AGILE DEVELOPMENT AND UI/UX DESIGN

SUBJECT CODE: 3171610

UNIT -1

AGILE DEVELOPMENT METHODS:

PHILOSOPHY AND PRACTICE

History of Agile Methods

- Particularly in 1990s, some developers reacted against traditional “heavyweight” software development processes.
- New methods were being developed and tested,
  - e.g. extreme programming, SCRUM, Feature-driven development
  - Generally termed “light” processes
- “Representatives” from several of these methods got together in Utah in 2001
  - Settled on term “Agile” as a way to describe these methods
  - Called themselves the “Agile Alliance”
  - Developed a “manifesto” and a statement of “principles”
  - Focuses on common themes in these alternative methodologies

History of Agile Methods

- Particularly in 1990s, some developers reacted against traditional “heavyweight” software development processes.
- New methods were being developed and tested,
  - e.g. extreme programming, SCRUM, Feature-driven development
  - Generally termed “light” processes
- “Representatives” from several of these methods got together in Utah in 2001
  - Settled on term “Agile” as a way to describe these methods
  - Called themselves the “Agile Alliance”
  - Developed a “manifesto” and a statement of “principles”
  - Focuses on common themes in these alternative methodologies

Manifesto for Agile Software Development

We are uncovering 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

12 Principles behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

12 Principles behind the Agile Manifesto

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

12 Principles behind the Agile Manifesto

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.

12 Principles behind the Agile Manifesto

10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Individuals and Interactions over Processes and Tools

- People make biggest impact on success
  - Process and environment help, but will not create success
- Strong individuals not enough without good team interaction.
  - Individuals may be stronger based on their ability to work on a team
- Tools can help, but bigger and better tools can hinder more than help
  - Simpler tools can be better

Working Software over Comprehensive Documentation

- Documentation important, but too much is worse than too little
  - Long time to produce, keep in sync with code
  - Keep documents short and salient
- Focus effort on producing code, not descriptions of it
  - Code should document itself
  - Knowledge of code kept within the team
- *Produce no document unless its need is immediate and significant.*

Customer Collaboration over Contract Negotiation

- Not reasonable to specify what's needed and then have no more contact until final product delivered
- Get regular customer feedback
- Use contracts to specify customer interaction rather than requirements, schedule, and cost

Responding to Change over Following a Plan

- Environment, requirements, and estimates of work required will change over course of large project.
- Planning out a whole project doesn't hold up
  - Changes in shape, not just in time
- Keep planning realistic
  - Know tasks for next couple of weeks
  - Rough idea of requirements to work on next few months
  - Vague sense of what needs to be done over year

Extreme Programming (XP)

- One of the most well-known agile methods
- Developed in 1990s
  - Kent Beck, 1996
  - Chrysler Comprehensive Compensation Project
  - Published book in 1999

Extreme Programming Practices

1. On-Site Customer
   - Customer is actively involved with development process
   - Customer gives “User Stories”
     - Short, informal “stories” describing features
     - Keep on “story cards”
2. Planning Game
   - Developers and customers together plan project
   - Developers give cost estimates to “stories” and a budget of how much they can accomplish
     - Can use abstract accounting mechanism
     - Later compare to actual cost, to improve estimates over time
   - Customer prioritizes stories to fit within budget

Extreme Programming Practices

3. Metaphor
   - Come up with metaphor that describes how the whole project will fit together
   - The picture in a jigsaw puzzle
   - Provides framework for discussing project in team
   - Tools and materials often provide good metaphors
4. Small Releases
   - Time between releases drastically reduced
     - A few weeks/months
   - Multiple iterations
   - Can have intermediate iterations between bigger “releases”

Extreme Programming Practices

5. Testing
   - Test-first programming
   - Unit testing frequently by developers
   - Acceptance tests defined by customers
6. Simple Design
   - Principles discussed in earlier lectures
   - Design should be quick, not elaborate
   - Pick the simplest thing that could possibly work
   - Resist adding stuff not ready yet

