1. **How to achieve nearly zero bugs in agile XP?**

No Bugs

How to Achieve Nearly Zero Bugs

Many approaches to improving software quality revolve around finding and removing more defects through traditional testing, inspection, and automated analysis.

To achieve faster results, XP uses a potent cocktail of techniques:

1. Write fewer bugs by using a wide variety of technical and organizational practices.
2. Eliminate bug breeding grounds by refactoring poorly designed code.
3. Fix bugs quickly to reduce their impact, write tests to prevent them from reoccurring, then fix the associated design flaws that are likely to breed more bugs.
4. Test your process by using exploratory testing to expose systemic problems and hidden assumptions.
5. Fix your process by uncovering categories of mistakes and making those mistakes impossible.

This may seem like a lot to do, but most of it comes naturally as part of the XP process. Most of these activities improve productivity by increasing code quality or by removing obstacles.

***Tumko kitna thokna hai ingredients se thoko bas upar ka theek se likho.**

Ingredient #1: Write Fewer Bugs

Start with test-driven development (TDD), which is a proven technique for reducing the number of defects you generate . It leads to a comprehensive suite of unit and integration tests, and it structures your work into small, easily verifiable steps.

Teams using TDD report that they rarely need to use a debugger.

To enhance the benefits of test-driven development, work sensible hours and program all production code in pairs. This improves your brainpower, which helps you make fewer mistakes and allows you to see mistakes more quickly.

Pair programming also provides positive peer pressure, which helps you maintain the self-discipline you need to follow defect-reduction practices.

Test-driven development helps you eliminate coding defects, but code isn't your only source of defects.

You can also produce good code that does the wrong thing. To prevent these requirements-oriented defects, work closely with your stakeholders. Enlist on-site customers to sit with your team.

Use customer tests to help communicate complicated domain rules. Have testers work with customers to find and fix gaps in their approach to requirements.

Demonstrate your software to stakeholders every week, and act on their feedback.

Supplement these practices with good coding standards and a "done done" checklist. These will help you remember and avoid common mistakes.

Ingredient #2: Eliminate Bug Breeding Grounds

Writing fewer bugs is an important first step to reducing the number of defects your team generates.

Even with test-driven development, your software will accumulate technical debt over time.

Most of it will be in your design, making your code defect-prone and difficult to change, and it will tend to congregate in specific parts of the system.

These design flaws are unavoidable. Sometimes a design that looks good when you first create it won't hold up over time. Sometimes a shortcut that seems like an acceptable compromise will come back to bite you.

Sometimes your requirements change and your design will need to change as well.

Whatever its cause, technical debt leads to complicated, confusing code that's hard to get right. It breeds bugs.

To generate fewer defects, pay down your debt.

Although you could dedicate a week or two to fixing these problems, the best way to pay off your debt is to make small improvements every week. Keep new code clean by creating simple designs and refactoring as you go.

Ingredient #3: Fix Bugs Now

Programmers know that the longer you wait to fix a bug, the more it costs to fix. In addition, unfixed bugs probably indicate further problems. Each bug is the result of a flaw in your system that's likely to breed more mistakes. Fix it now and you'll improve both quality and productivity.

To fix the bug, start by writing an automated test that demonstrates the bug. It could be a unit test, integration test, or customer test, depending on what kind of defect you've found. Once you have a failing test, fix the bug. Get a green bar.

Don't congratulate yourself yet—you've fixed the problem, but you haven't solved the underlying cause. Why did that bug occur? Discuss the code with your pairing partner. Is there a design flaw that made this bug possible? Can you change an API to make such bugs more obvious? Is there some way to refactor the code that would make this kind of bug less likely?

Improve your design. If you've identified a systemic problem, discuss it with the rest of your team in your next stand-up meeting or iteration retrospective.

Tell people what went wrong so they can avoid that mistake in the future.

Fixing bugs quickly requires the whole team to participate. Programmers, use **collective code ownership** so that any pair can fix a buggy module. Customers and testers, personally bring new bugs to the attention of a programmer and help him reproduce it. These actions are easiest when the whole team sits together.

Ingredient #4: Test Your Process

An exploratory tester uses her intuition and experience to tell her what kinds of problems programmers and customers have the most trouble considering. For example, she might unplug her network cable in the middle of an operation or perform a SQL injection attack on your database.

