Agile Software Development

Developing

(Unit 4B)

Developing - Introduction

- Practices that keep the code clean and allow the entire team to contribute to development
  - Incremental Requirements
  - Customer Tests
  - Test-Driven Development
  - Refactoring
  - Simple Design
  - Incremental Design and Architecture
  - Spike Solutions
  - Performance Optimization
  - Exploratory Testing

Practices that keep the code clean

- ✧ Incremental Requirements
  - allows the team to get started while customers work out requirements details.
- ✧ Customer Tests
  - help communicate tricky domain rules.
- ✧ Test-Driven Development
  - allows programmers to be confident that their code does what they think it should.
- ✧ Refactoring
  - Enables programmers to improve code quality without changing its behavior.

Practices that keep the code clean - Cond

✧ Simple Design

- Allows the design to change to support any feature request

✧ Incremental Design and Architecture

- Allows programmers to work on features in parallel with technical infrastructure.

✧ Spike Solutions

- Use controlled experiments to provide information.

✧ Performance Optimization

- uses hard data to drive optimization efforts.

✧ Exploratory Testing

- enables testers to identify gaps in the team's thought processes.

Incremental Requirements

- We define requirements in parallel with other work.
- A team using an up-front requirements phase keeps their requirements in a requirements document.
- An XP team doesn't have a requirements phase and story cards aren't miniature requirements documents, so where do requirements come from?

Incremental Requirements – Contd

- ✧ **The Living Requirements Document**
- ✧ **Work Incrementally**

The Living Requirements Document

- ✧ In XP, the on-site customers sit with the team.
- ✧ They're expected to have all the information about requirements at their fingertips.
- ✧ When somebody needs to know something about the requirements for the project, he/she asks one of the on-site customers rather than looking in a document.
- ✧ The key to successful requirements analysis in XP is expert customers.
- ✧ Involve real customers, an experienced product manager, and experts in your problem domain

Work Incrementally

- ✧ Work on requirements incrementally, in parallel with the rest of the team's work
  - Vision, features, and stories
    - Start by clarifying your project vision, then identify features and stories (Release Planning)
  - Rough expectations
  - Mock-ups, customer tests, and completion criteria
  - Customer review
    - Only working applications show customers what they're really going to get

Customer Tests

- ✧ Customers have specialized expertise, or domain knowledge, that programmers don’t have.
- ✧ Some areas of the application - what programmers call domain rules - require this expertise
- ✧ To create customer tests, follow these processes
  - Describe,
  - Demonstrate, and
  - Develop

Customer Tests – Contd.

✧ Describe

- At the beginning of the iteration, look at your stories and decide whether there are any aspects that programmers might misunderstand.
- Customer tests are for communication, not for proving that the software works.

Customer Tests – Contd.

- **Demonstrate**
  - After a brief discussion of the rules, provide concrete examples that illustrate the scenario.
  - Tables are often the most natural way to describe this information

Demonstrate (Using Tables)

Sent	User	OK to delete
N	Account Rep	Y

N	CSR	Y
N	Manager	Y
N	Admin	Y

Y	Account Rep	N
Y	CSR	N
Y	Manager	Audited
Y	Admin	Audited

Demonstrate (Using Tables) – Contd.

- As an example, invoice hasn't been sent to customers, so an Account Rep can delete it.
- In fact, anybody can delete it—CSRs, managers, and admins.
- But once it's sent, only managers and admins can delete it, and even then it's audited.

Customer Tests – Contd.

✧ Develop

- When you’ve covered enough ground, document your discussion so the programmers can start working on implementing your rules
- Programmers: Once they have some examples, can start implementing the code using normal Test Driven Development (TDD)
- Don’t use the customer tests as a substitute for writing your own tests

Customer Tests – Contd.

- ✧ Focus on Business Rules
- ✧ Ask Customers to Lead
- ✧ Automating the Examples

Customer Tests – Contd.

- Focus on Business Rules
  - To show that an account rep must not delete a mailed invoice, you might make the mistake of writing this:
    - Log in as an account rep
    - Create new invoice
    - Enter data
    - Save invoice
    - Email invoice to customer
    - Check if invoice can be deleted (should be “no”)
  - What happened to the core idea? It’s too hard to see
  - Compare that to the previous approach:

Sent	User	OK to delete
Emailed	Account Rep	N

Customer Tests – Contd.

- Ask Customers to Lead
  - Remember the "customer" in "customer tests"

Test-Driven Development

- We produce
  - Well-designed,
  - Well-tested, and
  - Well-factored code in small, verifiable steps.

Test-Driven Development (TDD)

- It is a rapid cycle of testing, coding, and refactoring.
- When adding a feature, a pair may perform dozens of these cycles, implementing and refining the software in baby steps until there is nothing left to add and nothing left to take away.
- Research shows that TDD substantially reduces the incidence of defects.
- When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

Why TDD

- ✧ TDD applies a principle similar to the IDEs (Integrated Development Environment) to programmer intent.
- ✧ Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code.
- ✧ Every few minutes—as often as every 20 or 30 seconds—TDD verifies that the code does what you think it should do.
- ✧ If something goes wrong, there are only a few lines of code to check. Mistakes are easy to find and fix.