Extreme Programming Practices

7. Refactoring
   - Code gets worse with feature adds, bug fixes
   - Rewrite small sections of code *regularly*
   - Rerun all unit tests to know nothing broken
     - Means you should have designed comprehensive tests
8. Pair Programming
   - Discussed later
9. Collective Ownership
   - Anyone can edit anything
   - Errors are the fault of the whole team

Extreme Programming Practices

10. Continuous Integration
    - Commit changes frequently (several times a day)
    - Verify against entire test suite!
11. Coding Standards
    - Enables effective teamwork
12. Sustainable Pace
    - No overtime
    - Only exceptions in final week
    - Good estimation skills for budgeting will help ensure reasonable times
    - Time less likely to be wasted in pairs, bullpen rooms
    - Plan time each day for administrative work (<1 hour), breaks

SCRUM

- Idea first appeared in a business journal in 1986 (applied to product development management).
- Used in software development and presented in 1995 paper.
- Term is based on rugby term
- Small cross-functional teams

SCRUM Practices

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

SCRUM Practices

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

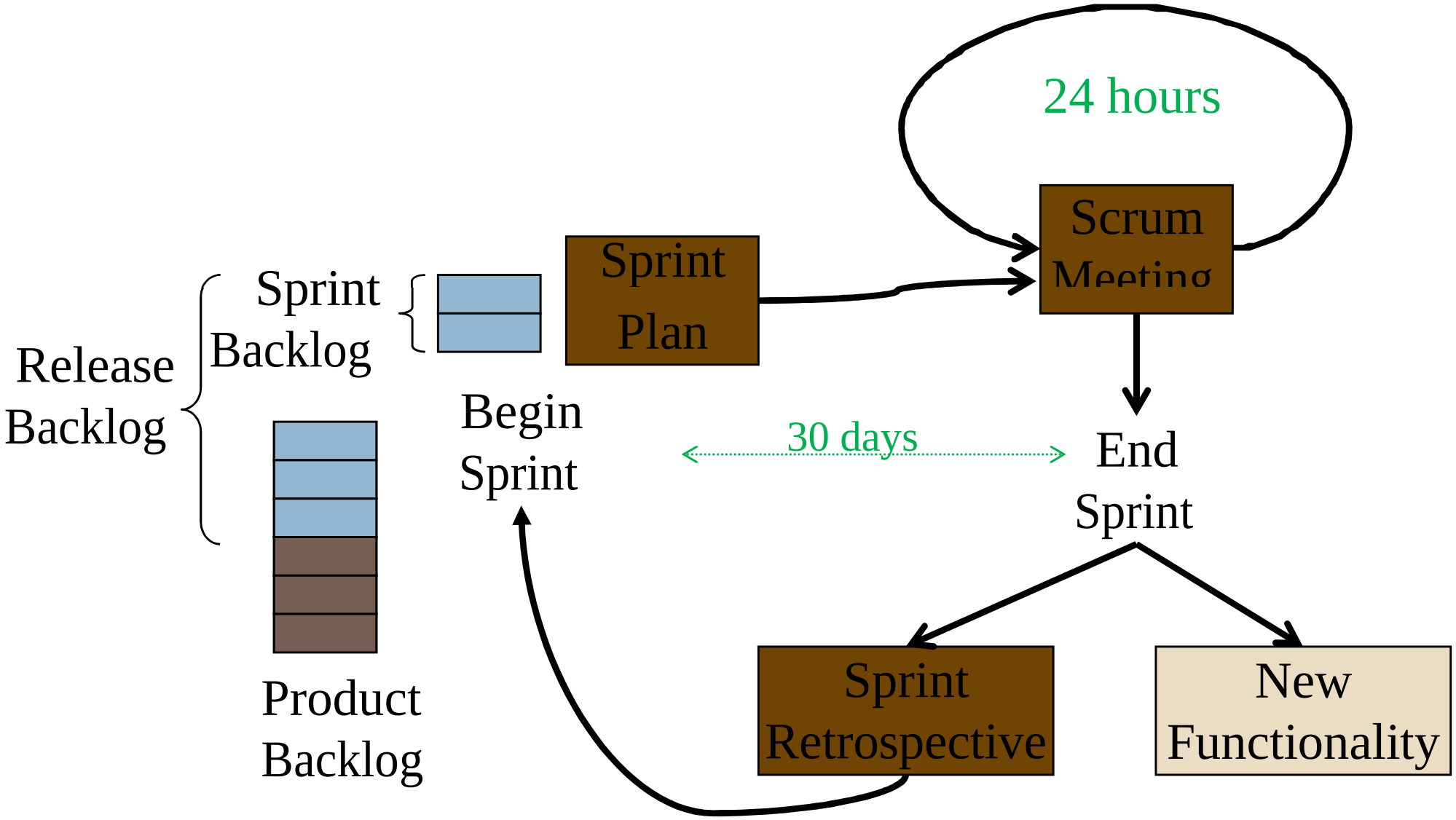
SCRUM Practices

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

SCRUM Practices

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

SCRUM Practices



SCRUM Practices

AGILE DEVELOPMENT

Drawbacks and Challenges of Agile Methods

- Undefined goals
  - Feature creep
  - Neverending project with overruns
- Need clear, single, invested customer
- All-or-nothing adoption of techniques
  - Some parts work only if lots of other aspects used
- Team size limited (smaller teams)

Planning

- Initial Exploration
- Spiking, Splitting, and Velocity
- Release Planning
- Iteration Planning
- Task Planning
- The Halfway Point
- Iterating

Planning

- Initial Exploration
  - At the start of the project, the developers and customers try to identify all the really significant user stories they can. However, they don't try to identify all user stories.
  - As the project proceeds, the customers will continue to write new user stories.
  - The flow of user stories will not shut off until the project is over.
  - The developers work together to estimate the stories. The estimates are relative, not absolute.

Planning

- Spiking, Splitting, and Velocity
- Splitting
