1. Mention the principle that is behind the Agile Manifesto.

Ans: The Twelve Principles are the guiding principles for the methodologies that are included under the title “The Agile Movement.” They describe a culture in which change is welcome, and the customer is the focus of the work.

The twelve principles of agile development include:

1. **Customer satisfaction through early and continuous software delivery:** Customers are happier when they receive working software at regular intervals, rather than waiting extended periods of time between releases.
2. **Accommodate changing requirements throughout the development process:** The ability to avoid delays when a requirement or feature request changes.
3. **Frequent delivery of working software:** Scrum accommodates this principle since the team operates in software sprints or iterations that ensure regular delivery of working software.
4. **Collaboration between the business stakeholders and developers throughout the project:** Better decisions are made when the business and technical team are aligned.
5. **Support, trust, and motivate the people involved:** Motivated teams are more likely to deliver their best work than unhappy teams.
6. **Enable face-to-face interactions:** Communication is more successful when development teams are co-located.
7. **Working software is the primary measure of progress:** Delivering functional software to the customer is the ultimate factor that measures progress.
8. **Agile processes to support a consistent development pace:** Teams establish a repeatable and maintainable speed at which they can deliver working software, and they repeat it with each release.
9. **Attention to technical detail and design enhances agility:** The right skills and good design ensures the team can maintain the pace, constantly improve the product, and sustain change.
10. **Simplicity:** Develop just enough to get the job done for right now.
11. **Self-organizing teams encourage great architectures, requirements, and designs:** Skilled and motivated team members who have decision-making power, take ownership, communicate regularly with other team members, and share ideas that deliver quality products.

12. **Regular reflections on how to become more effective:** Self-improvement, process improvement, advancing skills, and techniques help team members work more efficiently.

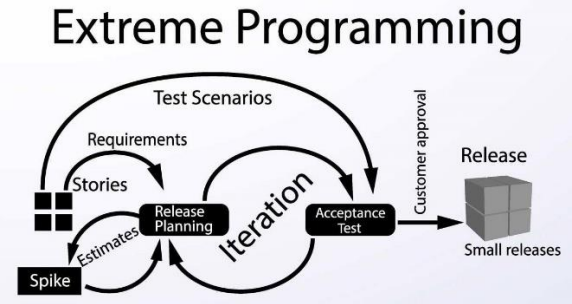
The intention of Agile is to align development with business needs, and the success of Agile is apparent. Agile projects are customer focused and encourage customer guidance and participation. As a result, Agile has grown to be an overarching view of software development throughout the software industry and an industry all by itself.

2. What Is Extreme Programming.

Ans: Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

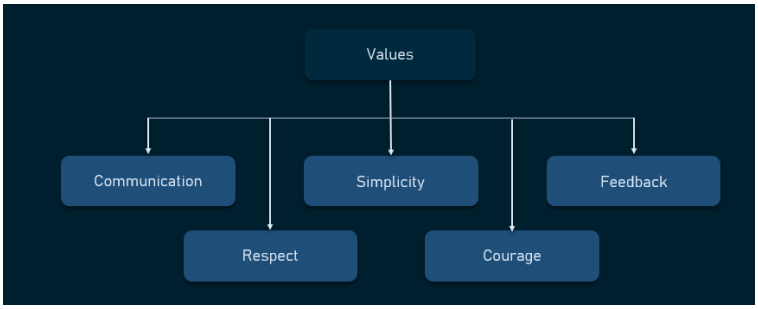
Characteristics:

- Dynamically changing software a requirements.
- Risks caused by fixed time projects using new technology.
- Small, co-located extended development team.
- The technology you are using allows for automated unit and functional tests.