If your team has typical adversarial relationships between programmers, customers, and testers, these sorts of unfair tests might elicit bitter griping from programmers. The testers expose holes in your thought process and, by doing so, save you from having to make uncomfortable apologies to stakeholders or from dramatic failures in production. Exploratory testing is a very effective way of finding unexpected bugs. It's so effective that the rest of the team might start to get a little lazy. Don't rely on exploratory testing to find bugs in your software.

Your primary defense against bugs is test-driven development. **Use exploratory testing** to test your process. When an exploratory test finds a bug, it's a sign that your work habits—your process—have a hole in them. Fix the bug, then fix your process.

Testers, only conduct exploratory testing on stories that the team agrees are "done done."

Programmers and customers, if your testers find any problems, think of them as bugs. Take steps to prevent them from occurring, just as you would for any other bug. Aim for a defect rate of under one or two bugs per month includingbugs found in exploratory testing.

A good exploratory tester will find more bugs than you expect. To make the bug rate go down, fix your process.

Ingredient #5: Fix Your Process

When the number of bugs you generate gets low enough, you can do something usually associated with NASA's Space Shuttle software: root-cause analysis and process improvement on every bug.

When you fix a bug, start by writing an automated test and improving your design to make the bug less likely. This is the beginning of root-cause analysis, but you can go even further.

As you write the test and fix the design, ask questions. Why was there no test preventing this bug? Why does the design need fixing? Use the "five whys" technique to consider the root cause. Then, as a team, discuss possible root causes and decide how best to change your work habits to make that kind of bug more difficult.

Results

When you produce nearly zero bugs, you are confident in the quality of your software. You're comfortable releasing your software to production without further testing at the end of any iteration. Stakeholders, customers, and users rarely encounter unpleasant surprises, and you spend your time producing great software instead of fighting fires.

2. **List and explain any 10 terminologies used by different version control system.**

VERSION CONTROL TERMINOLOGY

Repository: The repository is the master storage for all your files and and their history. It's typically stored on the version control server. Each standalone project should have its own repository.

Sandbox: Also known as a working copy, a sandbox is what team members work out of on their local development machines.

Check out: To create a sandbox, check out a copy of the repository. In some version control systems, this term means "update and lock."

Update: Update your sandbox to get the latest changes from the repository. You can also update to a particular point in the past.

Lock: A lock prevents anybody from editing a file but you.

Check in or commit : Check in the files in your sandbox to save them into the repository.

Revert: Revert your sandbox to throw away your changes and return to the point of your last update. This is handy when you've broken your local build and can't figure out how to get it working again. Sometimes reverting is faster than debugging, especially if you have checked in recently.

Tip or head: The tip or head of the repository contains the latest changes that have been checked in. When you update your sandbox, you get the files at the tip. (This changes somewhat when you use branches.)

Tag or label: A tag or label marks a particular time in the history of the repository, allowing you to easily access it again.

Roll back: Roll back a check-in to remove it from the tip of the repository. The mechanism for doing so varies depending on the version control system you use.

Branch: A branch occurs when you split the repository into distinct "alternate histories," a process known as branching. All the files exist in each branch, and you can edit files in one branch independently of all other branches.

Merge: A merge is the process of combining multiple changes and resolving any conflicts. If two programmers change a file separately and both check it in, the second programmer will need to merge in the first person's changes.

3. Explain continuous integration and how to practice continuous integration.

Continuous Integration

Continuous integration is a better approach. It keeps everybody's code integrated and builds release infrastructure along with the rest of the application. The ultimate goal of continuous integration is to be able to deploy all but the last few hours of work at any time.

Practically speaking, you won't actually release software in the middle of an iteration. Stories will be half-done and features will be incomplete. The point is to be technologically ready to release even if you're not functionally ready to release.

How to Practice Continuous Integration

In order to be ready to deploy all but the last few hours of work, your team needs to do two things:

1. Integrate your code every few hours.
2. Keep your build, tests, and other release infrastructure up-to-date.

To integrate, update your sandbox with the latest code from the repository, make sure everything builds, then commit your code back to the repository. You can integrate any time you have a successful build. I integrate whenever I make a significant change to the code or create something I think the rest of the team will want right away. Each integration should get as close to a real release as possible. Some teams that

use continuous integration automatically burn an installation CD every time they integrate. Others create a disk image or, for network-deployed products, automatically deploy to staging servers.

4. **Explain the term collective code ownership**

Collective Code Ownership

Collective code ownerships preads responsibility for maintaining the code to all the programmers. Collective code ownership is exactly what it sounds like: everyone shares responsibility for the quality of the code. No single person claims ownership over any part of the system, and anyone can make any necessary changes anywhere. In fact, improved code quality may be the most important part of collective code ownership.

