1 Agile Development

Syllabus

Agile Practices, Overview of Extreme Programming, Planning, Testing, Refactoring.

Contents

1.0 Introduction

1.1 Agile Practices

1.2 Overview of Extreme Programming

1.3 Planning

1.4 Testing

1.5 Refactoring

1.6 Review Questions

1.0 Introduction

- Principles, patterns and practices are significant, but it's the people that mark them effort. As Alistair Cockburn says, 1 "Process and technology are a second-order conclusion on the outcome of a project. The first-order conclusion is the people."
- We cannot cope teams of programmers as if they were systems made up of components driven by a process. People are not "plug-compatible programming units."2 If our projects are to flourish, we are going to have to construct collective and self-organizing teams.
- Those companies that inspire the formation of such teams will have an enormous competitive advantage over those who hold the view that a software-development organization is nothing more than a mountain of twisty little people all alike.

1.1 Agile Practices

- The absence of effective practices in project leads to impulsiveness, repeated error and wasted effort. Customers are dissatisfied by slipping schedules, increasing budgets and poor quality. Developers are discouraged by working ever longer hours to yield ever poorer software.
- Once we have practiced such a fiasco, we become frightened of repeating the experience. Our fears encourage us to create a process that constrains our activities and stresses certain outputs and artifacts. We appeal these constraints and outputs from past experience, selecting things that appeared to work healthy in previous projects. Our hope is that they will work again and take away our doubts.
- However, projects are not so modest that a rare constraints and artifacts can dependably avoid error. As errors continue to be made, we identify those errors and put in place even more constraints and artifacts in order to avoid those errors in the future. After many, projects we may find ourselves loaded with a huge awkward process that greatly obstructs our ability to get anything done.
- A big awkward process can create the very difficulties that it is designed to avoid. It can slow the team to the degree that schedules slip and budgets swell. It can diminish responsiveness of the team to the point where they are always creating the incorrect product. Inappropriately, this leads many teams to believe that they don't have sufficient process. So, in a kind of runaway-process increase, they make their process ever larger.

1.1.1 Agile Alliance

- A group of industry professionals met in early 2001, inspired by the observation that software teams in many organizations were stuck in a maze of ever-increasing process, to describe the values and principles that would allow software teams to operate fast and respond to change. They called themselves the Agile Alliance. They spent the following few months putting together a statement of values. The Agile Alliance's Manifesto was the result.

The Manifesto of the Agile Alliance

Manifesto for Agile Software Development

We are revealing better ways of developing
software by doing it and helping others do it.
Through this work we have come to value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

1.1.1.1 Individuals and Interactions over Processes and Tools

- People are the most vital ingredient of success. A decent process will not save the project from failure if the team doesn't have solid players, but a corrupt process can make even the solidest of players unproductive. Even a group of strong players can fail severely if they don't work as a team.
- A solid player is not inevitably a first-rate programmer. A solid player may be a normal programmer, but someone who works well with others. Working well with others, cooperative and interacting, is more important than raw programming talent. A team of regular programmers who communicate well are more likely to thrive than a group of superstars who fail to interact as a team.
- The correct tools can be very imperative to success. Compilers, IDEs, source-code control systems, etc. are all vital to the correct functioning of a team of developers. However, tools can be exaggerated. An excess of big, cumbersome tools is just as bad as a lack of tools.
- Remember, building the team is more vital than building the environment. Many teams and managers make the blunder of building the environment first and

guessing the team to gel automatically. In its place, work to create the team and then let the team organize the environment on the basis of need.

1.1.1.2 Working Software over Comprehensive Documentation

- Software without documentation is a tragedy. Code is not the perfect medium for communicating the rationale and structure of a system. Somewhat, the team needs to produce human-readable documents that define the system and the rationale for their design decisions.
- However, too much documentation is worse than not enough. Massive software documents require a long time to create and considerably longer to maintain in sync with the code. If they are not kept in sync, then they turn into huge, complex lies and become a substantial source of misdirection.
- It is always a good idea for the team to write and preserve a rationale and structure document, but that document requires to be small and salient. By "short," I mean one or two dozen pages at most. By "salient," I mean it should discuss the complete design rationale and only the highest-level structures in the system.
- The two documents that are the greatest at transferring information to new team members are the code and the team. The code does not falsehood about what it does. It may be hard to extract rationale and intent from the code, but the code is the only unmistakable source of information. The team members grasp the ever-changing road map of the system in their heads. There is no quicker and more efficient way to transfer that road map to others than human to human communication.
- Many teams have grown suspended up in the chase of documentation instead of software. This is frequently a fatal flaw. There is a simple rule called Martin's first law of documentation that avoids it :

  Produce no document unless its need is immediate and significant.

1.1.1.3 Customer Collaboration over Contract Negotiation

- Software is not a commodity that can be bought and sold. You can't create a representation of the software you want and then have it developed on a certain schedule for a set price. Attempts to treat software projects in this manner have often failed. The failures might be spectacular at times.
- It is alluring for the managers of a company to say their development staff what their needs are and then think that staff to go away for a while and return with a system that pleases those needs. However, this mode of operation hints to poor quality and failure.

- Successful projects contain customer feedback on a consistent and frequent basis. Rather than contingent on a contract or a statement of work, the customer of the software works thoroughly with the development team, providing regular feedback on their efforts.
- A contract that stipulates the requirements, schedule and cost of a project is essentially flawed. In most cases, the terms it specifies become worthless long before the project is completed. The best contracts are those that govern the way the development team and the customer will work together.

1.1.1.4 Responding to Change over Following a Plan

- It is the ability to react to change that frequently determines the success or failure of a software project. When we build plans, we need to make sure that our plans are flexible and ready to adjust to changes in the business and technology.
- The course of a software project cannot be scheduled very far into the future. First of all, the business environment is likely to change, producing the requirements to shift. Second, customers are expected to alter the requirements once they see the system start to function. Finally, even if we recognize the requirements and we are sure they won't change, we are not very good at guessing how long it will take to develop them.
- It is alluring for trainee managers to create a nice PERT or Gantt chart of the whole project and tape it to the wall. They may sense that this chart gives them control over the project. They can track the individual tasks and cross them off the chart as they are finished. They can compare the actual dates with the planned dates on the chart and respond to any discrepancies.
- What really happens is that the structure of the chart damages ? As the team gains knowledge about the system, and as the customers gain knowledge about their needs, certain tasks on the chart become pointless. Other tasks will be exposed and will need to be added. In short, the plan will undergo changes in shape, not just changes in dates.
- An improved planning strategy is to make detailed plans for the next two weeks, very coarse plans for the next three months and enormously crude plans beyond that. We should know the tasks we will be working on for the next two weeks. We should coarsely know the requirements we will be working on for the next three months. And we should have only an imprecise idea what the system will do after a year.

1.1.2 Principles

- The above values motivated the following 12 principles, which are the characteristics that distinguish a set of agile practices from a heavyweight process :
- *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
  - The MIT Sloan Management Review published an analysis of software development practices that help companies construct high-quality products. The article found a number of practices that had a significant influence on the quality of the final system. One practice was a strong association between quality and the early delivery of a moderately functioning system. The article reported that the less functional the initial delivery, the higher the quality in the final delivery.
  - An agile set of practices delivers early and often. We struggle to deliver a fundamental system within the first few weeks of the start of the project. Then, we try to continue to deliver systems of increasing functionality every two weeks.
  - Customers may pick to put these systems into production if they reason that they are functional adequate. Or they may choose simply to review the current functionality and report on changes they need made.
- *Changes in requirements are welcome, especially if they occur late in the development process. Agile methods facilitate change for the benefit of the customer's competitive position.*
  - This is an attitude statement. In an agile process, participants are not afraid of change. They see modifications in the specifications as positive since it means the team has gained a better understanding of what it will take to delight the market.
  - When needs change, an agile team works very hard to maintain the structure of its software adaptable so that the impact on the system is low.
- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the tinier time scale.*
  - We deliver working software and we delivery it early and frequently. We are not content with delivering packets of documents or plans. We don't count those as real deliveries. Our eye is on the goal of delivering software that satisfies the customer's needs.
- *Business people and developers must work together every day throughout the project.*
  - Customers, developers and stakeholders must have regular and meaningful

interactions in order for a project to be agile. A software project isn't like a weapon that you shoot and forget about. A software project must be guided indefinitely.