  - For example, “Users can securely transfer money into, out of, and between their accounts.” - big story. Estimating will be hard and probably inaccurate. Further Splitting above story in
    - Users can log in.
    - Users can log out.
    - Users can deposit money into their account.
    - Users can withdraw money from their account.
    - Users can transfer money from their account to another account.

Planning

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

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

Planning

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

Planning

- Iteration Planning
  - typically two weeks long
  - the customers choose the stories that they want implemented in the first iteration.
  - The order of the stories within the iteration is a technical decision that makes the most technical sense.
  - The customers cannot change the stories in the iteration once the iteration has begun - Reordering is possible
  - The iteration ends on the specified date, *even if all the stories aren't done.*

Planning

- If the team got 31 story points done last iteration, then they should plan to get 31 story points done in the next. Their velocity is 31 points per iteration.
- If the team gains in expertise and skill, the velocity will rise commensurately. If someone is lost from the team, the velocity will fall. If an architecture evolves that facilitates development, the velocity will rise.

Planning

- Task Planning
  - The developers break the stories down into development tasks. A task is something that one developer can implement in 4-16 hours. The stories are analyzed, with the customers' help, and the tasks are enumerated as completely as possible.
  - A list of the tasks is created on a flip chart, whiteboard, or some other convenient medium. Then, one by one, the developers sign up for the tasks they want to implement.

Planning

- Developers, Database Guys, GUI guys (whole Project team) may sign up for respective kind of task.
- Task selection continues until either all tasks are assigned

Planning

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

Planning

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

Planning

- Iterating
  - Every two weeks, the current iteration ends and the next begins.
  - At the end of each iteration, the running executable is demonstrated to the customers.
  - The customers are asked to evaluate the look, feel, and performance of the project.
  - The customers can measure velocity. They can predict how fast the team is going, and they can schedule high-priority stories early.