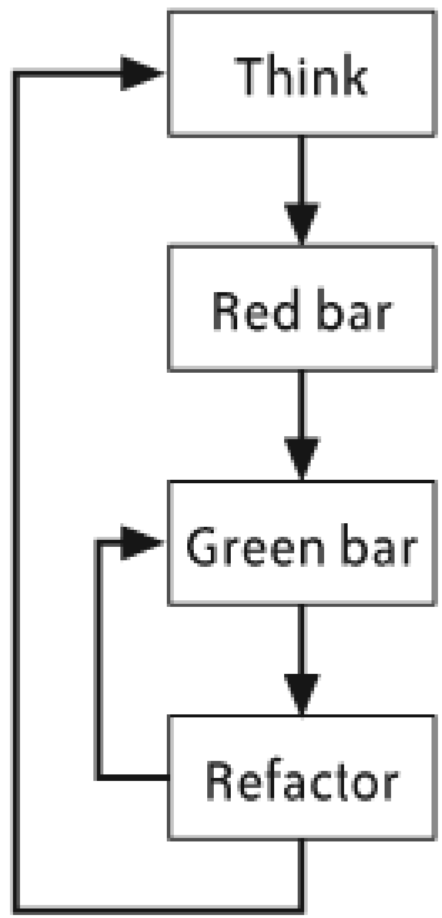
Why TDD – Contd.

- ✧ TDD uses an approach similar to double-entry bookkeeping.
- ✧ You communicate your intentions twice, stating the same idea in different ways:
  - First with a test.
  - Then, with production code.
- ✧ When they match, it’s likely they were both coded correctly.
- ✧ If they don’t, there’s a mistake somewhere.

How to Use TDD

- ✧ Every few minutes, TDD provides proven code that has been tested, designed, and coded.
- ✧ To use TDD, follow the “red, green, refactor” cycle.