- *Build projects around people who are passionate about what they're doing. Give them the environment and support they require, and trust them to do the task.*
  - A project that is agile considers people to be the most crucial aspect in its success.
  - All other factors - process, environment, management, etc. - are measured to be second order effects and they are subject to change if they are having a contrary effect upon the people.
  - For example, if the office environment is an obstacle to the team, the office environment must be changed. If certain process steps are an obstacle to the team, the process steps must be changed.
- *The most efficient and effective method of assigning information to and within a development team is face-to- face conversation.*
  - In an agile project, people talk to each other. The main mode of communication is conversation. Documents may be created, but there is no effort to capture all project information in writing. An agile project team does not claim written specs, written plans or written designs. Team members may create them if they notice an immediate and significant need, but they are not the default. The default is conversation.
- *Working software is the prime measure of progress.*
  - Agile projects measure their progress by measuring the amount of software that is presently meeting the customer's need. They don't measure their progress in terms of the phase that they are in or by the volume of documentation that has been produced or by the amount of infrastructure code they have created. They are 30 % done when 30 % of the required functionality is working.
- *Agile processes support long-term development. Sponsors, developers and consumers should all be able to keep up a steady pace indefinitely.*
  - An agile project is not run like a 50-yard dash; it is run like a marathon. The team does not lift off at full speed and attempts to maintain that speed during the duration. Rather, they move at a quick yet steady speed.
  - Burnout, shortcuts and disaster are all consequences of going too fast. Agile teams work at their own speed. They don't let themselves become too exhausted. They don't spend tomorrow's energy in order to get a little more done today.

They operate at a pace that allows them to retain the highest level of quality throughout the project.

- *A constant focus on technical excellence and smart design improves agility.*
  - The importance of great quality above rapid speed cannot be overstated. Keeping the software as clean and healthy as possible is the fastest way to progress. As a result, every member of the agile team is dedicated to generating only the finest quality code possible.
  - They don't make mistakes and then convince themselves that they'll fix it when they have more time. If they make a chaos, they clean it up before they finish for the day.
- *Simplicity - The art of exploiting the amount of work not done is crucial.*
  - Agile teams do not try to build the magnificent system in the sky. Rather, they always take the simplest path that is reliable with their goals. They don't put a lot of importance on antedating tomorrow's problems, nor do they try to defend against all of them today. Instead, they do the simplest and highest-quality work today, confident that it will be easy to change if and when tomorrow's problems ascend.
- *The best architectures, requirements and designs arise from self-organizing teams.*
  - An agile team is a self-organizing team. Individual team members are not given responsibilities from the outside. Responsibilities are conveyed to the entire team, and the team decides how best to carry them out.
  - Members of the agile team collaborate on all aspects of the project. Each person is allowed to contribute to the whole. No one in the team is responsible for the architecture, requirements, or tests. These tasks are shared by the team and each team member has influence over them.
- *At regular intervals, the team echoes on how to become more effective, then refrains and adjusts its behaviour accordingly.*
  - An agile team repeatedly adjusts its organization, rules, conventions, relationships, etc. An agile team knows that its environment is unceasingly changing and knows that they must change with that environment to persist agile.

1.1.3 Conclusion

- The professional goal of every software developer and every development team is to deliver the uppermost possible value to their employers and customers. And yet

our projects fail or fail to bring value, at a horrifying rate. Though well intentioned, the ascendant spiral of process inflation is responsible for at least some of this failure. Agile software development principles and values were developed to assist teams in breaking the cycle of process inflation and focusing on basic strategies for achieving their objectives.

1.2 Overview of Extreme Programming

- Extreme programming is the most well-known of the agile methods. It is made up of a set of simple, yet co-dependent practices. These practices work together to form a whole that is greater than its parts.

1.2.1 Customer Team Member

- We want the customer and developers to work closely with each other so that they are both aware of each other's problems and are working together to resolve those problems.
- Customers are the individual or group who describes and prioritizes features is the client of an XP team.
- The customer is sometimes a team of business analysts or marketing experts who work for the same organization as the engineers. The customer is sometimes a user representative appointed by the user body.
- Sometimes the paying customer is the customer. In an XP project, however, the clients are members of the team and are available to them.
- The top case is for the customer to work in the same room as the developers. Next best is if the customer works in within 100 feet of the developers. The superior the distance, the tougher it is for the customer to be a true team member. If the customer is in another building or another state, it is very tough to integrate him or her into the team.
- It is recommended to find someone who can be close by and who is eager and able to stand in for the true customer.

1.2.2 User Stories

- In order to plan a project, we must know something about the requirements, but we don't need to know very much.
- For planning purposes, we only want to know enough about a requirement to guess it. You may think that in order to estimate a requirement you need to know all its details, but that's not relatively true. You have to know that there are details and

you have to know coarsely the kinds of details there are, but you don't have to know the particulars.

- The specific details of a requirement are likely to change with time, particularly once the customer initiates to see the system come together. There is nothing that emphases requirements better than seeing the embryonic system come to life. Therefore, seizing the specific details about a requirement long before it is implemented is likely to result in unused effort and impulsive focusing.
- When using XP, we get the sense of the details of the requirements by talking them over with the customer, but we do not seizure that detail. Rather, the customer writes a few words on an index card that we agree will prompt us of the conversation. The developers write an estimate on the card at coarsely the same time that the customer writes it. They base that estimate on the sense of detail they got during their conversations with the customer.
- A user story is a mnemonic token of an ongoing conversation about a requirement. It is a planning instrument that the customer uses to schedule the implementation of a requirement based upon its priority and estimated cost.

1.2.3 Short Cycles

- Every two weeks, an XP project delivers working software. Each of these two-week iterations results in working software that addresses part of the stakeholders' requirements. The system is presented to the stakeholders at the end of each iteration in order to obtain feedback.

1.2.3.1 Iteration Plan

- An iteration lasts about two weeks on average. It's a little delivery that might or might not be put into production. It's a collection of user stories chosen by the client within the parameters of a budget set by the developers.
- The budget for an iteration is determined by the amount of work completed in the previous iteration. The customer may choose any number of stories for the iteration, so long as the total of their estimates does not surpass that budget.
- Once an iteration has been started, the customer approves not to change the definition or priority of the stories in that iteration. During this time, the developers are free to censor the stories up in to tasks and to develop the tasks in the order that makes the best technical and business sense.

1.2.3.2 Release Plan

- XP teams often create a release plan that plans out the next six iterations or so. That plan is known as a release plan. A release is usually three months' worth of work. It characterizes a main delivery that can usually be put into production. A release plan entails of prioritized collections of user stories that have been selected by the customer conferring to a budget given by the developers.
- The developers set the budget for the release by measuring how much they got done in the preceding release. The customer may choose any number of stories for the release so long as the total of the estimates does not surpass that budget. The customer also governs the order in which the stories will be implemented in the release. If the team so desires, they can map out the first few iterations of the release by displaying which stories will be completed in which iterations.
- Releases are not cast in stone. The customer can change the content at any time. He or she can abandon stories, write new stories, or change the precedence of a story.

1.2.4 Acceptance Tests

- The client specifies acceptance tests in which the details of the user stories are collected. Acceptance tests for a narrative are prepared soon before or even simultaneously with, the story's implementation. They are written in a scripting language that allows them to be executed repeatedly and automatically. They work together to ensure that the system is performing as the consumers have requested.
- The language of the acceptance tests grows and advances with the system. The customers may recruit the developers to create a simple scripting system, or they may have a discrete Quality Assurance (QA) department that can develop it. Many customers solicit the help of QA in developing the acceptance-testing tool and with writing the acceptance tests themselves.
- Once an acceptance test passes, it is added to the body of passing acceptance tests and is never permitted to fail again. This growing body of acceptance tests is run numerous times per day, every time the system is built. If an acceptance tests fails, the build is acknowledged a failure. Thus, once a requirement is implemented, it is never broken. The system transitions from one working condition to the next and is never allowed to be defective for more than a few hours.