Test Driven Development

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

Test Driven Development

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

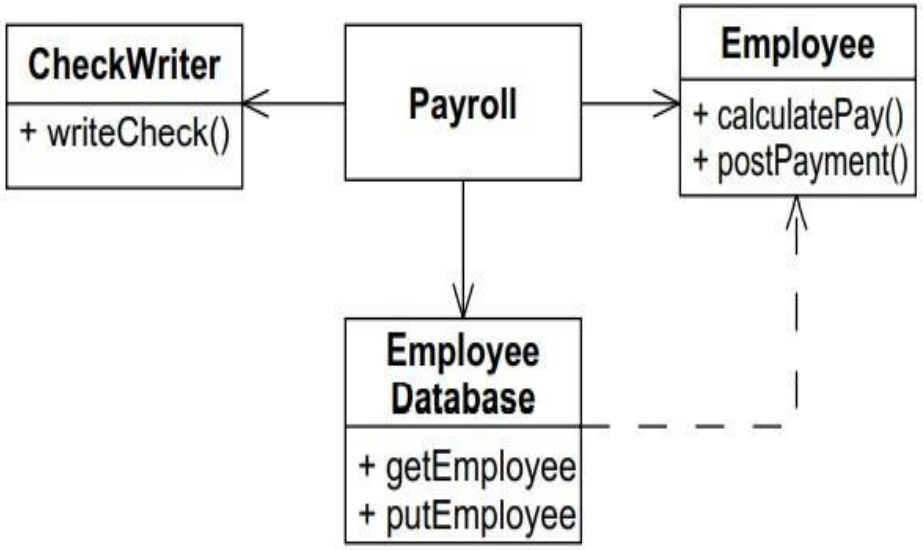
- we are immediately concerned with the interface of the program as well as its function. By writing the test first, we design the software to be ***conveniently callable***.
- we are immediately concerned with the interface of the program as well as its function. By writing the test first, we design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

Test Driven Development

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

Test Isolation

- The act of writing tests before production code often exposes areas in the software that ought to be decoupled.
- For example, Lets see a simple UML diagram of a ***payroll application***.

