unit3-chap7

Chapter 7 – Releasing (XP)

Big Idea of Chapter 7

> **Releasing is not a phase at the end.**
> **Releasing is a discipline practiced every day.**

XP assumes:

- Software is always **close to releasable**
- Quality is built **continuously**, not inspected later

PART 1: "DONE DONE"

What Does "Done Done" Mean?

In XP, **"Done" is not enough**.

> **"Done Done" means production-ready software.**

Not Done

- Code compiles
- Feature “mostly works”
- Tests written later
- Bugs deferred

Done Done

- Code written
- Tests written & passing
- Integrated into main codebase
- Ready to deploy **right now**

Production-Ready Software

Production-ready software means:

- No known bugs
- Fully tested
- Meets customer acceptance criteria
- Can be released without panic

Example (College Project)

- “Login works on my laptop”
- “Login works, tests pass, integrated, demo-ready”

XP demands the second.

How to Be "Done Done"

XP teams achieve "Done Done" by:

- Writing tests first
- Integrating continuously
- Fixing defects immediately
- Never postponing quality

  Quality is not negotiable. Scope is.

Making Time (Important Mindset Shift)

Common excuse:

> "We don't have time to do it properly."

XP response:

> "You don't have time *not* to."

Example:

- Fixing a bug now: 10 minutes
- Fixing the same bug after release: hours or days

"Done Done"

- Can we release this today?
- Are all tests passing?
- Are there known defects?

If **any answer is NO → not done done**.

Results of "Done Done"

- Predictable releases
- Low stress
- High trust
- Fast feedback

Contraindications

"Done Done" is difficult when:

- Manual testing dominates
- No automation
- Heavy legacy code
- Management rewards speed over quality

Alternatives

If "Done Done" is impossible:

- Gradually strengthen definition of done
- Start with critical features
- Invest in automation first

PART 2: NO BUGS

"No Bugs" — What XP Really Means

XP does **not** mean:

> "Bugs never happen"

XP means:

> **"Bugs are not allowed to accumulate."**

How Is This Possible?

Traditional teams:

- Collect bugs
- Create bug databases
- Schedule "bug-fix phases"

XP teams:

- Fix bugs immediately
- Treat bugs as emergencies
- Prevent bug creation

How to Achieve Nearly Zero Bugs

Follow these **5 ingredients**.

Ingredient #1: Write Fewer Bugs

How?

- Pair programming
- TDD
- Simple design

Example:

- Two people catch mistakes one person misses

Ingredient #2: Eliminate Bug Breeding Grounds

Bug breeding grounds:

- Complex code
- Duplication
- Poor design

XP response:

- Continuous refactoring
- Clean code
- Simple architecture

Ingredient #3: Fix Bugs Now

XP rule:

> **Never postpone a bug fix.**

Example:

- Bug found at 11 AM → fixed by 11:30 AM
- No "we'll fix it later"

Reason:

- Context is fresh
- Cost is minimal

Ingredient #4: Test Your Process

XP teams ask:

- Why did this bug escape?
- Which test was missing?

Tests evolve **with mistakes**.

Ingredient #5: Fix Your Process

If bugs recur:

- Improve practices
- Improve communication
- Improve tests

  XP fixes systems, not people.

Invert Your Expectations

Traditional thinking:

- Bugs are normal
- Quality improves later

XP thinking:

> Bugs are **abnormal** and indicate process failure.

"No Bugs" — Questions

- Are we hiding bugs?
- Are we normalizing defects?
- Are we fixing root causes?

Results

- Stable codebase
- Predictable delivery
- High customer confidence

Contraindications

"No bugs" is hard when:

- Manual regression testing
- Poor test coverage
- Large untested legacy systems

PART 3: VERSION CONTROL

Why Version Control Is Essential

XP teams:

- Integrate frequently
- Work in pairs
- Change code constantly

Without version control → chaos.

Concurrent Editing

XP allows:

- Many people editing same code
- Frequent commits
- Fast feedback

Tools:

- Git
- SVN

Time Travel

Version control allows:

- Rollback
- Blame (for learning, not punishment)
- Experiment safely

Whole Project in Version Control

XP rule:

> Everything goes into version control.