1.2.5 Pair Programming

- All production code is written by two programmers working at the same time on the same computer. Each pair has one person who operates the keyboard and types

the code. The other person in the pair keeps an eye on the code being typed, looking for problems and ways to improve it. The two have a strong working relationship. Both are engrossed in the process of creating software.

- The roles change often. The driver may get exhausted or stuck, and his pair partner will grasp the keyboard and start to drive. The keyboard will move back and forth between them numerous times in an hour. Both members designed and wrote the code that resulted. Neither of them is entitled to more than half of the credit.
- Every programmer works in two separate pairs every day, therefore pair membership changes at least once a day. Every member of the team should have worked with every other member of the team at some point during the iteration, and they should have worked on just about everything that was going on.
- This dramatically increases the spread of knowledge over the team. While specialties persist and tasks that require certain specialties will typically belong to the suitable specialists, those specialists will pair with nearly everyone else on the team. This will spread the specialty out through the team such that other team members can fill in for the specialists in a pinch.

1.2.6 Test - Driven Development

- All production code is written in order to make failing unit tests pass. First we write a unit test that fails because the functionality for which it is testing doesn't exist. Then we write the code that marks that test pass.
- This iteration between writing test cases and code is very swift, on the order of a minute or so. The test cases and code develop together, with the test cases leading the code by a very small fraction.
- As a result, a very complete body of test cases grows along with the code. These tests permit the programmers to check whether the program works. If a pair makes a little modification, they can run the tests to ensure that nothing has been broken. Refactoring becomes considerably easier as a result of this.
- When you write code to make test cases pass, that code is testable by definition. Furthermore, there is a strong need to disconnect modules from one another so that they can be tested independently. As a result, the design of code created in this manner tends to be significantly less linked. The concepts of object-oriented design can help you with this decoupling in a big way.

1.2.7 Collective Ownership

- A pair has the right to check out any module and improve it. No programmers are separately responsible for any one particular module or technology. Everybody works on the GUI. Everyone works on the middleware. Everyone works on the database. Nobody has more authority over a module or a technology than anyone else.
- This doesn't mean that XP rejects specialties. If your specialty is the GUI, you are most likely to work on GUI tasks, but you will also be requested to pair on middleware and database tasks. If you choose to learn a second specialty, you can sign up for tasks and work with specialists who will teach it to you. You are not limited to your specialty.

1.2.8 Continuous Integration

- The programmers check in their code and integrate numerous times per day. The rule is simple. The first one to check in wins, everyone else merges.
- XP teams use non-blocking source control. This means that programmers are acceptable to check any module out at any time, regardless of who else may have it checked out. When the programmer checks the module back in after modifying it, he must be ready to merge it with any changes made by anyone who checked the module in onward of him. To evade long merge sessions, the members of the team check in their modules very often.
- A pair will work for an hour or two on a task. They create test cases and production code. At certain suitable breaking point, perhaps long before the task is complete, the pair decides to check the code back in. They first make certain that all the tests run. They integrate their new code into the present code base. If there is a merge to do, they do it. If necessary, they refer with the programmers who beat them to the check in. Once their changes are integrated, they build the new system. They run every test in the system, including all currently running acceptance tests. If they broke anything that used to work, they fix it. Once all the tests run, they finish the check in.
- Thus, XP teams will build the system several times each day. They build the whole system from end to end. If the final result of a system is a CD, they cut the CD. If the final result of the system is a dynamic Web site, they install that Web site, perhaps on a testing server.

1.2.9 Sustainable Pace

- A software project is not a sprint; it is a marathon. A team that jumps off the starting line and starts racing as fast as it can will burn out long earlier they are close to finishing. In order to finish quickly, the team must run at a sustainable pace; it must preserve its energy and alertness. It must deliberately run at a steady, moderate pace.
- The XP rule is that a team is not permitted to work overtime. The only exception to that rule is the last week in a release. Overtime is permitted if the team is within striking distance of their release target and can sprint to the finish.

1.2.10 Open Workspace

- In an open room, the team collaborates. There are workstations set up on the tables. There are two or three of these workstations on each table. Each workstation has two seats in front of it for pairs to sit in. Status charts, task breakdowns, UML diagrams and other visual aids cover the walls.
- There's a gentle buzz of conversation in this space. Each pair is within hearing distance of the other. When an option is in trouble, everyone gets the opportunity to hear about it. Each is aware of the other's situation. The programmers are in a great position to communicate.
- One might think that this would be a disrupting environment. It would be easy to fear that you'd never get anything done because of the continuous noise and distraction. In fact, this doesn't turn out to be the case. Furthermore, instead of interfering with productivity, a University of Michigan study suggested that working in a "war room" environment may surge productivity by a factor of two.

1.2.11 Planning Game

- The core of the planning game is the division of responsibility between business and development. The business people (a.k.a. the customers) select how important a feature is, and the developers resolve how much that feature will cost to implement.
- At the opening of each release and each iteration, the developers give the customers a budget, built on how much they were able to get done in the last iteration or in the last release. The customers select stories whose costs total up to, but do not surpass that budget.
- With these simple rules in place, and with short iterations and frequent releases, it won't be long before the customers and developers get used to the rhythm of the

project. The customers will get a sense for how fast the developers are going. Based on that sense, the customers will be able to define how long their project will take and how much it will cost.

1.2.12 Simple Design

- An XP team makes their designs as simple and expressive as they can be. Also, they narrow their focus to consider only the stories that are scheduled for the present iteration. They don't worry about stories to come. Instead, they migrate the design of the system, from iteration to iteration, to be the best design for the stories that the system presently implements.
- This means that an XP team will perhaps not start with infrastructure. They possibly won't select the database first. They maybe won't select the middleware first. The team's first act will be to get the first batch of stories working in the simplest way possible. The team will only add the infrastructure when a story comes beside that force them to do so.
- The following three XP mantras guide the developer :
  - **Consider the Simplest Thing that Could Perhaps Work**
    - XP teams always try to find the simplest possible design choice for the current batch of stories. If we can make the current stories work with flat files, we might not use a database or EJB. If we can make the present stories work with a simple socket connection, we might not use an ORB or RMI. If we can make the existing stories work without multithreading, we might not contain multithreading. We attempt to consider the simplest way to implement the current stories. Then we choose a solution that is as near to that simplicity as we can practically get.
  - **You Aren't Going to Need It**
    - Yeah, but we know we're going to need that database one day. We know we're going to need an ORB one day. We know we're going to have to support multiple users one day.
    - An XP team seriously contemplates what will happen if they resist the attraction to add infrastructure before it is sternly needed. They start from the hypothesis that they aren't going to need that infrastructure. The team puts in the infrastructure, only if they have proof or at least very convincing evidence, that putting in the infrastructure now will be more cost effective than waiting.

- Once and Only Once
  - XPers don't abide code duplication. Wherever they find it, they remove it. There are numerous sources of code duplication. The most clear are those stretches of code that were taken with a mouse and plopped down in multiple places. When we find those, we remove them by creating a function or a base class. Occasionally two or more algorithms may be remarkably similar, and yet they vary in subtle ways. We turn those into functions or employ the TEMPLATE METHOD pattern. Whatever the source of duplication, once revealed, we won't tolerate it.
  - The finest way to remove redundancy is to create abstractions. After all, if two things are similar, there must be certain abstraction that unifies them. Thus, the act of removing redundancy forces the team to create many abstractions and further decrease coupling.