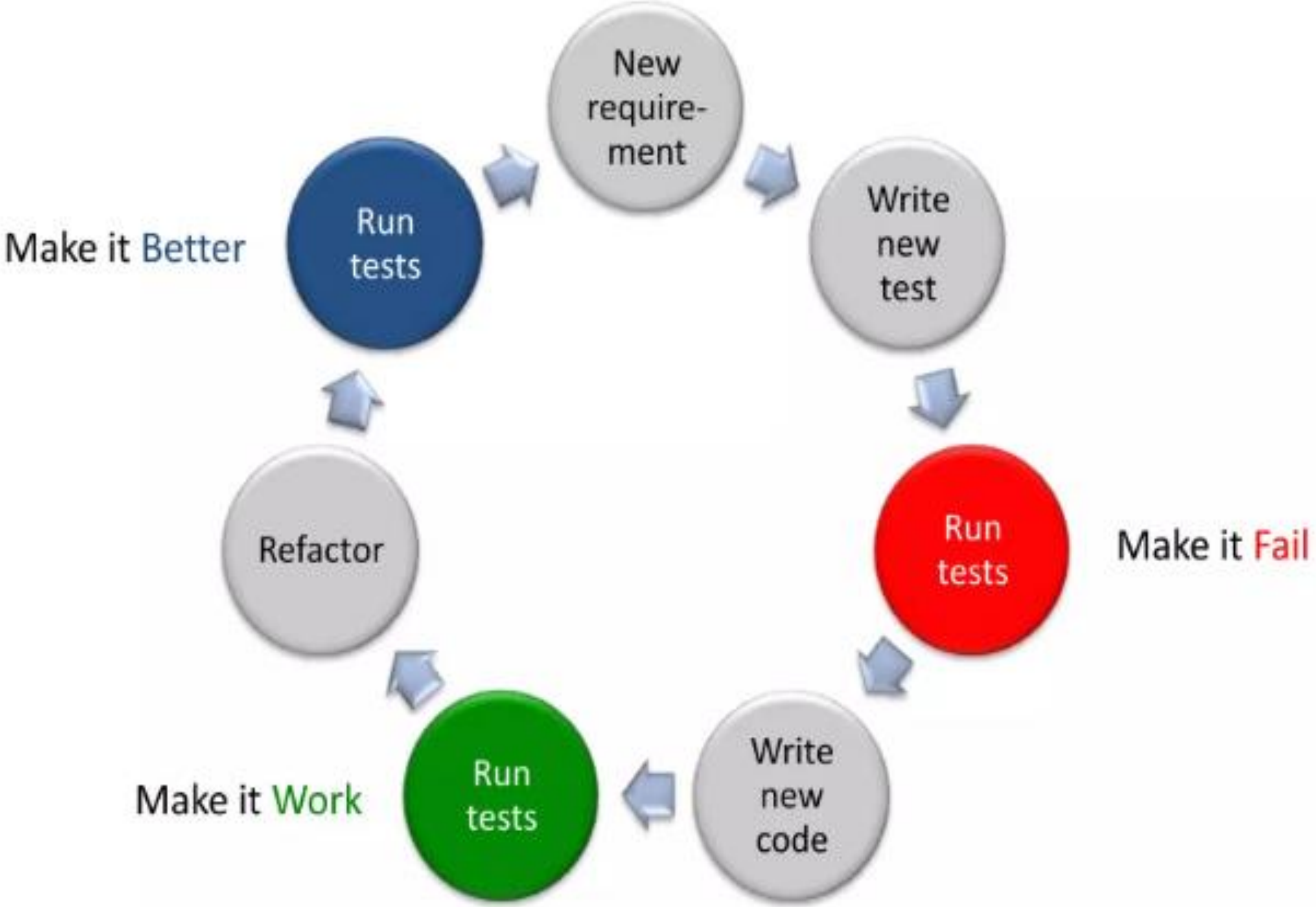
The TDD cycle



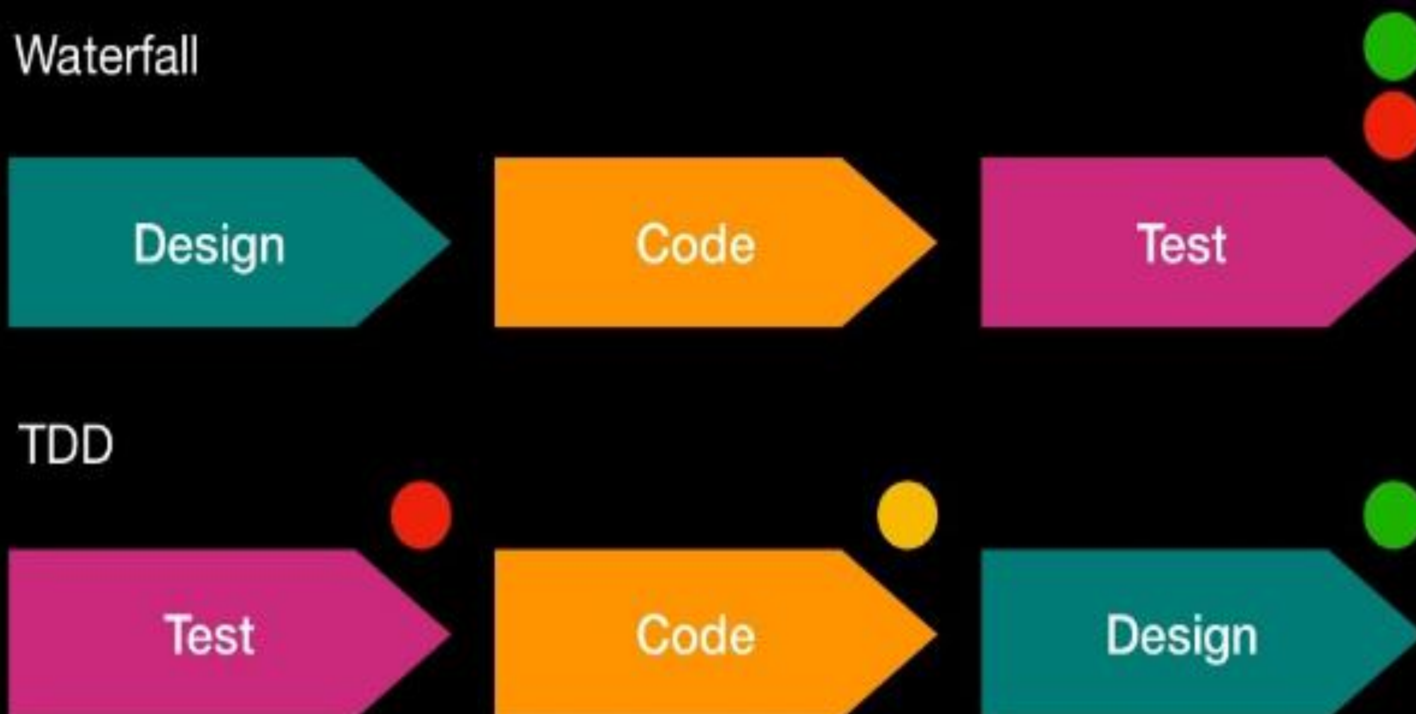
Understand the Red Green Refactor Methodology

- The Red, Green, Refactor method consists of three phases:
- Red - write a test that fails.
- Green - implement the test-supporting functionality to pass the test.
- Refactor - improve the production code AND the tests to absolute perfection.
- The Red, Green, Refactor cycle should be as short as possible.
- The main objective of this method is to advance in small increments.