Collective ownership allows—no, expects—everyone to fix problems they find.

If you encounter duplication, unclear names, or even poorly designed code, it doesn't matter who wrote it. It's your code. Fix it!

Making Collective Ownership Work

Collective code ownership requires letting go of a little bit of ego. Rather than taking pride in your code, take pride in your team's code. Rather than complaining when someone edits your code, enjoy how the code improves when you're not working on it. Rather than pushing your personal design vision, discuss design possibilities with the other programmers and agree on a shared solution. Collective ownership requires a joint commitment from team members to produce good code. When you see a problem, fix it. When writing new code, don't do a half-hearted job and assume somebody else will fix your mistakes. Write the best code you can.

5. **Write a brief note on**
   a. **Work-in progress documentation**
   b. **Product documentation**
   c. **Handoff documentation**

A.

Work-In-Progress Documentation

XP teams also use test-driven development to create a comprehensive test suite. When done well, this *captures and communicates details about implementation decisions as unambiguous, executable design specifications that are readable, runnable, and modifiable by other developers.*

Similarly, the team uses customer testing to communicate information about hard to-understand domain details.

The team does *document some things, such as the vision statement and story cards*, but these act more as reminders than as formal documentation. At any time, *the team can and should jot down notes that help them do their work, such as design sketches on a whiteboard, details on a story card, or hard-to-remember requirements in a wiki or spread sheet.*

B.

Product Documentation

Some projects need to produce specific kinds of documentation to provide business value. Examples include user manuals, comprehensive API reference documentation, and reports.

One team I worked with created code coverage metrics—not because they needed them, but because senior management wanted the report to see if XP would increase the amount of unit testing.

Because this documentation carries measurable business value but isn't otherwise necessary for the team to do its work, schedule it in the same way as all customer-valued work: with a story.

Create, estimate, and prioritize stories for product documentation just as you would any other story

C.

Handoff Documentation

If you're setting the code aside or preparing to hand off the project to another team, *create a small set of documents recording big decisions and information.*

Your *goal is to summarize the most important information you've learned while creating the software* —the kind of information necessary to sustain and maintain the project.

Error conditions are important. What can go wrong, when might it occur, and what are the possible remedies?

Are there any traps or sections of the code where the most straightforward approach was inappropriate?

Do certain situations reoccur and need special treatment?

This is all information you've discovered through development as you've learned from writing the code. In clear written form, this information helps mitigate the risk of handing the code to a fresh group

Results

When you communicate in the appropriate ways, you spread necessary information effectively. You reduce the amount of overhead in communication. You mitigate risk by presenting only necessary information.

Unit 4

1. **What is product vision, explain in brief?**
   i. **Documenting the Vision**
   ii. **How to Create a Vision Statement**

The Product Vision captures the shared understanding of the goal that is to be achieved with building the product. It is created by the Product Owner.

i.

Documenting the Vision

After you've worked with visionaries to create a cohesive vision, document it in a vision statement. It's best to do this collaboratively, as doing so will reveal areas of disagreement and confusion. Without a vision statement, it's all too easy to gloss over disagreements and end up with an unsatisfactory product.

Once created, the vision statement will help you maintain and promote the vision. It will act as a vehicle for discussions about the vision and a touch point to remind stakeholders why the project is valuable.

The vision statement should be a living document: the product manager should review it on a regular basis and make improvements. However, as a fundamental statement of the project's purpose, it may not change much.

ii.

How to Create a Vision Statement

The vision statement documents three things:

What the project should accomplish,

Why it is valuable, and

The project's success criteria

The vision statement is a clear and simple way of describing why the project deserves to exist.

In the first section—what the project should accomplish—describe the problem or opportunity that the project will address, expressed as an end result. Be specific, but not prescriptive. Leave room for the team to work out the details.

Here is a real vision statement describing "Sasquatch," a product developed by two entrepreneurs who started a new company:

Sasquatch helps teams collaborate over long distance. It enables the high-quality team dynamics that occur when teams gather around a table and use index cards to brainstorm, prioritize, and reflect.

2. **How to release frequently, explain with an example?**

How to Release Frequently

Releasing frequently doesn't mean setting aggressive deadlines. . In fact, aggressive deadlines extends schedules rather than reducing them. Minimum marketable features are an excellent tool for doing so. A minimum marketable feature, or MMF, is the smallest set of functionality that provides value to your market, whether that market is internal users (as with custom software) or external customers (as with commercial software).

MMFs provide value in many ways, such as competitive differentiation, revenue generation, and cost savings.