1.2.13 Refactoring

- Code tends to rot. As we add feature after feature and deal with bug after bug, the structure of the code damages. Left unchecked, this degradation leads to a twisted, unsustainable mess. XP teams contrary this degradation through frequent refactoring. Refactoring is the practice of making a sequence of tiny transformations that improve the structure of the system without disturbing its behavior. Each transformation is trivial, barely worth doing. But together, they syndicate into significant transformations of the design and architecture of the system.
- After every tiny transformation, we run the unit tests to make certain we haven't broken anything. Then we do the following transformation and the next and the next, running the tests after each. In this manner we keep the system working while transforming its design.
- Refactoring is done continuously rather than at the termination of the project, the end of the release, the end of the iteration, or even the end of the day. Refactoring is somewhat we do every hour or every half hour. Refactoring allows us to keep our code as clean, straightforward, and expressive as feasible.

1.2.14 Metaphor

- The most misinterpreted of all the XP practices is metaphor. Because XPers are rationalists at heart, the lack of a solid definition makes us uneasy. Indeed, XP adherents have regularly debated the removal of metaphor as a practise. Nonetheless, metaphor is, in some ways, one of the most significant activities of all.

- Consider a jigsaw puzzle. What method do you use to figure out how the components go together ? Each item, obviously, contacts others and its shape must be absolutely complementary to the ones it comes into contact with. If you were blind but had a good sense of touch, you could put the puzzle together by sifting through every piece and trying it in different positions.
- But there is something more powerful than the shape of the pieces binding the puzzle together. There is a picture. The picture is the real guide. The picture is so influential that if two neighbouring pieces of the picture do not have complementary shapes, then you know that the puzzle maker made an error.
- That is the metaphor. It's the big picture that ties the whole system together. It's the visualization of the system that makes the location and shape of all the individual modules clear. If a module's shape is erratic with the metaphor, then you know it is the module that is wrong.
- Often a metaphor boils down to a system of names. The names provide a terminology for elements in the system and benefit to define their relationships.
- For example, on a system that transmitted text to a screen at 60 characters per second. At that rate, a screen fill could take some time. So we'd permit the program that was producing the text to fill a buffer. When the buffer was full, we would exchange the program out to disk. When the buffer got close to empty, we would switch the program back in and let it run more.

1.2.15 Conclusion

- Extreme programming is a collection of straightforward, concrete approaches that are used into an agile development process. Many teams have followed this technique with positive outcomes.
- XP is a good general-purpose software development technique. Many project teams will be able to work with it in its current state. Many others will be able to adapt it by incorporating or changing practices.

1.3 Planning

- The following is a description of the Extreme Programming planning game (XP). It's comparable to how planning is done in SCRUM, Crystal, feature-driven development and Adaptive Software Development (ADP), among other agile methodologies. None of those procedures, on the other hand, go into as much detail and rigour.

1.3.1 Initial Exploration

- At the start of the project, the developers and customers try to recognize all the really important user stories they can. However, they don't try to identify all user stories. Customers will continue to develop additional user stories as the project progresses. The user stories flow will continue until the project is completed.
- To estimate the stories, the developers collaborate. The figures are relative rather than absolute. On a tale card, we write a number of "points" to represent the story's relative cost. We may not be sure just how much time a story point represents, but we do know that a story with eight points will take double as long as a story with four points.

1.3.1.1 Spiking, Splitting and Velocity

- Stories that are too large or too small are tough to estimate. Developers tend to underrate large stories and overrate small ones. Any story that is too big should be divided into pieces that aren't too big. Any story that is too small should be compound with other small stories.
- For example, consider the story, "Users can securely transfer money into, out of, and between their accounts." This is a big story. Approximating will be hard and probably incorrect. However, we can split it as follow, into many stories that are much easier to estimate :
    - 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.
- When a story is split or merged, it should be re-estimated. It is not clever to simply add or deduct the estimate.
- The main reason to split or merge a story is to get it to a size where estimation is precise. It is not surprising to find that a story estimated at five points breaks up into stories that add up to ten! Ten is the more precise estimate.
- Relative estimates don't tell us the complete size of the stories, so they don't help us determine when to split or merge them. In order to know the true size of a story, we want a factor that we call velocity. If we have an accurate velocity, we can multiply the estimate of any story by the velocity to get the real time estimate for that story.

For example, if our velocity is "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 continues, the measure of velocity will become ever more accurate because we'll be able to measure the number of story points completed per iteration. However, the developers will perhaps not have a very good idea of their velocity at the start. They must create an early guess by whatever means they feel will give the best results. The necessity for accuracy at this point is not mainly grave, so they don't need to spend an inordinate amount of time on it. Often, it is adequate 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.

1.3.2 Release Planning

- Given a velocity, the customers can get an impression of the cost of each of the stories. They also know the business value and priority of each story. This permits them to choose the stories they want done first. This choice is not purely a matter of priority. Somewhat that is important, but also expensive, may be overdue in favor of something that is less significant but much less expensive. Selections like this are business decisions. The business people decide which stories give them the most bangs for the buck.
- 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 choose the stories they want implemented within that release and the rough order in which they want them implemented. The customers cannot select more stories than will fit according to the current velocity. Since the velocity is primarily inaccurate, this selection is crude. But accuracy is not very important at this point in time. The release plan can be attuned as velocity becomes more accurate.

1.3.3 Iteration Planning

- Next, the developers and customers choose an iteration size. This is naturally two weeks long. Once again, the customers select the stories that they want implemented in the first iteration. They cannot pick more stories than will fit according to the current velocity.
- The order of the stories within the iteration is a technical decision. The developers implement the stories in the order that makes the most technical sense. They may work on the stories successively, finishing each one after the next, or they may split up the stories and work on them all simultaneously. It's completely up to them.

- The customers cannot change the stories in the iteration once the iteration has instigated. They are free to change or reorder any other story in the project, but not the ones that the developers are presently working on.
- The iteration ends on the specified date, even if all the stories aren't done. The estimates for all the completed stories are totaled and the velocity for that iteration is calculated. This measure of velocity is then used to plan the next iteration. The rule is very simple. The planned velocity for each iteration is the measured velocity of the preceding iteration. If the team got 31 story points done previous iteration, then they should plan to get 31 story points done in the next. Their velocity is 31 points per iteration.
- This feedback of velocity aids to keep the planning in sync with the team. If the team gains in expertise and skill, the velocity will increase commensurately. If somebody is lost from the team, the velocity will fall. If an architecture progresses that facilitates development, the velocity will rise.

1.3.4 Task Planning

- At the start of a new iteration, the developers and customers get together to plan. The developers break the stories down into development tasks. A task is somewhat that one developer can implement in 4 - 16 hours. The stories are analyzed, with the customers' help and the tasks are enumerated as fully as possible.
- A list of the tasks is created on a flip chart, whiteboard or some other suitable medium. Then, one by one, the developers sign up for the tasks they want to implement. As each developer signs up for a task, he or she estimates that task in arbitrary task points.
- Developers may sign up for any kind of task. Database guys are not constrained to sign up for database tasks. GUI guys can sign up for database tasks if they like. This may appear inefficient, but as you'll see, there is a mechanism that succeeds this. The advantage is obvious. The more the developers know about the whole project, the healthier and more informed the project team is. We need knowledge of the project to spread through the team regardless of specialty.
- Each developer knows how many task points he or she accomplished to implement in the last iteration. This number is their personal budget. No one signs up for more points than they have in their budget.
- Task selection continues until either all tasks are allotted or all developers have used their budgets. If there are tasks remaining, then the developers negotiate with

each other, trading tasks based on their various skills. If this doesn't make enough room to get all the tasks assigned, then the developers ask the customers to eliminate tasks or stories from the iteration. If all the tasks are signed up and the developers still have room in their budgets for more work, they ask the customers for more stories.