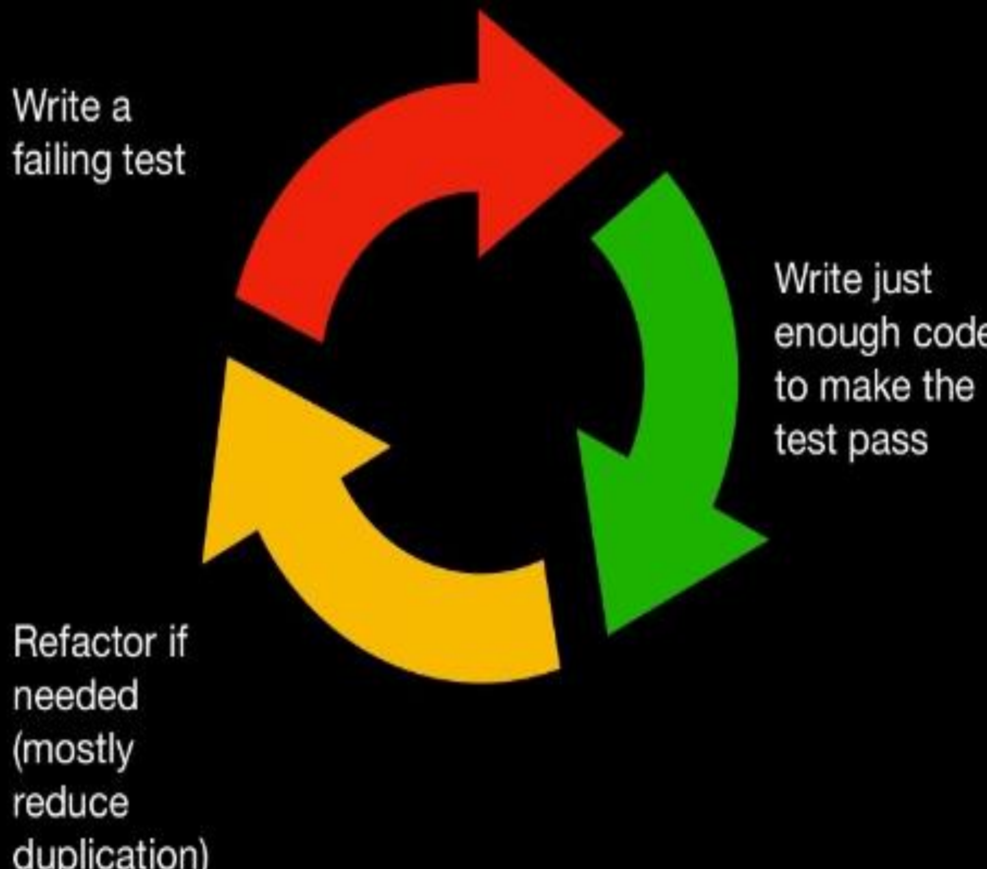
TDD CYCLE


New requirement
Write new test
Run tests
Make it Fail
Write new code
Run tests
Make it Work
Refactor
Run tests
Make it Better

Classic post-test vs TDD



TDD cycle


Write a failing test
Write just enough code to make the test pass
Refactor if needed (mostly reduce duplication)

Write a failing test

Write a test to
prove a simple case

Run the test and
see how it fails

Errors == test fails

Make the test pass

Write production code until:

- all errors are fixed
- the test passes

Refactor

- Keep tests passing
- Remove duplication
- Improve design

Outcomes

Always backed by tests

Decoupled design

Automatically you get:

- full coverage
- regression tests

TDD – Step 1: Think

- TDD uses small tests to force you to write your code—you only write enough code to make the tests pass.
  - Imagine what behavior you want your code to have,
  - Then think of a small increment that will require fewer than five lines of code.
  - Next, think of a test—also a few lines of code—that will fail unless that behavior is present.
- In other words, think of a test that will force you to add the next few lines of production code.
- This is the hardest part of TDD.

TDD – Step 2: Red bar

- ✧ Now write the test.
  - Write only enough code for the current increment of behavior—typically fewer than five lines of code.
  - Code in terms of the class' behavior and its public interface, not how you think you will implement the internals of the class. Respect encapsulation.
  - In the first few tests, this often means you write your test to use method and class names that don't exist yet.
- ✧ After the test is coded, run your entire suite of tests and watch the new test fail.
- ✧ In most TDD testing tools, this will result in a red progress bar.

TDD – Step 3: Green bar

- Next, write just enough production code to get the test to pass.

TDD - Step 4: Refactor

- ✧ With all your tests passing again, you can now refactor without worrying about breaking anything.
- ✧ Review the code and look for possible improvements.
- ✧ Ask your navigator if he’s made any notes.

TDD - Step 5: Repeat

- ✧ When you’re ready to add new behavior, start the cycle over again.
- ✧ The key to TDD is small increments.

TDD – Testing Tools

- To use TDD, you need a testing framework.
- The most popular are the open source xUnit tools
  - JUnit (for Java) and NUnit (for .NET)

Refactoring

- ✧ Every day, our code is slightly better than it was the day before.
- ✧ Refactoring is the process of changing the design of your code without changing its behavior—
  - what it does stays the same, but how it does it changes.
- ✧ Refactoring is also reversible.