As you create your release plan, think in terms of stakeholder value. Sometimes it's helpful to think of stories and how they make up a single MMF. Don't forget the minimum part of minimum

marketable feature—try to make each feature as small as possible.

Once you have minimal features, group them into possible releases. This is a brainstorming exercise, not your final plan, so try a variety of groupings. Think of ways to minimize the number of features needed in each release.

The most difficult part of this exercise is figuring out how to make small releases.

An Example: Imagine you're the product manager for a team that's creating a new word processor. The market for word processors is quite mature, so it might seem impossible to create a small first release. There's so much to do just to match the competition, let alone to provide something new and compelling. You need basic formatting, spellchecking, grammar checking, tables, images, printing... the list goes on forever.

Release those features first—they probably have the most value.

Suppose that the competitive differentiation for your word processor is its powerful collaboration capabilities and web-based hosting. The first release might have four features: basic formatting, printing, web-based hosting, and collaboration. You could post this first release as a technical preview to start generating buzz. Later releases could improve on the base features and justify charging a fee: tables, images, and lists in one release, spellchecking and grammar checking in another, and so on.

If this seems foolish, consider Writely, the online word processing application. It doesn't have the breadth of features that Microsoft Word does, and it probably won't for many years. Instead, it focuses on what sets it apart: collaboration, remote document editing, secure online storage, and ease of use.

3. **Explain the bellow terms.**
   a. **Generic risk-management plan**
   b. **Project-specific risk**

A.

A Generic Risk-Management Plan

Every project faces a set of common risks: turnover, new requirements, work disruption, and so forth. These risks act as a multiplier on your estimates, doubling or tripling the amount of time it takes to finish your work.

Provided some generic risk multipliers These multipliers show your chances of meeting

various schedules. For example, in a "Risky" approach, you have a 10 percent chance of

finishing according to your estimated schedule. Doubling your estimates gives you a 50 percent chance of on-time completion, and to be virtually certain of meeting your schedule, you have to quadruple your estimates.

B.

Project-Specific Risks:

The generic risk multipliers include the normal risks of a flawed release plan, ordinary requirements growth, and employee turnover. In addition to these risks, you probably face some that are specific to your project. To manage these, create a risk census—that is, a list of the risks your project faces that focuses on your project's unique risks. suggest starting work on your census by brainstorming catastrophes. Gather the whole team and hand out index cards. Remind team members that during this exercise, negative thinking is not only OK, it's necessary. Ask them to consider ways in which the project could fail. Write several questions on the board:

4. **Explain the stories and story cards**

Story Cards

Write stories on index cards.

During release planning, customers and stakeholders gather around a big table to select stories for the next release. It's a difficult process of balancing competing desires. Index cards help prevent these disputes by visually showing priorities, making the scope of the work more clear, and directing conflicts toward the plan rather than toward personalities.

Story cards also form an essential part of an informative workspace. After the planning meeting, move the cards to the release planning board—a big, sixfoot whiteboard, placed prominently in the team's open workspace. You can post hundreds of cards and still see them all clearly. For each iteration, place the story cards to finish during the iteration on the iteration planning board.

Index cards also help you be responsive to stakeholders. When you talk with a stakeholder and she has a suggestion, invite her to write it down on a blank index card. Afterward, take the stakeholder and her card to the product manager. They can walk over to the release planning board and discuss the story's place in the overall vision. Again, physical cards focus the discussion on relative priorities rather than on contentious "approved/disapproved" decisions.

If the product manager and stakeholder decide to add the story to the release plan, they can take it to the programmers right away. A brief discussion allows the programmers to estimate the story. Developers write their estimate—and possibly a few notes—on the card, and then the stakeholder and product manager place the card into the release plan.

Physical index cards enable these ways of working in a very easy and natural way that's surprisingly difficult to replicate with computerized tools.

5. Explain the planning game.

The Planning Game

Audience
Whole Team

Our plans take advantage of both business and technology expertise.

You may know when and what to release, but how do you actually construct your release plan? That's where *the planning game* comes in.

In economics, a *game* is something in which "players select actions and the payoffs depend on the actions of all players."* The study of these games "deals with strategies for maximizing gains and minimizing losses... [and are] widely applied in the solution of various decision making problems."†

That describes the planning game perfectly. It's a structured approach to creating the best possible plan given the information available.

The planning game is most notable for the way it maximizes the amount of information contributed to the plan. It is strikingly effective. Although it has limitations, if you work within them, I know of no better way to plan.