1.3.4.1 Halfway Point

- Halfway through the iteration, the team holds a meeting. At this point, half of the stories planned for the iteration should be complete. If half the stories aren't complete, then the team tries to reapportion tasks and responsibilities to guarantee that all the stories will be complete by the end of the iteration. If the developers cannot find such a reapportionment, then the customers need to be told. The customers may decide to pull a task or story from the iteration. At the very least, they will name the lowermost priority tasks and stories so that the developers avoid working on them.
- For example, suppose the customers selected eight stories totaling 24 story points for the iteration. Suppose also that these were broken down into 42 tasks. At the halfway point of the iteration, we would expect to have 21 tasks and 12 story points complete. Those 12 story points must represent solely completed stories. Our aim is to complete stories, not just tasks. The frightening scenario is to get to the end of the iteration with 90 % of the tasks complete, but no stories complete. At the middle point, we want to see completed stories that represent half the story points for the iteration.

1.3.5 Iterating

- Every two weeks, the current iteration ends and the next begin. At the end of each iteration, the present running executable is demonstrated to the customers. The customers are asked to evaluate the look, feel and performance of the project. They will deliver their feedback in terms of new user stories.
- The customers see progress regularly. They can measure velocity. They can forecast how fast the team is going, and they can schedule high-priority stories early. In short, they have all the data and control they need to manage the project to their liking.

1.3.6 Conclusion

- From iteration to iteration and release to release, the project falls into a predictable

and comfortable rhythm. Everyone knows what to expect and when to expect it. Stakeholders see development often and significantly. Rather than being presented notebooks full of diagrams and plans, they are revealed working software that they can touch, feel, and provide feedback on.

- Developers see a rational plan based upon their own estimates and controlled by their own measured velocity. They select the tasks on which they feel contented working and keep the quality of their workmanship high.
- Managers receive data every iteration. They use this data to control and manage the project. They don't have to resort to pressure, threats, or appeals to reliability to meet a random and improbable date.
- The stakeholders won't continually be happy with the data that the process produces, particularly not at first. Using an agile method does not mean that the stakeholders will get what they want. It simply means that they will be able to control the team to get the utmost business value for the least cost.

1.4 Testing

- The act of writing a unit test is added an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test ends a remarkable number of feedback loops, the slightest of which is the one relating to verification of function.

1.4.1 Test Driven Development

- What if we designed our tests earlier we designed our programs ? What if we declined to implement a function in our programs until there was a test that failed because that function wasn't present ? What if we declined to add even a single line of code to our programs unless there was a test that was failing because of its absence ? What if we incrementally added functionality to our programs by first writing failing tests that proclaimed the existence of that functionality, and then made the test pass ? What outcome would this have on the design of the software we were writing ? What welfare would we derive from the existence of such a comprehensive bevy of tests ?
- The first and most pure effect is that every single function of the program has tests that verify its operation. This set of tests acts as a backstop for further development. It tells us every time we unintentionally break some existing functionality. We can add functions to the program, or change the structure of the program, without anxiety that we will break something important in the process. The tests tell us that

the program is still behaving properly. We are thus much freer to make changes and improvement to our program.

- A more important, but less noticeable, effect is that the act of writing the test first forces us into a diverse point of view. We must view the program we are about to write from the vantage point of a caller of that program. Thus, we are instantly concerned with the interface of the program as well as its function. By writing the test first, we design the software to be suitably callable.
- What's more, by writing the test first, we force ourselves to design the program to be testable. Designing the program to be callable and testable is unusually important. In order to be callable and testable, the software has to be decoupled from its environments. Thus, the act of writing tests first forces us to decouple the software!
- Another significant effect of writing tests first is that the tests act as a priceless form of documentation. If you want to know how to call a function or create an object, there is a test that shows you. The tests act as a suite of examples that help other programmers figure out how to work with the code. This documentation is compileable and executable. It will stay up-to-date. It cannot lie.

1.4.1.1 Example of Test - First Design

- Hunt the Wumpus, this program is a simple adventure game in which the player moves through a cave trying to kill the Wumpus before the Wumpus eats him. The cave is a set of rooms that are linked to each other by passageways. Each room may have passages to the north, south, east, or west. The player moves about by telling the computer which track to go.
- One of the leading tests for this program was moveTest in Listing - 1. This function creates a new WumpusGame, connects room 4 to room 5 via an east passage, places the player in room 4, issues the command to move east, and then proclaims that the player should be in room 5.

Listing - 1

```
public void moveTest()
{
    WumpusGame wg = new WumpusGame();
    wg.connect(4,5,"E");
    wg.setPlayerRoom(4);
    wg.east();
    assertEquals(5, wg.getPlayerRoom());
}
```

- All this code was written before any part of WumpusGame was written. By using Ward Cunningham's advice and the test the way we wanted it to read.
- The test pass by writing the code that adapted to the structure inferred by the test. This is called intentional programming. You state your intent in a test before you implement it, making your intent as simple and readable as possible. You trust that this easiness and clarity points to a good structure for the program.
- Programming by intent instantly led to an interesting design decision. The test makes no use of a Room class. The act of connecting one room to another communicates intent. We don't seem to need a Room class to simplify that communication. Instead, we can just use integers to represent the rooms.
- This may seem counter instinctive to you. After all, this program may seem to you to be all about rooms; moving between rooms; finding out what rooms contain; etc. Is the design implicit by our intent flawed because it lacks a Room class ?
- We could dispute that the concept of connections is remote more central to the Wumpus game than the concept of room. We could argue that this initial test pointed out a good way to solve the problem. Certainly, I think that is the case, but it is not the point we are trying to make. 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 discriminating between design decisions.
- Notice that the test tells you how the program works. Most of us could simply write the four named methods of WumpusGame from this simple description. We could also name and write the three other direction commands without much concern. If later we want to know how to connect two rooms or move in a specific direction, this test will show us how to do it in no indefinite terms. This test acts as a compileable and executable document that defines the program.

1.4.1.2 Test Isolation

- The act of writing tests before production code regularly exposes areas in the software that ought to be decoupled. For example, Fig. 1.4.1 shows a simple UML diagram1 of a payroll application. The payroll class uses the EmployeeDatabase class to fetch an Employee object. It requests the Employee to calculate its pay. Then it passes that pay to the CheckWriter object to yield a check. Lastly, it posts the payment to the Employee object and writes the object back to the database.

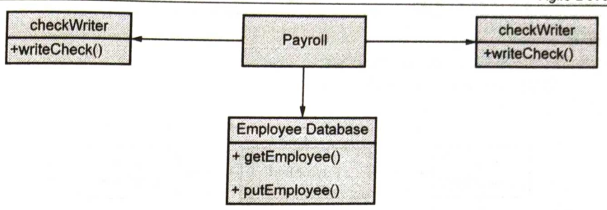

Fig. 1.4.1 Coupled payroll model

- Believe that we haven't written any of this code yet. So far, this diagram is just sitting on a whiteboard after a quick design session. Now, we want to write the tests that stipulate the behavior of the Payroll object. There are a number of problems related with writing these tests. First, what database do we use ? Payroll needs to read from some kind of database. Must we write a fully functioning database before we can test the Payroll class ?
- What data do we load into it ? Second, how do we verify that the proper check got printed ? We can't write an automated test that looks on the printer for a check and confirms the amount on it !
- The answer to these problems is to use the MOCK OBJECT pattern. We can insert interfaces among all the collaborators of Payroll and create test stubs that implement these interfaces.
- Fig. 1.4.2 shows the structure. The Payroll class now uses interfaces to communicate with the EmployeeDatabase, CheckWriter and Employee. Three MOCK OBJECTS have been created that implement these interfaces. These MOCK OBJECTS are inquired by the PayrollTest object to get if the Payroll object achieves them properly. (See Fig. 1.4.2 on next page).
- Listing - 2 shows the intent of the test. It creates the suitable mock objects, passes them to the Payroll object, tells the payroll object to pay all the employees and then asks the mock objects to confirm that all the checks were written properly and that all the payments were posted appropriately.

