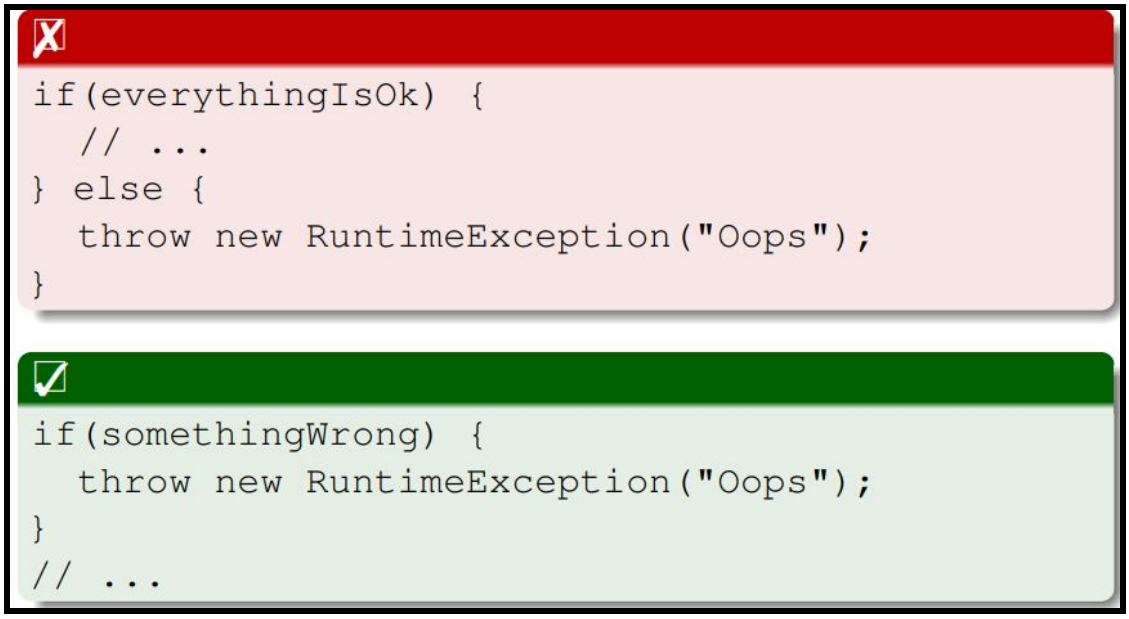
Bad	Good	Notes
List<Double> lst	List<Double> marks	Names should be meaningful
int weight	weightInGrams	Names should be precise
Set<Employee> employeeList	Set<Employee> employeeSet	Names shouldn't be misleading
Timestamp tsMod	Timestamp modificationTime	Names should be pronounceable

- You should generally avoid one-letter names as they have no meaning and are not easily searchable.
  One exception to this rule is loops.
  I.e. **for i in range(0, 10)** or **while i in range(0, 10)** ← It's fine to use the variable "i" here.
  The length of variable name should be proportional to the variable scope.
- Program towards an interface, not towards an implementation, especially important when declaring method arguments.
  I.e.

✗
```
ArrayList<String> words;
```
✓
```
List<String> words;
```

✗
```
public int count(ArrayList<String> words);
```
✓
```
public int count(List<String> words);
```

- **Functions:**
- Functions should do one thing and one thing only.
- Keep functions small.
- Use helper functions.
- Use descriptive names for functions and their arguments.
- Throw early and avoid deeply nested blocks.
  I.e.



- Avoid functions with too many arguments. If your function has too many arguments, it's easy for the caller to get things wrong and harder to test.
- Use standard terminology, avoid slang, and be consistent.
- **Static Factory Methods:**
- A design pattern.
- Wrap constructors in a static method with a meaningful name.
- Helps developers using your class by making their code easier to read and reduces the chance the developers will get it wrong.
- **Cohesion:**
- **Cohesion** refers to the degree to which the elements inside a module belong together.
- High-cohesion means that each class takes care of one thing, and one thing only.
- Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.
- Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.
- Think of building a physical robot. Many small parts (highly cohesive), versus a few "mega parts" (low cohesion, monolithic).
- Cohesive classes have a single responsibility.
- Cohesion is desirable because smaller classes are easier to understand, easier to test and easier to maintain.
- **Coupling:**
- **Coupling** refers to the interdependencies between modules.
  I.e. Components that are mutually dependent are also called coupled.

- A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.
- **Tight coupling/tightly coupled** is a type of coupling that describes a system in which hardware and software are not only linked together, but are also dependent upon each other.
- Loose coupling is important because it enables isolation.
- We want to keep the coupling low and the cohesion high.
- **Code Quality Tools:**
- **Linting** is the process of checking the source code for programmatic as well as stylistic errors. This is most helpful in identifying some common and uncommon mistakes that are made during coding. A **Lint** is a program that supports linting. Examples include Pycharm, Pylint, Lint4j. It can detect suspicious parts of the code such as unused variables and dead code, constant conditions and statements with no effect.
- **Style checkers** can be integrated in your workflow and ensure that your code follows specific styling rules, such as
    - 80 character per line
    - Indentation rules
    - Where to put curly braces
- **Peer Review:**
- Code review is a very common practice.
- Experienced developers can help beginners maintain code quality & consistency.
- More eyes on the code means there is a better chance of detecting problems.
- Relates to agile values like open communication and transparency.