Values:

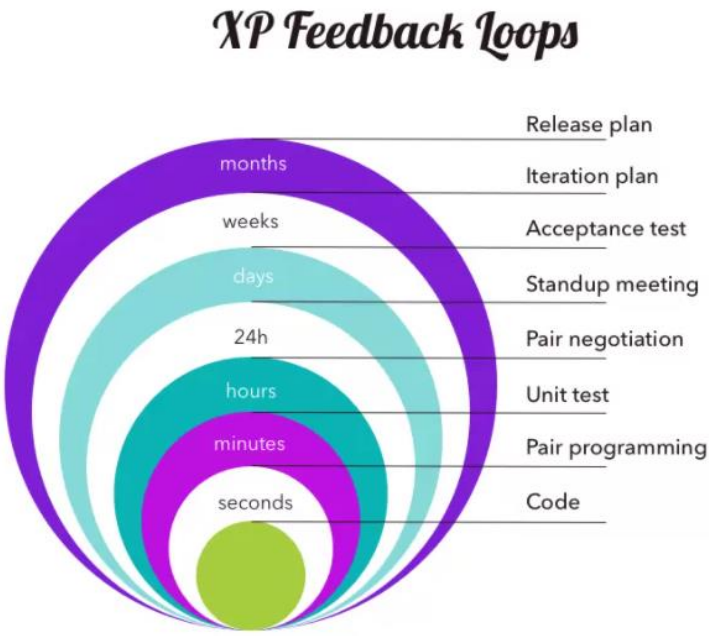
The XP values: communication, simplicity, feedback, courage, and respect. Let's look at each one of them in more detail.



Communication: Lack of communication prevents knowledge from flowing inside a team. Often, when there's a problem, someone already knows how to solve it. But lack of communication prevents them from learning about the problem or contributing to its solution. So, the problem ends up being solved twice, generating waste.

Simplicity: Simplicity says you always strive to do the simplest thing that works. It's often misunderstood and taken as *the simplest thing*, period, ignoring the "that works" part.

Feedback: Feedback in more traditional, waterfall-like software development methodologies often is "too little, too late".



Courage: Kent Beck defines courage as "effective action in the face of fear." As a software engineer, you have plenty to be afraid of and therefore plenty of opportunities to show courage.

Respect: A fundamental premise of XP is that everyone cares about their work. No amount of technical excellence can save a project if there's no care and respect.

Principles:

Principles provide more specific guidance than values. They're guidelines that illuminate the values and make them more explicit and less ambiguous.

Humanity: Humans create software for humans, a fact that's often overlooked. But taking human basic needs, strengths, and weaknesses into account creates products humans *want* to use. And a work environment that gives you the opportunity for accomplishment and growth, the feeling of belonging, and basic safety, is a place where you more easily take others' needs into account.

Economics: In XP, teams heed the economic realities of software development all the time, they constantly assess the economic risks and needs of the project.

.

Mutual Benefit: Following XP, you avoid solutions that benefit one party to the detriment of another. For instance, extensive specifications might help someone else understand it, but it takes you away from implementing it, and it delays it for your users.

Self-similarity: If a given solution works at a level, it might also work at a higher or lower level. For instance, obtaining early and constant feedback is at play at various levels in XP.

- at the developer level, programmers receive feedback from their work using the test-first approach;
- at the team level, the continuous integration pipeline integrates, builds, and tests the code several times a day;
- at the organization level, the weekly and quarterly cycles allow teams to get feedback and improve their work as needed.

Improvement: According to the principle of improvement, teams don't strive for perfection in an initial implementation but for a good enough one, and to then learn and improve it continuously with feedback from real users.

Diversity: You and your coworkers benefit from a diversity of perspectives, skills, and attitudes. Such diversity often leads to conflict, but that's okay.

Reflection: Great teams reflect on their work and analyze how to be better. XP offers plenty of opportunities for that. Not just in its weekly and quarterly cycles, but in every practice it promotes.

Flow: Traditional software development methodologies have discrete phases, which last for a long time and have few feedback and course correction opportunities. Instead, software development in XP occurs in activities that happen all of the time, in a consistent "flow" of value.

Opportunity: Problems are inevitable in software development. However, every problem is an opportunity for improvement. Learn to look at them that way and you're far more likely to come up with creative, goal-oriented solutions that also serve to prevent them from happening again.

Redundancy: The redundancy principle says that if a given problem is critical, you must employ many tactics to counter it.

Failure: Failure isn't waste when it results in knowledge. Acting and quickly learning what doesn't work is way more productive than inaction caused by indecision in choosing between many options.