Listing 1 - 2 Payroll Test

```
public void payrollTest()
{
    MockEmployeeDatabase medb = new MockEmployeeDatabase();
    MockCheckWriter mcw = new MockCheckWriter();
    Payroll p = new Payroll(medb, mcw);
```

```
    p.payEmployees();
    assert(mcw.checksWereWrittenCorrectly());
    assert(medb.paymentsWerePostedCorrectly());
}
```

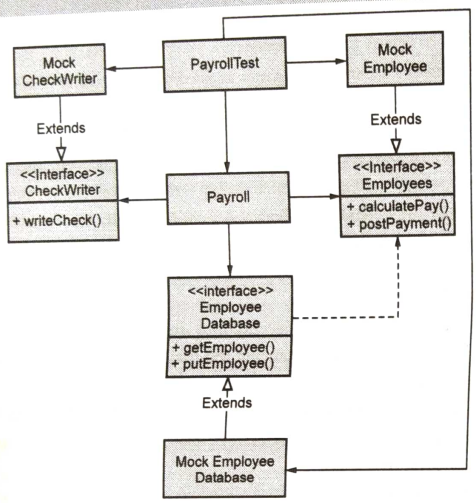

Fig. 1.4.2 Decoupled payroll using mock objects for testing

- Of course all this test is testing is that Payroll called all the correct functions with all the right data. It's not really checking that checks were written. It's not truly checking that a true database was properly updated. Rather, it is checking that the Payroll class is acting as it should in isolation.
- You might wonder what the MockEmployee is for. It looks feasible that the real Employee class could be used in its place of a mock. If that were so, then we would have no regret about using it. In this case, we supposed that the Employee class was more complex than needed to check the function of Payroll.

1.4.1.3 Serendipitous Decoupling

- The decoupling of Payroll is a good thing. It permits us to exchange in different databases and check writers, both for the purpose of testing and for extension of the application. I think it is interesting that this decoupling was driven by the need to

test. Seemingly, the need to isolate the module under test forces us to decouple in ways that are useful to the overall structure of the program. Writing tests before code improves our designs.

- Design principles for managing dependencies are the main part in agile development. Those principles give you some rules and techniques for decoupling classes and packages. You will find these principles most useful if you practice them as part of your unit testing policy. It is the unit tests that will provide much of the impetus and direction for decoupling.

1.4.2 Acceptance Tests

- Unit tests are essential but inadequate as verification tools. Unit tests confirm that the small elements of the system work as they are expected to, but they do not prove that the system works properly as a whole. Unit tests are white-box tests that verify the separate mechanisms of the system. Acceptance tests are black-box tests that confirm that the customer requirements are being encountered.
- Acceptance tests are written by people who do not know the internal mechanisms of the system. They may be written straight by the customer or by some technical people close to the customer, possibly QA. Acceptance tests are programs and are so executable. However, they are typically written in a special scripting language created for customers of the application.
- Acceptance tests are the decisive documentation of a feature. Once the customer has written the acceptance tests, which confirm that a feature is correct, the programmers can read those acceptance tests to really understand the feature. So, just as unit tests aid as compileable and executable documentation for the internals of the system, acceptance tests help as compileable and executable documentation of the features of the system.
- Additionally, the act of writing acceptance tests first has a reflective effect upon the architecture of the system. In order to make the system testable, it has to be decoupled at the high architecture level. For example, the User Interface (UI) has to be decoupled from the business rules in such a way that the acceptance tests can increase access to those business rules without going through the UI.
- In the initial iterations of a project, the attraction is to do acceptance tests manually. This is inadvisable because it withdraws those early iterations of the decoupling pressure exerted by the need to automate the acceptance tests. When you start the very first iteration, knowing full well that you must mechanize the acceptance tests,

you make very dissimilar architectural trade-offs. And, just as unit tests drive you to make superior design decisions in the small, acceptance tests drive you to make greater architecture decisions in the large.

- Creating an acceptance testing framework may look a daunting task. However, if you take only one iteration's worth of features and create only that part of the framework essential for those few acceptance tests, you will find it's not that hard to write. You will also find that the effort is worth the cost.

1.4.2.1 Example of Acceptance Testing

- Consider, again, the payroll application. In our first iteration, 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 presently in the database. Luckily, we only have to deal with salaried employees. The other kinds of employees have been detained back until a future iteration.
- We haven't written any code yet and we haven't capitalized in any design yet. This is the best time to start thinking about acceptance tests. Once again, intentional programming is a valuable tool. We should write the acceptance tests the way we think they should look, and then we can structure the scripting language and payroll system around that structure.
- We want to make the acceptance tests suitable to write and easy to change. And also need them to be placed in a configuration-management tool and saved so that we can run them anytime we please. Therefore, it makes sense that the acceptance tests should be written in simple text files.
- The following is an example of an acceptance-test script :

  AddEmp 1429 "Ramesh Mathur" 3215.88

  Payday

  Verify Paycheck EmpId 1429 GrossPay 3215.88

- In this example, we add employee number 1429 to the database. His name is "Ramesh Mathur," and his monthly pay is Rs.3215.88. Following, we tell the system that it is payday and that it requests to pay all the employees. Lastly, we confirm that a paycheck was generated for employee 1429 with a GrossPay field of Rs.3215.88.
- Clearly, this kind of script will be very informal for customers to write. Also, it will be easy to add new functionality to this kind of script. However, think about what it infers about the structure of the system.

- The first two lines of the script are functions of the payroll application. We might call these lines payroll transactions. These are functions that payroll users expect. However, the Verify line is not a transaction that the users of payroll would imagine. This line is a directive that is precise to the acceptance test.
- Thus, our acceptance testing framework will have to analyse this text file, separating the payroll transactions from the acceptance-testing directives. It must send the payroll transactions to the payroll application and then use the acceptance-testing directives to query the payroll application in order to confirm data.
- This already puts architectural stress on the payroll program. The payroll program is going to have to accept input straight from users and also from the acceptance-testing framework. We need to bring those two paths of input together as early as possible. So, it looks as if the payroll program will need a transaction processor that can deal with transactions of the form AddEmp and Payday coming from more than one source. We need to find some mutual form for those transactions so that the amount of specialized code is kept to a minimum.
- One answer would be to feed the transactions into the payroll application in XML. The acceptance-testing framework could definitely generate XML, and it appears likely that the UI of the payroll system could also generate XML. Thus, we might see transactions that looked like the following :

```
<AddEmp PayType=Salaried>
    <EmpId>1429</EmpId>
    <Name>Ramesh Mathur</Name>
    <Salary>3215.88</Salary>
</AddEmp>
```

- These transactions might enter the payroll application through a subroutine call, a socket, or even a batch input file. Certainly, it would be a trivial matter to change from one to the other during the course of development. So through the early iterations, we could decide to read transactions from a file, drifting to an API or socket much advanced.
- We can have the payroll application produce its paychecks in XML. The acceptance-testing framework can then catch this XML and query it for the suitable data. The final step of printing the check from the XML may be trivial sufficient to handle through manual acceptance tests.

Therefore, the payroll application can create an XML document that covers all the paychecks. It might look like this :

```
<Paycheck>
    <EmpId>1429</EmpId>
    <Name>Ramesh Mathur</Name>
    <GrossPay>3215.88</GrossPay>
</Paycheck>
```

- Clearly, the acceptance-testing framework can execute the Authenticate directive when supplied with this XML. Once again, we can spit the XML out through a socket, through an API, or into a file. For the early iterations, a file is probably easiest. Therefore, the payroll application will instigate its life reading XML transactions in from a file and outputting XML paychecks to a file. The acceptance-testing framework will read transactions in text form, translating them to XML and writing them to a file. It will then invoke the payroll program. Lastly, it will read the output XML from the payroll program and invoke the validate directives.

1.4.2.2 Serendipitous Architecture

- Notice the burden that the acceptance tests placed upon the architecture of the payroll system. The actual fact that we considered the tests first led us to the idea of XML input and output very quickly. This architecture has decoupled the transaction sources from the payroll application. It has also decoupled the paycheck printing mechanism from the payroll application. These are decent architectural decisions.