Refactoring - Reflective Design

- ✧ Refactoring enables an approach to design, what is called reflective design.
- ✧ In addition to creating a design and coding it, you can now analyze the design of existing code and improve it.
- ✧ One of the best ways to identify improvements is with code smells:
  - condensed nuggets of wisdom that help you identify common problems in design.
  - A code smell doesn't necessarily mean there's a problem with the design.
  - It's like a funky smell in the kitchen: it could indicate that it's time to take out the garbage.

Refactoring

✧ Analyzing Existing Code

- Reflective design requires that you understand the design of existing code.
  - The easiest way to do so is to ask someone else on the team.
  - A conversation around a whiteboard design sketch is a great way to learn.
- In some cases, no one on the team will understand the design, or you may wish to dive into the code yourself.

Refactoring

✧ How to Refactor

- When you refactor, proceed in a series of small transformations.
- Each refactoring is like making a turn on a Rubik's cube.
- To achieve anything significant, you have to string together several individual refactorings, just as you have to string together several turns to solve the cube.

✧ Refactoring isn't rewriting.

Simple Design

- ✧ Design should be easy to modify and maintain.
- ✧ Appropriate for the intended audience
  - It doesn’t matter how brilliant and elegant a piece of design is; if the people who need to work with it don’t understand it, it isn’t simple for them.
- ✧ Communicative
  - Every idea that needs to be communicated is represented in the system.
  - Like words in a vocabulary, the elements of the system communicate to future readers.

Simple Design – Contd.

- ✧ Factored
  - Duplication of logic or structure makes code hard to understand and modify.
- ✧ Minimal
  - Within the above three constraints, the system should have the fewest elements possible.
  - Fewer elements means less to test, document, and communicate.

Simple Design – Contd.

✧ Points to keep in mind as you strive for simplicity.

- You Aren’t Gonna Need It (YAGNI)
- Once and Only Once
- Self-Documenting Code
- Isolate Third-Party Components
- Limit Published Interfaces
- Fail Fast

Incremental Design and Architecture

- ✧ We deliver stories every week without compromising design quality
- ✧ XP makes challenging demands of its programmers
  - Every week, programmers should finish 4 to 10 customer-valued stories.
- ✧ Your customers need you to deliver completed stories.
- ✧ XP provides a solution for this dilemma: Incremental Design

Incremental Design

- It is also called Evolutionary Design
- It allows you to build technical infrastructure (such as domain models and persistence frameworks) incrementally, in small pieces, as you deliver stories.

Incremental Design and Architecture – Contd.

- ✧ Continuous Design
- ✧ Incrementally Designing Methods
- ✧ Incrementally Designing Classes
- ✧ Incrementally Designing Architecture
- ✧ Risk-Driven Architecture