Quality: Often people think there's a dilemma between quality and speed. It's the opposite: pushing for quality improvements is what makes you go faster.

Baby Steps: Big changes are risky. XP mitigates that risk by doing changes in tiny steps, at all levels.

Accepted Responsibility: In XP, responsibility should be accepted, never assigned.

3. What is Extreme Programming Practices.

Ans: There are four basic activities in Extreme Programming. They are:

- Coding
- Testing
- Listening
- Designing

These four basic activities need to be structured in the light of the Extreme Programming principles. To accomplish this, the Extreme Programming practices are defined.

These 12 Extreme Programming practices achieve the Extreme Programming objective and wherever one of the practices is weak, the strengths of the other practices will make up for it.

Kent Beck, the author of 'Extreme Programming Explained' defined 12 Extreme Programming practices as follows −

- The Planning Game
- Short Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40 hour Week
- On-site Customer
- Coding Standards

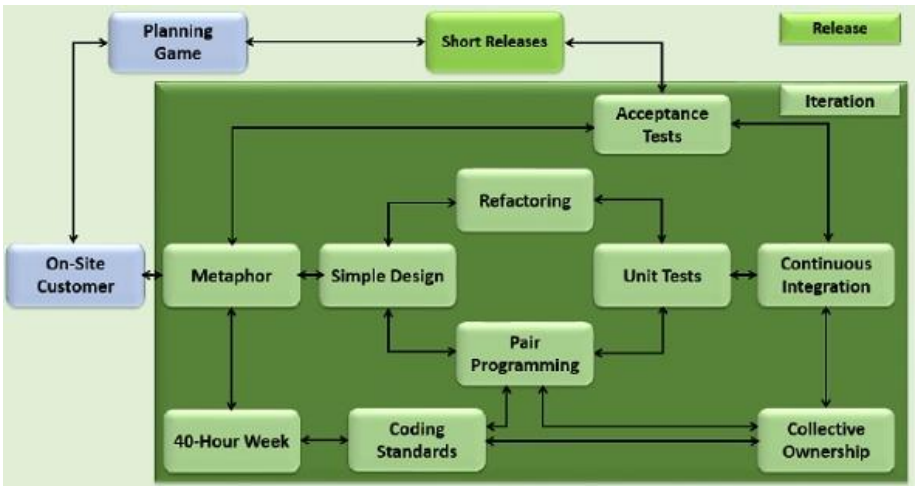
Four Areas of Extreme Programming

The Extreme Programming practices can be grouped into four areas −

- Rapid, Fine Feedback −
    - Testing
    - On-site customer
    - Pair programming

- Continuous Process −
  - Continuous Integration
  - Refactoring
  - Short Releases
- Shared Understanding −
  - The Planning Game
  - Simple Design
  - Metaphor
  - Collective Ownership
  - Coding Standards
- Developer Welfare −
  - Forty-Hour Week

The following diagram shows how Extreme Programming is woven around the Extreme Programming practices −



4. What is SCRUM.

Ans: Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Idea first appeared in a business journal in 1986 (applied to product development management).

- Used in software development and presented in 1995 paper.
- Term is based on rugby term
- Small cross-functional teams

5. Explain the SCRUM Practices.

Ans: There are mainly 4 types of SCRUM Practices, which are given below:

Product and release backlog:

- A list of the features to be implemented in the project (subdivided to next release), ordered by priority
- Can adjust over time as needed, based on feedback
- A product manager is responsible for maintaining

Burn-down chart:

- Make best estimate of time to complete what is currently in the backlog
- Plot the time on a chart
- By studying chart, understand how team functions
- Ensure burndown to 0 at completion date
    - By adjusting what's in the backlog
    - By adjusting the completion date

The sprint Chart:

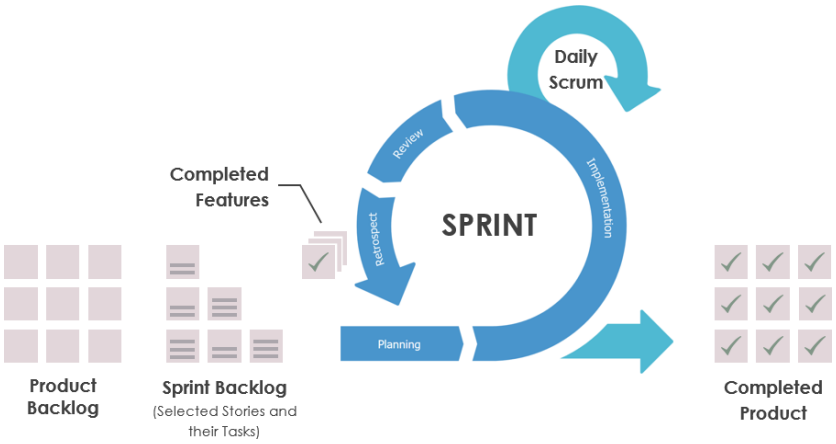
- The sprint is a 1 month period after which some product is delivered
- Features are assigned from the product backlog to a sprint backlog
    - Features divided into smaller tasks for sprint backlog
    - Feature list is fixed for sprint
- Planning meeting
    - Tasks can be assigned to team members
    - Team members have individual estimates of time taken per item
- During sprint, work through features, and keep a burn-down chart for the sprint
- New functionality is produced by the end of the sprint
- After sprint, a review meeting is held to evaluate the sprint.

Scrum meeting:

- 15 minute daily meeting
- All team members show up
- Quickly mention what they did since last Scrum, any obstacles encountered, and what they will do next
- Some team member volunteers or is appointed to be the Scrum Master - in charge of Scrum meeting, and responsible for seeing that issues raised get addressed
- Customers, management encouraged to observe

6. Explain the Sprint in SCRUM Practices.

Ans: The Chart gives a brief idea about the Sprint Chart:



Sprints are the heartbeat of Scrum, where ideas are turned into value.

They are fixed length events of one month or less to create consistency. A new Sprint starts immediately after the conclusion of the previous Sprint.

All the work necessary to achieve the Product Goal, including Sprint Planning, Daily Scrums, Sprint Review, and Sprint Retrospective, happen within Sprints.

During the Sprint:

- No changes are made that would endanger the Sprint Goal
- Quality does not decrease
- The Product Backlog is refined as needed
- Scope may be clarified and renegotiated with the Product Owner as more is learned.

Sprints enable predictability by ensuring inspection and adaptation of progress toward a Product Goal at least every calendar month. When a Sprint's horizon is too long the Sprint Goal may become invalid, complexity may rise, and risk may increase. Shorter Sprints can be employed to generate more learning cycles and limit risk of cost and effort to a smaller time frame. Each Sprint may be considered a short project.

Various practices exist to forecast progress, like burn-downs, burn-ups, or cumulative flows. While proven useful, these do not replace the importance of empiricism. In complex environments, what will happen is unknown. Only what has already happened may be used for forward-looking decision making.

7. What is Spiking, Splitting, and Velocity in Planning?

Ans: The definition of all three planning phases are given below:

Splitting:

For example, “Users can securely transfer money into, out of, and between their accounts.” - big story. Estimating will be hard and probably inaccurate. Further Splitting above story in

- Users can log in.
- Users can log out.
- Users can deposit money into their account.
- Users can withdraw money from their account.
- Users can transfer money from their account to another account.

Velocity:

- If we have an accurate velocity, we can multiply the estimate of any story by the velocity to get the actual time
- For example, our velocity is “2 days per story point,” and we have a story with a relative estimate of four points, then the story should take eight days to implement.
- As the project proceeds, the measure of velocity will become ever more accurate

Spiking:

- Often, it is sufficient to spend a few days prototyping a story or two to get an idea of the team’s velocity. Such a prototype session is called a spike.

8. Explain Release Planning, Iteration Planning, Task planning.

Ans:

Release Planning:

- The developers and customers agree on a date for the first release of the project. This is usually a matter of 2–4 months in the future. The customers pick the stories they want implemented within that release and the rough order in which they want them implemented.
- The release plan can be adjusted as velocity becomes more accurate.