1.4.3 Conclusion

- The simpler it is to run a suite of tests, the more regularly those tests will be run. The more the tests are run, the faster any deviation from those tests will be found. If we can run all the tests numerous times a day, then the system will never be broken for more than a few minutes. This is a rational goal. We simply don't let the system to backslide.
- Once it works to a certain level, it never backslides to a lower level. Yet confirmation is just one of the benefits of writing tests. Both unit tests and acceptance tests are a form of documentation. That documentation is compileable and executable; therefore, it is precise and trustworthy.
- Moreover, these tests are written in unmistakable languages that are made to be legible by their audience. Programmers can read unit tests because they are written in their programming language. Customers can read acceptance tests because they are written in a language that they themselves designed.
- Perhaps the most important advantage of all this testing is the impact it has on

architecture and design. To make a module or an application testable, it must also be decoupled. The more testable it is, the more decoupled it is. The performance of considering comprehensive acceptance and unit tests has a deeply positive effect upon the structure of the software.

1.5 Refactoring

- It is about paying attention to what you are doing and making sure you are doing your best. It is about the variance between getting something to work and getting something right. It is about the value we place in the structure of our code.
- Martin Fowler describes refactoring as "the process of changing a software system in such a way that it does not change the external behavior of the code yet recovers its internal structure." But why would we need to improve the structure of working code ? What about the old saw, "if it's not broken, don't fix it!" ?
- Every software module has three functions. First, there is the function it performs while executing. This function is the cause for the module's existence. The second function of a module is to afford change. Almost all modules will change in the course of their lives, and it is the accountability of the developers to make sure that such changes are as simple as possible to make. A module that is stiff to change is broken and needs fixing, even though it works. The third function of a module is to communicate to its readers. Developers unacquainted with the module should be able to read and understand it without undue menta gymnastics. A module that does not communicate is broken and wants to be fixed.

1.5.1 Generating Primes : A Simple Example of Refactoring

- Consider the code in Listing - 3. This program generates prime numbers. It is one big function with many single letter variables and comments to help us read it.

Listing - 3

```
GeneratePrimes.java version 1

import java.util.*;
public class GeneratePrimes
{
public static int[] generatePrimes(int maxValue)
{
    if (maxValue >= 2) // the only valid case
    {
        // declarations
        int s = maxValue + 1; // size of array
```

```
        boolean[] f = new boolean[s];
        int i;

    // initialize array to true.
    for (i = 0; i < s; i++)
      f[i] = true;

    // get rid of known non-primes
    f[0] = f[1] = false;

    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
            for (j = 2 * i; j < s; j += i)
              f[j] = false; // multiple is not prime
        }
    }

    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
        if (f[i])
            count++; // bump count.
    }

    int[] primes = new int[count];

    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
        if (f[i]) // if prime
            primes[j++] = i;
    }
    return primes; // return the primes
    }
    else // maxValue < 2
        return new int[0]; // return null array if bad input.
  }
}
```

- The unit test for GeneratePrimes is shown in Listing 4. It takes a statistical method, checking to see if the generator can generate primes up to 0, 2, 3, and 100. In the first

case there should be no primes. In the second there should be one prime and it should be 2. In the third there should be two primes, and they should be 2 and 3. In the last case there should be 25 primes, the last of which is 97. If all these tests pass, then make the hypothesis that the generator is working. This is not foolproof, but we can't think of a realistic scenario where these tests would pass and yet the function would fail.

Listing - 4

GeneratePrimesTest.java

```
import junit.framework.*;
import java.util.*;

public class GeneratePrimesTest extends TestCase
{
    public static void main(String args[])
    {
        junit.swingui.TestRunner.main(
        new String[] {"GeneratePrimesTest"});
    }
    public GeneratePrimesTest(String name)
    {
        super(name);
    }
    public void testPrimes()
    {
        int[] nullArray = GeneratePrimes.generatePrimes(0);
        assertEquals(nullArray.length, 0);

        int[] minArray = GeneratePrimes.generatePrimes(2);
        assertEquals(minArray.length, 1);
        assertEquals(minArray[0], 2);

        int[] threeArray = GeneratePrimes.generatePrimes(3);
        assertEquals(threeArray.length, 2);
        assertEquals(threeArray[0], 2);
        assertEquals(threeArray[1], 3);

        int[] centArray = GeneratePrimes.generatePrimes(100);
        assertEquals(centArray.length, 25);
        assertEquals(centArray[24], 97);
    }
}
```

- To refactor this program, we can use the Idea refactoring browser from IntelliJ. This tool makes it trivial to excerpt methods and rename variables and classes.
- It looks pretty clear that the main function wants to be three separate functions. The first initializes all the variables and sets up the sieve. The second really executes the sieve, and the third loads the sieved results into an integer array. To rendering this structure more clearly in Listing 5, we extracted those functions into three distinct methods. We also removed a few needless comments and changed the name of the class to PrimeGenerator .The tests all still ran.
- Extracting the three functions enforced us to promote some of the variables of the function to static fields of the class.

Listing - 5

```
PrimeGenerator.java, version 2
import java.util.*;
public class PrimeGenerator
{
   private static int s;
   private static boolean[] f;
   private static int[] primes;

   public static int[] generatePrimes(int maxValue)
   {
      if (maxValue < 2)
      return new int[0];
      else
      {
          initializeSieve(maxValue);
          sieve();
          loadPrimes();
          return primes; // return the primes
      }
   }
   private static void loadPrimes()
   {
       int i;
       int j;

   // how many primes are there?
   int count = 0;
   for (i = 0; i < s; i++)
   {
       if (f[i])
```



```
            count++; // bump count.
        }

        primes = new int[count];

    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
    if (f[i]) // if prime
        primes[j++] = i;
    }
    }
private static void sieve()
{
        int i;
        int j;
        for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
        if (f[i]) // if i is uncrossed, cross out its multiples.
            {
              for (j = 2 * i; j < s; j += i)
                  f[j] = false; // multiple is not prime
            }
    }
}
private static void initializeSieve(int maxValue)
    {
        // declarations
        s = maxValue + 1; // size of array
        f = new boolean[s];
        int i;
        // initialize array to true.
        for (i = 0; i < s; i++)
          f[i] = true;
        // get rid of known non-primes
        f[0] = f[1] = false;
    }
}
```

- The initializeSieve function is a slight messy, so in Listing - 6, we cleaned it up considerably. First, substituted all usages of the s variable with f.length. Then, Ichanged the names of the three functions to somewhat a bit more expressive. Finally, rearranged the innards of initializeArrayOfIntegers to be a slight nicer to read. The tests all still ran.

Listing - 6

PrimeGenerator.java, version 3 (partial)

```
public class PrimeGenerator
{
    private static boolean[] f;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
        {
          if (maxValue < 2)
          return new int[0];
          else
            {
                initializeArrayOfIntegers(maxValue);
                crossOutMultiples();
                putUncrossedIntegersIntoResult();
                return result;
            }
        }
    private static void initializeArrayOfIntegers(int maxValue)
    {
        f = new boolean[maxValue + 1];
        f[0] = f[1] = false; //neither primes nor multiples.
        for (int i = 2; i < f.length; i++)
        f[i] = true;
    }
}
```

- Next, saw at crossOutMultiples. There were a number of statements in this function, and in others, of the form if (f[i] == true). The intent was to check to see if it was uncrossed, so we changed the name off to unCrossed. But this lead to horrible statements like unCrossed[i] = false. We found the double negative unclear. So we altered the name of the array to isCrossed and altered the sense of all the booleans. The tests all still ran.
- We got rid of the initialization that set isCrossed[0] and isCrossed[1] to true and just made certain that no part of the function used the isCrossed array for indexes less than 2. Also extracted the inner loop of the crossOutMultiples function and called it crossOutMultiplesOf. We also thought that if (isCrossed[i] == false) was confusing, so we created a function called notCrossed and altered the if statement to if (notCrossed(i)). The tests all still ran.
- The result of all these refactorings is in Listing - 7. The tests all still ran.