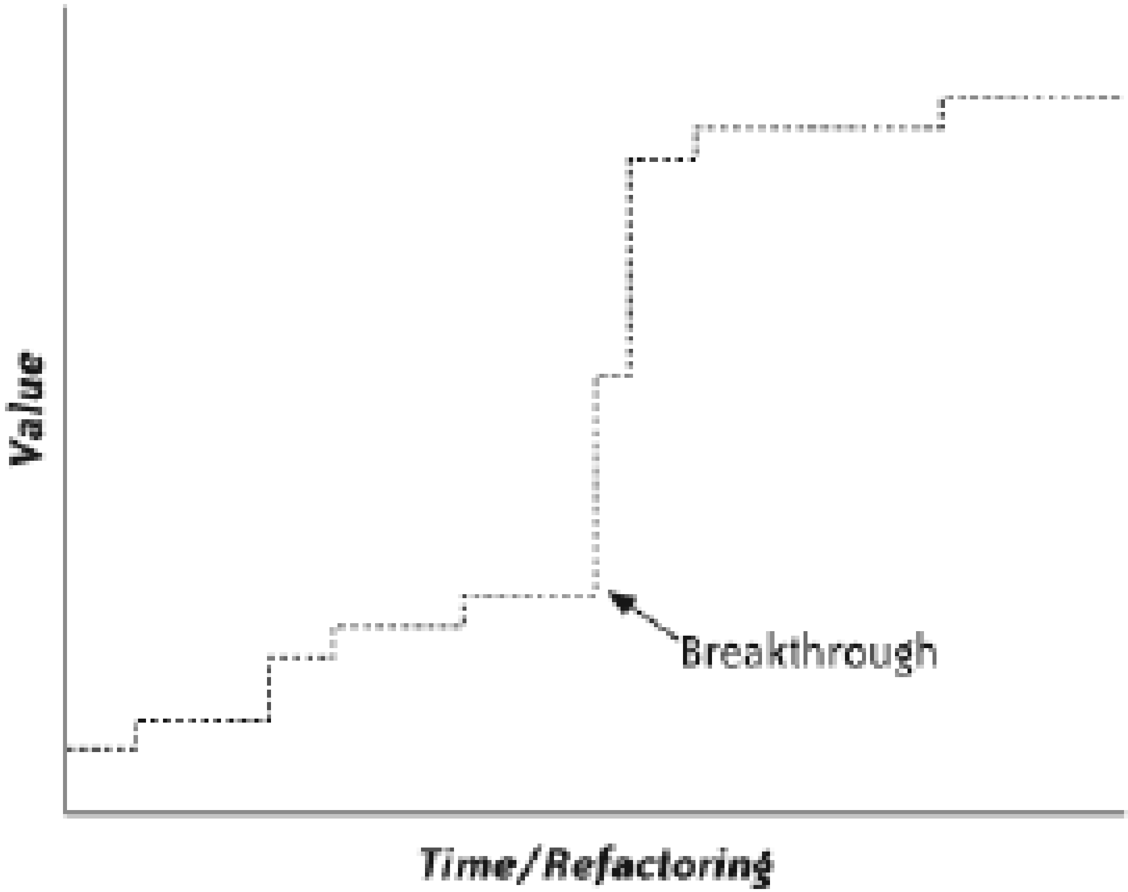
Continuous Design

- ✧ Incremental design initially creates every design element
  - Method, class, namespace, or even architecture to solve a specific problem.
- ✧ Additional customer requests guide the incremental evolution of the design.
- ✧ No matter what level of design you're looking at, the design tends to improve in bursts.

Continuous Design

- ✧ You will implement code into the existing design for several cycles, making minor changes as you go.
- ✧ Then something will give you an idea for a new design approach, which will require a series of refactoring's to support it.
- ✧ This is called a Breakthrough
- ✧ Breakthroughs happen at all levels of the design, from methods to architectures
- ✧ Breakthroughs are the result of important insights and lead to substantial improvements to the design.

Continuous Design - Breakthrough



Incrementally Designing Methods

- It’s test-driven development.
- While the driver implements, the navigator thinks about the design.
- He looks for overly complex code and missing elements, which he writes on his notecard.
- He thinks about which features the code should support next, what design changes might be necessary, and which tests may guide the code in the proper direction.

Incrementally Designing Classes

- ✧ When TDD is performed well, the design of individual classes and methods is beautiful:
  - they're simple, elegant, and easy to use.
- ✧ This isn't enough.
- ✧ Without attention to the interaction between classes, the overall system design will be muddy and confusing.

Incrementally Designing Classes

- During TDD, the navigator should also consider the wider scope.
- Ask these questions:
  - Are there similarities between the code you're implementing and other code in the system?
  - Are class responsibilities clearly defined and concepts clearly represented?
  - How well does this class interact with other classes?

Incrementally Designing Classes

- Class-level refactoring's happen several times per day.
- Depending on your design, breakthroughs may happen a few times per week and can take several hours to complete.
- Nonetheless, remember to proceed in small, test-verified steps.
- Use your iteration slack to complete breakthrough refactorings

Incrementally Designing Architecture

- Large programs use overarching organizational structures called architecture.
  - For example, many programs segregate user interface classes, business logic classes, and persistence classes into their own namespaces
  - This is a classic three-layer architecture