Iteration Planning:

- typically, two weeks long
- the customers choose the stories that they want implemented in the first iteration.
- The order of the stories within the iteration is a technical decision that makes the most technical sense.
- The customers cannot change the stories in the iteration once the iteration has begun - Reordering is possible

- The iteration ends on the specified date, even if all the stories aren't done.
- If the team got 31 story points done last iteration, then they should plan to get 31 story points done in the next. Their velocity is 31 points per iteration.
- If the team gains in expertise and skill, the velocity will rise commensurately. If someone is lost from the team, the velocity will fall. If an architecture evolves that facilitates development, the velocity will rise.

Task Planning:

- The developers break the stories down into development tasks. A task is something that one developer can implement in 4–16 hours. The stories are analyzed, with the customers' help, and the tasks are enumerated as completely as possible.
- A list of the tasks is created on a flip chart, whiteboard, or some other convenient medium. Then, one by one, the developers sign up for the tasks they want to implement.
- Developers, Database Guys, GUI guys (whole Project team) may sign up for respective kind of task.
- Task selection continues until either all tasks are assigned

9. What is Halfway Point in Planning?

Ans:

The Halfway Point:

- Halfway through the iteration, the team holds a meeting.
- At this point, half of the stories scheduled for the iteration should be complete. If aren't complete, then the team tries to reapportion tasks and responsibilities to ensure that all the stories will be complete by the end of the iteration

For Example,

    - Suppose the customers selected 8 Stories
    - 8 Stories Totally 24 story points 42 Tasks
    - Halfway point 12 story points 21 Tasks should be completed
- At the halfway point, we want to see completed stories that represent half the story points for the iteration.

10. Explain Iterating in brief.

Ans:

Iterating:

- Every two weeks, the current iteration ends and the next begins.
- At the end of each iteration, the running executable is demonstrated to the customers.
- The customers are asked to evaluate the look, feel, and performance of the project.

- The customers can measure velocity. They can predict how fast the team is going, and they can schedule high- priority stories early.

11. Explain Test Driven Development in brief.

Ans:

- What if we designed our tests before we designed our programs?
- What if we refused to implement a function in our programs until there was a test that failed because that function wasn't present?
- What if we refused to add even a single line of code to our programs unless there were a test that was failing because of its absence?
- What if we incrementally added functionality to our programs by first writing failing tests that asserted the existence of that functionality, and then made the test pass?
- What effect would this have on the design of the software we were writing? What benefits would we derive from the existence of such a comprehensive bevy of tests?

We can add functions to the program, or change the structure of the program, without fear that we will break something important in the process.

- we are immediately concerned with the interface of the program as well as its function. By writing the test first, we design the software to be conveniently callable.
- we are immediately concerned with the interface of the program as well as its function. By writing the test first, we design the software to be conveniently callable.
- The act of writing tests first forces us to decouple the software!

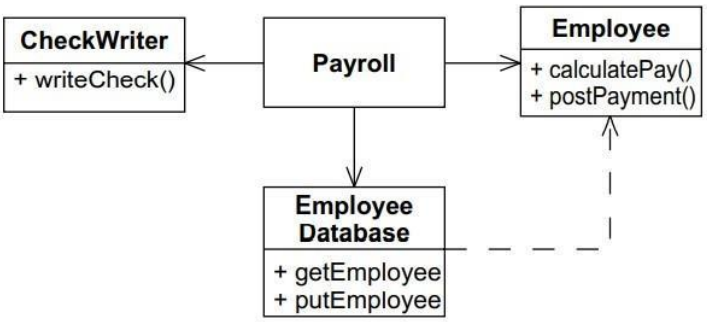
An Example of Test-First Design:

- I trusted that I could make the test pass by writing the code that conformed to the structure implied by the test. This is called intentional programming.
- The point is that the test illuminated a central design issue at a very early stage. The act of writing tests first is an act of discerning between design decisions.
- The test acts as a compile able and executable document that describes the program