Listing - 7

PrimeGenerator.java version 4 (partial)

```
public class PrimeGenerator
{
        private static boolean[] isCrossed;
        private static int[] result;

            public static int[] generatePrimes(int maxValue)
        {
            if (maxValue < 2)
                return new int[0];
            else
            {
                initializeArrayOfIntegers(maxValue);
                crossOutMultiples();
                putUncrossedIntegersIntoResult();
                return result;
            }
        }
    private static void initializeArrayOfIntegers(int maxValue)
    {
            isCrossed = new boolean[maxValue + 1];
            for (int i = 2; i < isCrossed.length; i++)
                isCrossed[i] = false;
    }
    private static void crossOutMultiples()
    {
        int maxPrimeFactor = calcMaxPrimeFactor();
        for (int i = 2; i <= maxPrimeFactor; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }
    private static int calcMaxPrimeFactor()
    {
        // We cross out all multiples of p, where p is prime.
        // Thus, all crossed out multiples have p and q for
        // factors. If p > sqrt of the size of the array, then
        // q will never be greater than 1. Thus p is the
        // largest prime factor in the array, and is also
        // the iteration limit.
        double maxPrimeFactor = Math.sqrt(isCrossed.length) + 1;
        return (int) maxPrimeFactor;
    }
    private static void crossOutMultiplesOf(int i)
```

```
    {
        for (int multiple = 2*i;
            multiple < isCrossed.length;
            multiple += i)
        isCrossed[multiple] = true;
    }
    private static boolean notCrossed(int i)
    {
        return isCrossed[i] == false;
    }
```

- The last function to refactor is putUncrossedIntegersIntoResult. This technique has two parts. The first counts the number of uncrossed integers in the array and creates the result array of that size. The second transfers the uncrossed integers into the result array.

Listing - 8

PrimeGenerator.java, version 5 (partial)

```
private static void putUncrossedIntegersIntoResult()
{
        result = new int[numberOfUncrossedIntegers()];
        for (int j = 0, i = 2; i < isCrossed.length; i++)
            if (notCrossed(i))
                result[j++] = i;
}
private static int numberOfUncrossedIntegers()
{
        int count = 0;
        for (int i = 2; i < isCrossed.length; i++)
            if (notCrossed(i))
                count++;
        return count;
}
```

1.5.1.1 The Final Reread

- Next, one final pass over the whole program, reading it from start to end, rather like one would read a regular proof. This is an important step. So far, we have been refactoring fragments.
- First, we don't like the name initializeArrayOfIntegers. What's being initialized is not, in fact, an array of integers; it's an array of booleans. However, initializeArrayOfBooleans is not an enhancement. What we are really doing in this method is uncrossing all the relevant integers so that we can then cross out the

multiples. So change the name to uncrossIntegersUpTo. And also the name isCrossed for the array of booleans, change it to crossedOut. The tests all still run.

- The rest of the code reads pretty well. So we're done. The final version is shown in Listings - 9 and 10.

Listing - 9

PrimeGenerator.java (final)

```
public class PrimeGenerator
{
        private static boolean[] crossedOut;
        private static int[] result;

        public static int[] generatePrimes(int maxValue)
        {
                if (maxValue < 2)
                        return new int[0];
                else
                {
                        uncrossIntegersUpTo(maxValue);
                        crossOutMultiples();
                        putUncrossedIntegersIntoResult();
                        return result;
                }
        }

    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
                        crossedOut[i] = false;
    }
    private static void crossOutMultiples()
    {
            int limit = determineIterationLimit();
            for (int i = 2; i <= limit; i++)
                    if (notCrossed(i))
                            crossOutMultiplesOf(i);
    }
    private static int determineIterationLimit()
    {
            // Every multiple in the array has a prime factor that
            // is less than or equal to the sqrt of the array size,
            // so we don't have to cross out multiples of numbers
```

```
        // larger than that root.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }
    private static void crossOutMultiplesOf(int i)
    {
        for (int multiple = 2*i;
                multiple < crossedOut.length;
                multiple += i)
        crossedOut[multiple] = true;
    }
    private static boolean notCrossed(int i)
    {
        return crossedOut[i] == false;
    }
    private static void putUncrossedIntegersIntoResult()
    {
        result = new int[numberOfUncrossedIntegers()];
        for (int j = 0, i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
            result[j++] = i;
    }
    private static int numberOfUncrossedIntegers()
    {
        int count = 0;
        for (int i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
                count++;
        return count;
    }
}
```

Listing - 10

TestGeneratePrimes.java (final)

```
import junit.framework.*;
public class TestGeneratePrimes extends TestCase
{
    public static void main(String args[])
    {
        junit.swingui.TestRunner.main(
        new String[] {"TestGeneratePrimes"});
    }
    public TestGeneratePrimes(String name)
    {
        super(name);
    }
    public void testPrimes()
    {
        int[] nullArray = PrimeGenerator.generatePrimes(0);
        assertEquals(nullArray.length, 0);

        int[] minArray = PrimeGenerator.generatePrimes(2);
        assertEquals(minArray.length, 1);
        assertEquals(minArray[0], 2);

        int[] threeArray = PrimeGenerator.generatePrimes(3);
        assertEquals(threeArray.length, 2);
        assertEquals(threeArray[0], 2);
        assertEquals(threeArray[1], 3);

        int[] centArray = PrimeGenerator.generatePrimes(100);
        assertEquals(centArray.length, 25);
        assertEquals(centArray[24], 97);
    }
    public void testExhaustive()
    {
        for (int i = 2; i<500; i++)
            verifyPrimeList(PrimeGenerator.generatePrimes(i));
    }
    private void verifyPrimeList(int[] list)
    {
        for (int i=0; i<list.length; i++)
            verifyPrime(list[i]);
    }
    private void verifyPrime(int n)
    {
        for (int factor=2; factor<n; factor++)
            assert(n%factor != 0);
    }
}
```


1.5.2 Conclusion

- The end result of this program reads much improved than it did at the start. The program also works a bit improved. The program is much easier to understand and is therefore much easier to change.
- Also, the structure of the program has inaccessible its parts from one another. This also makes the program much easier to change. The increased readability is

substance a few extra nanoseconds in most cases. However, there may be deep inner loops where those few nanoseconds will be expensive.

- It is strongly recommended that constantly practice such refactoring for every module you write and for every module you maintain. The time investment is very minor compared to the effort you will be saving yourself and others in the near future.
- Refactoring is like cleaning up the kitchen after dinner. The first time you skip it, you are done with dinner more rapidly. But that absence of clean dishes and clear working space makes dinner take longer to prepare the next day. This makes you want to skip cleaning again. Certainly, you can always finish dinner quicker today if you skip cleaning, but the mess builds and builds. Finally you are spending an excessive amount of time hunting for the right cooking utensils, chiseling the encrusted dried food off the dishes, and scrubbing them down so that they are appropriate to cook with. Dinner takes forever. Skipping the cleanup does not really make dinner go faster.
- The goal of refactoring, as depicted in this chapter, is to clean your code each day. We don't want the mess to build. We don't want to have to chisel and scrub the coated bits that accumulate over time. We want to be able to spread and modify our system with a minimum of effort. The most significant enabler of that ability is the cleanliness of the code.

1.6 Review Questions

1. *List and explain manifesto for agile software development.* **(Refer section 1.1.1)**
2. *Explain 12 agile principles in detail.* **(Refer section 1.1.2)**
3. *What is extreme programming ? Write short note on user stories.* **(Refer section 1.2)**
4. *Explain short cycles in XP.* **(Refer section 1.2)**
5. *Describe about integration testing and pair programming.* **(Refer section 1.2.5)**
6. *Describe the planning and estimation strategies used in extreme programming.* **(Refer section 1.3)**
7. *Explain refactoring with the help of examples.* **(Refer section 1.2.13)**