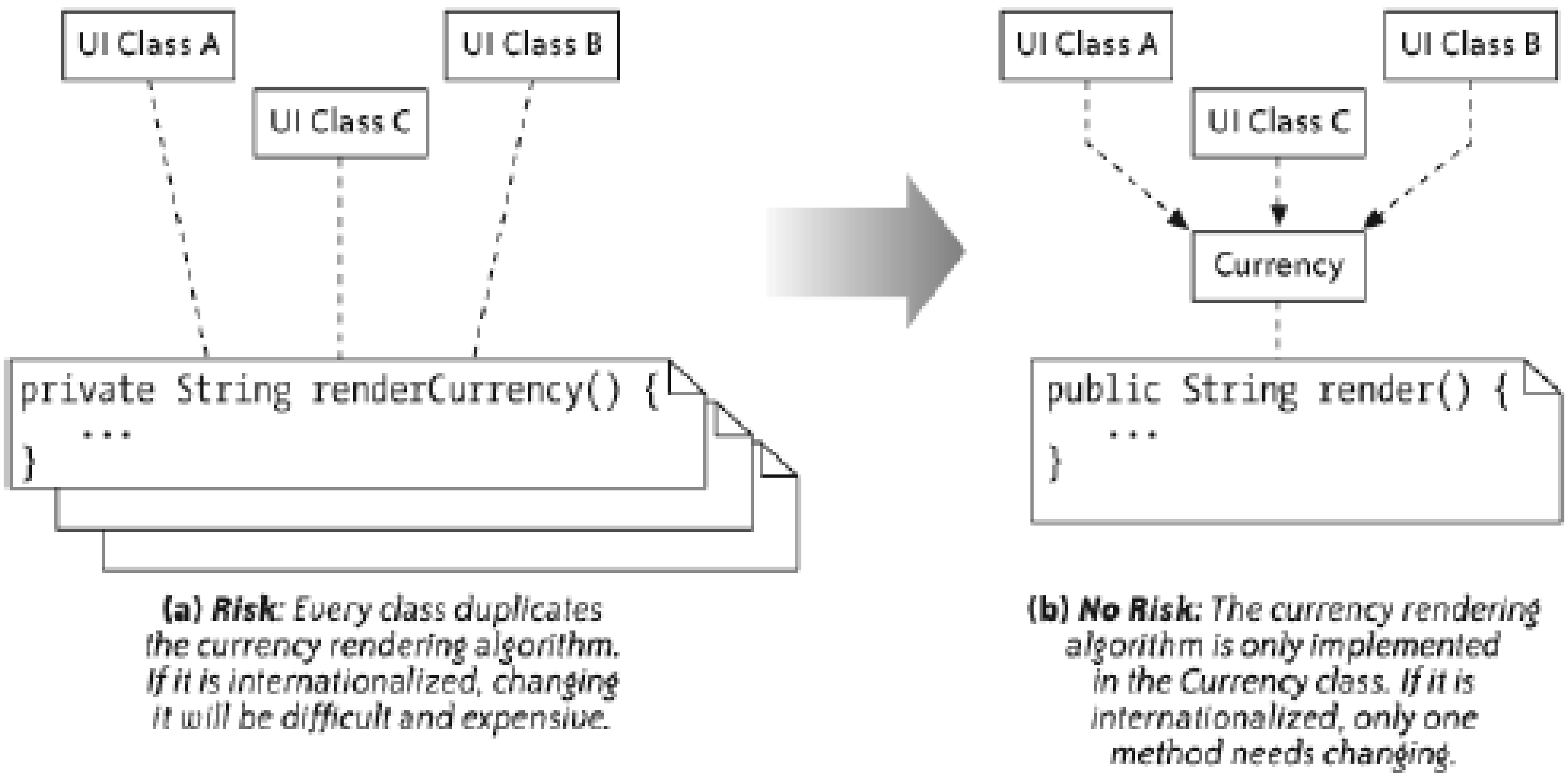
Incrementally Designing Architecture

- ✧ You can also design architectures incrementally.
- ✧ As with other types of continuous design, use TDD and pair programming as your primary vehicle.
- ✧ While your software grows, be conservative in introducing new architectural patterns:
  - Introduce just what you need to for the amount of code you have and the features you support at the moment
- ✧ Breakthroughs in architecture happen every few months

Incrementally Designing Architecture

- Refactoring to support the breakthrough can take several weeks or longer because of the amount of duplication involved.
- Although changes to your architecture may be tedious, they usually aren’t difficult once you’ve identified the new architectural pattern
- Start by trying out the new pattern in just one part of your design
- Keep delivering stories while you refactor

Risk-Driven Architecture



Risk-Driven Architecture

- ✧ Your power lies in your ability to choose which refactorings to work on.
- ✧ Although it would be inappropriate to implement features your customers haven't asked for, you can direct your refactoring efforts toward reducing risk.
- ✧ Anything that improves the current design is OK
- ✧ So, choose improvements that also reduce future risk.

Spike Solutions

- ✧ We perform small, isolated experiments when we need more information
- ✧ XP values concrete data over speculation
- ✧ Whenever you're faced with a question, don't speculate about the answer - conduct an experiment!
- ✧ Figure out how you can use real data to make progress
- ✧ That's what spike solutions are for

Spike Solutions

- A spike solution, or spike, is a technical investigation.
- It's a small experiment to research the answer to a problem.
  - For example, a programmer might not know whether Java throws an exception on arithmetic overflow.
  - A quick 10-minute spike will answer the question

Spike Solutions – Contd.

- ✧ The best way to implement a spike is usually to create a small program or test that demonstrates the feature in question
- ✧ Spike solutions clarify technical issues by setting aside the complexities of production code

Scheduling Spikes

- You see a need to clarify a small technical issue, and you write a quick spike to do so.
  - If the spike takes more than a few minutes, your iteration slack absorbs the cost.
- Sometimes you won't be able to estimate a story at all until you've done your research
  - In this case, create a spike story and estimate that instead

Performance Optimization

- ✧ Modern computers are complex.
- ✧ Reading a single line of a file from a disk requires the coordination of the CPU, the kernel, a virtual file system, system bus, system memory, etc.
- ✧ There are so many variables it’s nearly impossible to predict the general performance of any single method.

Performance Optimization

- ✧ The days in which a programmer could accurately predict performance by counting instruction clock cycles are long gone.
- ✧ A holistic approach is the only accurate way to optimize such complex systems.
  - Measure the performance of the entire system, make an educated guess about what to change, then remeasure.
  - If the performance gets better, keep the change.
  - If it doesn’t, discard it.
  - Once your performance test passes, stop - you’re done

Performance Optimization

- ✧ Use a profiler to guide your optimization efforts
  - Find the bottlenecks, and focus your efforts on reducing them
- ✧ Although optimizations often make code more complex, keep your code as clean and simple as possible.