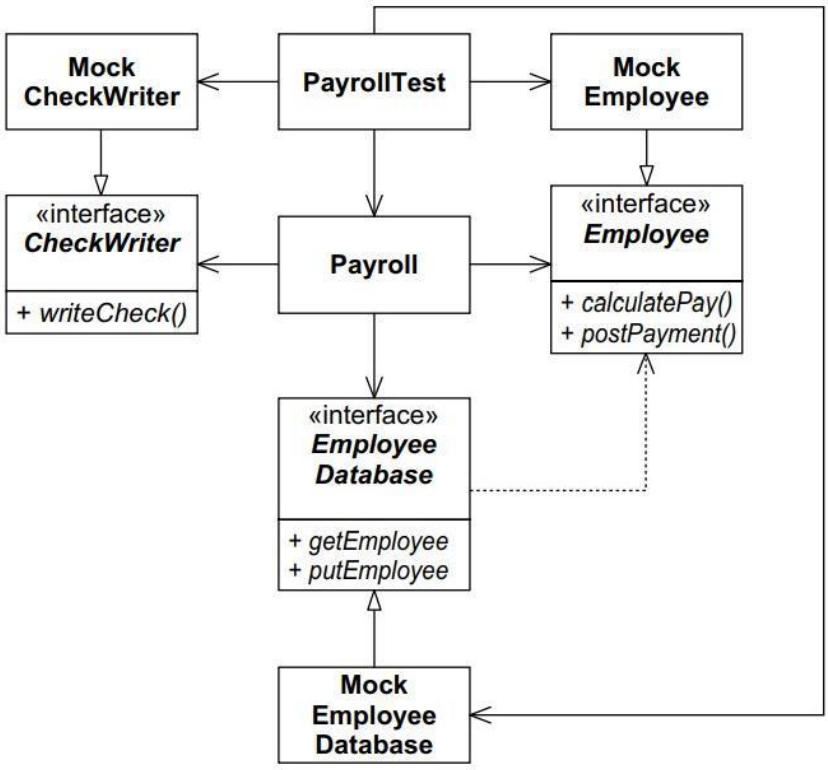
12. Explain Test Isolation in Test Driven Development with appropriate diagram.

Ans: The act of writing tests before production code often exposes areas in the software that ought to be decoupled.

For example, Let's see a simple UML diagram of a payroll application.



- This diagram is just sitting on a whiteboard after a quick design session and that we haven't written any of this code yet.
- There are a number of problem
- First, what database do we use? Payroll needs to read from some kind of database. What data do we load into it?
- Second, how do we verify that the appropriate check got printed? We can't write an automated test that looks on the printer for a check and verifies the amount on it's associated with writing these tests.
- It creates the appropriate mock objects, passes them to the Payroll object, tells the Payroll object to pay all the employees, and then asks the mock objects to verify that all the checks were written correctly and that all the payments were posted correctly.



13. Explain Acceptance Test with example.

Ans:

- Unit tests are necessary but insufficient as verification tools.
- Unit tests verify that the small elements of the system work as they are expected to, but they do not verify that the system works properly as a whole.
- Unit tests are white-box tests that verify the individual mechanisms of the system.
- Acceptance tests are black-box tests that verify that the customer requirements are being met.
- Acceptance tests are written by folks who do not know the internal mechanisms of the system.
- They may be written directly by the customer or by some technical people attached to the customer, possibly QA. Acceptance tests are programs and are therefore executable.
- Acceptance tests are the ultimate documentation of a feature.
- Creating an acceptance testing framework may seem a daunting task.
    - Example of Acceptance Testing
    - Consider, the payroll application
    - we must be able to add and delete employees to and from the database. We must also be able to create paychecks for the employees currently in the database. Fortunately, we only have to deal with salaried employees. The other kinds of employees have been held back until a later iteration.
    - it makes sense that the acceptance tests should be written in simple text files.