Includes:

- Code
- Tests
- Build scripts
- Documentation

Customers and Version Control

Customers:

- Don't commit code
- But can **see progress**
- May access demos or tagged releases

Keep It Clean

XP practices:

- Small commits
- Meaningful messages
- No broken builds

Single Codebase

XP strongly prefers:

> **One shared codebase**

Reason:

- Avoid merge hell
- Avoid divergence
- Encourage collaboration

Appropriate Uses of Branches

Branches are allowed for:

- Experiments
- Emergency fixes

But:

- Short-lived
- Merged quickly

PART 4: TEN-MINUTE BUILD

Ten-Minute Build — Meaning

XP rule:

> **The full build must complete in 10 minutes or less.**

Why?

- Developers run it often
- Fast feedback

- Less waiting

Automate Your Build

Manual builds:

- Error-prone
- Slow
- Inconsistent

XP demands:

- Fully automated build

How to Automate

Include:

- Compilation
- Tests
- Packaging
- Reports

Triggered by:

- Every commit

When to Automate

XP answer:

> **Immediately.**

Automation is not optional.

Automating Legacy Projects

Strategies:

- Automate gradually
- Start with critical paths
- Improve step by step

Ten Minutes or Less — Why Strict?

If build takes:

- 30 minutes → people skip it
- 1 hour → integration delayed

XP belief:

> Fast builds encourage discipline.

Results

- Continuous integration
- Early defect detection
- High confidence

Contraindications

- Extremely large systems
- Heavy integration tests

Alternatives

- Split builds
- Parallel execution
- Fast smoke tests + slower nightly builds

SUMMARY

- "Done Done" ensures releasability
- No bugs = no bug accumulation
- Version control enables collaboration
- Ten-minute build ensures fast feedback
- Releasing is a **daily discipline**

XP does not prepare for release.
XP lives in a releasable state.

Part 5: Continuous Integration (XP)

Continuous Integration – Core Idea

- Continuous Integration (CI) is a key XP practice
- Code is integrated frequently into a shared codebase
- Integration happens every few hours

Key belief:
Integration problems should be found early, not at the end

What Is Continuous Integration?

Definition:
Continuous Integration is the practice of frequently merging code changes into a shared repository and running an automated build and tests each time.

Why Continuous Integration Is Needed

- Avoids last-minute integration problems
- Reduces merge conflicts
- Detects defects early
- Improves team confidence

Analogy:
Daily cleaning vs cleaning once a month

Traditional Integration vs Continuous Integration

Traditional Integration

- Developers work separately for days/weeks
- Integration happens late
- High risk of failure

Continuous Integration

- Small changes integrated often
- Low risk
- Problems fixed quickly

How Continuous Integration Works

- Developer completes a small change
- Runs tests locally
- Commits code to shared repository
- Automated build runs
- Team gets immediate feedback

Integrate Frequently

- Integration every few hours
- Small changes only
- No long-lived branches

XP Rule:
If you wait too long, integration becomes risky

What Is a Build?

A build includes:

- Compiling the code
- Running automated tests
- Producing deployable software

What Does "Breaking the Build" Mean?

- Build fails after integration
- Code does not compile or tests fail

Impact:
A broken build stops the whole team

Never Break the Build

- Developers must run tests before committing
- Broken build must be fixed immediately

XP Mindset:
The build is sacred

Why Agreement Matters

- CI cannot be enforced by tools alone
- Team discipline is required
- Everyone agrees to protect the build

Key idea:
CI is a social agreement

Continuous Integration vs Ten-Minute Build

- CI is a practice
- Ten-minute build is a constraint
- Fast builds enable CI

Role of Automation in CI

- Builds must be automated
- Tests must be automated
- Feedback must be fast

Analogy:
Automatic signals vs manual traffic control

Tools Commonly Used for CI

- Version control: Git
- Build tools: Maven, Gradle
- CI servers: Jenkins, GitHub Actions

Benefits of Continuous Integration

- Early defect detection
- Reduced integration risk
- Faster development
- Higher software quality

Results of Practicing CI

- Stable main codebase
- Predictable releases
- Increased trust among team members

When CI Is Difficult

- Slow builds
- Manual testing
- Large untested legacy systems

Alternatives and Adaptations

- Partial CI for legacy code
- Fast smoke tests + full nightly build