When to Optimize

- ✧ Performance optimizations must serve the customer's needs
- ✧ Optimization has two major drawbacks:
  - It often leads to complex, buggy code, and
  - It takes time away from delivering features.
- ✧ Neither is in your customer's interests.

When to Optimize

- ✧ Optimize only when it serves a real, measurable need.
- ✧ It means your priority should be code that's clean and elegant.
- ✧ Once a story is done, if you're still concerned about performance, run a test.
- ✧ If performance is a problem, fix it
  - But, let your customer make the business decision about how important that fix is.

How to Write a Performance Story

- Performance stories need a concrete, customer-valued goal
  - Throughput
    - How many operations should complete in a given period of time?
  - Latency
    - How much delay is acceptable between starting and completing a single operation?
  - Responsiveness
    - How much delay is acceptable between starting an operation and receiving feedback about that operation?

Exploratory Testing

- ✧ We discover surprises and untested conditions
- ✧ XP teams have no separate QA department.
  - There’s no independent group of people responsible for assessing and ensuring the quality of the final release.
  - Instead, the whole team - customers, programmers, and testers – is responsible for the outcome

Exploratory Testing

- ✧ Good testers have the ability to look at software from a new perspective, to find surprises, gaps, and holes.
- ✧ It takes time for the team to learn which mistakes to avoid.
- ✧ By providing essential information about what the team overlooks, testers enable the team to improve their work habits and achieve their goal of producing zero bugs.

Exploratory Testing

- ✧ One particularly effective way of finding surprises, gaps, and holes is exploratory testing:
  - A style of testing in which you learn about the software while simultaneously designing and executing tests, using feedback from the previous test to inform the next.
- ✧ Exploratory testing enables you to discover
  - Emergent behavior
  - Unexpected side effects,
  - Holes in the implementation, and
  - Risks related to quality attributes
- ✧ It's the perfect complement to XP's raft of automated testing techniques.

Exploratory Testing

- ✧ Exploratory testers use the following four tools to explore the software
  - Charters
  - Observation
  - Note Taking
  - Heuristics

Exploratory Testing - Charters

- Before beginning an exploratory session, a tester should have some idea of what to explore in the system and what kinds of things to look for.
  - This charter helps keep the session focused.
- The charter for a given exploratory session
  - Might come from a just-completed story
    - e.g., “Explore the Coupon feature”
  - It might relate to the interaction among a collection of stories
    - e.g., “Explore interactions bet. the Coupon and the Bulk Discount feature”.
  - It might involve a quality attribute, such as stability, reliability, or performance
    - “Look for evidence that the Coupon feature impacts performance”

Exploratory Testing - Observation

- ✧ Automated tests only verify the behavior that programmers write them to verify
- ✧ But humans are capable of noticing subtle clues that suggest all is not right.
- ✧ Exploratory testers are continuously alert for anything out of the ordinary.
  - This may be an editable form field that should be read-only,
  - A hard drive that spun up when the software should not be doing any disk access, or
  - A value in a report that looks out of place.

Exploratory Testing - Notetaking

- ✧ Exploratory testers keep a notepad beside them as they explore and periodically take notes on the actions they take
- ✧ You can also use screen recorders to keep track of what you do
- ✧ Notes and recordings tell you what you were doing not just at the time you encountered surprising behavior but also in the minutes or hours before.

Exploratory Testing - Heuristics

- A heuristic is a guide
  - a technique that aids in your explorations.
  - Boundary testing is an example of a test heuristic.
- If you have a field that’s supposed to accept numbers from 0 -100,
  - you’ll probably try valid values like 0, 100, and
  - something in the middle, and
  - invalid values like –1 and 101

UNIT-IV

1. What is a Vision Statement? Elaborate.
2. What are the approaches followed in Release Planning? Discuss.
3. Discuss the role of programmers and customers in a *planning game*.
4. Suggest ways to overcome disagreements during planning in XP.
5. Suppose your product manager doesn’t want to prioritize. He says everything is important. What will be your strategy in such a situation in agile development?
6. What do you mean by ‘miracle of collaboration’ with respect to planning?
7. Define the terms: transition indicators, mitigation activity, contingency activity, and risk exposure.
8. How do we plan Iteration in XP?
9. If we don’t estimate stories during iteration planning, when do we estimate stories?
10. How do you reduce the need for Slack in XP?
11. What are ‘Story Cards’/ Explain with an example.
12. How can we encourage stakeholders to use stories for requesting features?
13. Is it not a waste of time for all the programmers to estimate stories together?
14. Explain Test-Driven Development in detail.
15. Explain why test-first development helps the programmer to develop a better understanding of the system requirements. What are the potential difficulties with test-first development?
16. What do you mean by ‘Spike Solutions’?
17. Discuss the benefits of “Fail Fast”.
18. Criticize the role of optimizing in software development
19. What is velocity? How to improve velocity?
20. Justify why we need to make work-in progress documentation