AddEmp 1429 “Robert Martin” 3215.88 Payday
Verify Paycheck EmpId 1429 GrossPay 3215.88

- We need to find some common form for those transactions so that the amount of specialized code is kept to a minimum.
- One solution would be to feed the transactions into the payroll application in XML

<AddEmp PayType=Salaried>
<EmpId>1429</EmpId>
<Name>Robert Martin</Name>
<Salary>3215.88</Salary>
</AddEmp>

- We can have the payroll application produce its paychecks in XML

<Paycheck>
<EmpId>1429</EmpId>
<Name>Robert Martin</Name>
<GrossPay>3215.88</GrossPay>
</Paycheck>

14. What is Refactoring and when should it be used?

Ans: Process of changing a computer program's source code without modifying its external functional behavior.

To improve some of the nonfunctional attributes: code readability reduced complexity to improve maintainability better design

- Extensibility

"By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new feature. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code."

Refactoring – When do I use?

- Bad smells!
- A structure in the code suggests, and sometimes even scream for, opportunities for refactoring
- This humorous advice relies on the experience of programmers and on the clarity of code
- Indications to possible bad smells:
- Duplicated code shares one same method
- Long method extracts one or more smaller methods
- Large class divides in more cohesive classes
- Long parameter list encapsulates them
- There is a almost a hundred catalogued refactoring's
    - each one has a name, a motivation, and a mechanics
- They are usually organized into the following groups:
    - Composing Methods (e.g., Extract Method)
    - Moving Features Between Objects (e.g., Move Method)
    - Organizing Data (e.g., Encapsulate Field)
    - Simplifying Conditional Expressions (e.g., Consolidate Conditional Expression)
    - Making Method Calls Simpler (e.g., Rename Method)
    - Dealing with Generalization (e.g., Pull Up Method)
    - Big Refactoring (e.g., Extract Hierarchy)

15. Explain Extract method with appropriate example in Refactoring.

Ans: You have a code fragment that can be grouped together then turn the fragment into a method whose name explains the purpose of the method.

The Example below gives a brief idea about the Extract method.

```
void printOwing ( double amount){
printBanner( );
// print details
System.out.println("name: " + this.name);
System.out.println("amount:" + amount);
}
```

⇓

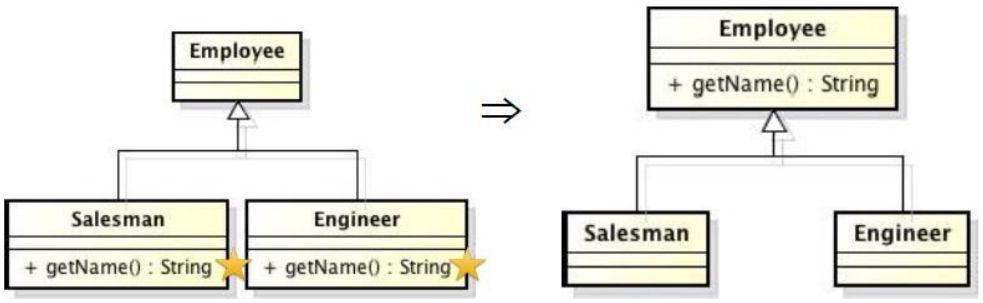
```
void printOwing (double amount) {
printBanner( ) ;
PrintDetails(amount) ;
}

void PrintDetails(double amount)    {
System.out.println("name: " + this.name) ;
System.out.println("amount: " + amount);
}
```

- It is one of the most common refactoring's
- Long methods or look at code that needs a comment to understand its purpose
- So, I turn that fragment of code into its own method.
- Reasons to do that:
    - Increases the chances that other methods can use a method when the method is fine-grained
    - Allows the higher-level methods to read more like a series of comments
    - Overriding also is easier

16. Explain Pull up method with appropriate example in Refactoring.

Ans: You have methods with identical results on subclasses then move them to the superclass.

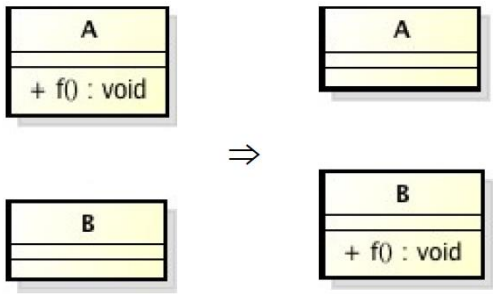


- Eliminating duplicate behavior is important
  - if not, the risk that a change to one not be made to other
- The easiest case: methods have the same body
  - of course, it is not always obvious as that. So, look for the differences and test for safety
- Special case: you have a subclass method that overrides a superclass method yet does the same thing
- The most awkward case: the method may refer to features that are on the subclass but not on the superclass
  - possible solutions: generalize methods, create abstract method on superclass, change a method's signature.

17. Explain Move method with appropriate example in Refactoring.

Ans: A method is, or will be, using or used by more features of another class than the class on which it is defined as Feature Envy.

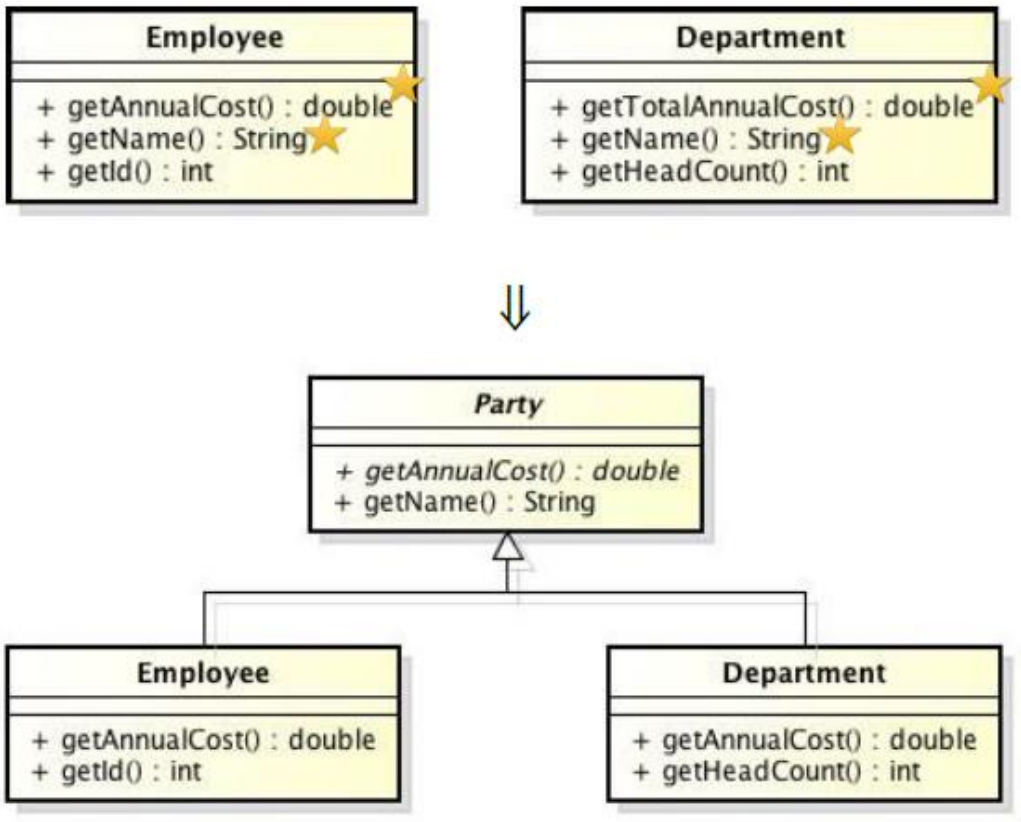
Create a new method with similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it.



- Moving methods is the bread and butter of refactoring
  - classes with too much behavior
  - classes are collaborating too much (too highly coupled)
- By moving methods around:
  - make classes simpler
  - high cohesion (classes end up being a more crisp implementation of a set of responsibilities).

18. Explain Extract Super Class method with appropriate example in Refactoring.

Ans: You have two classes with similar features then create a superclass and move the common features to the superclass.



- Duplicate code is one of the principal bad things in systems
- Duplicate code:
  - two classes that do similar things in the same way or two classes that do similar things in different ways
- A well-known solution: inheritance (everything)
  - However, you often do not notice the commonalities until you have created some classes
  - In this case, you need to create the inheritance structure later.