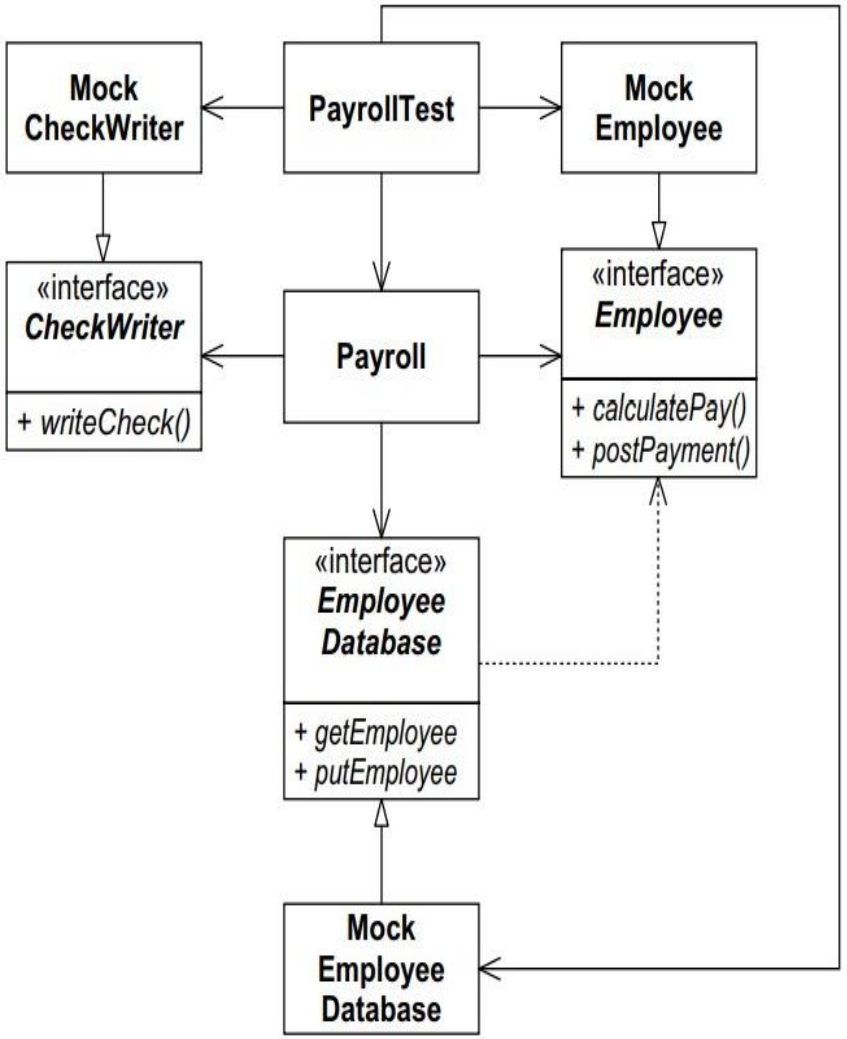


Test Isolation

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

Test Isolation

- It creates the appropriate mock objects, passes them to the **Payroll** object, tells the **Payroll** object to pay all the employees, and then asks the mock objects to verify that all the checks were written correctly and that all the payments were posted correctly.



Acceptance Test

- Unit tests are necessary but insufficient as verification tools.
- Unit tests verify that the small elements of the system work as they are expected to, but they do not verify that the system works properly as a whole.
- ***Unit tests*** are ***white-box tests*** that verify the individual mechanisms of the system.
- ***Acceptance tests*** are ***black-box tests*** that verify that the customer requirements are being met.

Acceptance Test

- Acceptance tests are written by folks who do not know the internal mechanisms of the system.
- They may be written directly by the customer or by some technical people attached to the customer, possibly QA. Acceptance tests are programs and are therefore executable.
- Acceptance tests are the ultimate documentation of a feature.
- Creating an acceptance testing framework may seem a daunting task.

Acceptance Test

- **Example of Acceptance Testing**
  - Consider, the payroll application
  - we must be able to add and delete employees to and from the database. We must also be able to create paychecks for the employees currently in the database. Fortunately, we only have to deal with salaried employees. The other kinds of employees have been held back until a later iteration.
  - it makes sense that the acceptance tests should be written in simple text files.

```
AddEmp 1429 "Robert Martin" 3215.88
Payday
Verify Paycheck EmpId 1429 GrossPay 3215.88
```

Acceptance Test

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

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

Acceptance Test

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

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

Refactoring – What is it?

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

To improve some of the nonfunctional attributes:

code readability

reduced complexity to improve maintainability better design

➔ extensibility

Refactoring – What is it?

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

— Joshua Kerievsky [Refactoring to Patterns]

Refactoring – What is it?

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

Refactoring – What is it?

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

— Joshua Kerievsky [Refactoring to Patterns]

Refactoring - When do I use?

- Bad smells!
- A structure in the code suggests, and sometimes even scream for, opportunities for refactoring
- This humorous advice rely on the experience of programmers and on the clarity of code

- Indications to possible bad smells:
- Duplicated code *share one same method*
- Long method *extract one or more smaller methods*
- Large class *divide in more cohesive classes*
- Long parameter list *encapsulate them*

Refactoring

- There is a almost a hundred catalogued refactorings
  - each one has a name, a motivation, and a mechanics
- They are usually organized into the following groups:
  - Composing Methods (e.g., Extract Method)
  - Moving Features Between Objects (e.g., Move Method)
  - Organizing Data (e.g., Encapsulate Field)
  - Simplifying Conditional Expressions (e.g., Consolidate Conditional Expression)
  - Making Method Calls Simpler (e.g., Rename Method)
  - Dealing with Generalization (e.g., Pull Up Method)
  - Big Refactoring (e.g., Extract Hierarchy)

Refactoring - Extract Method

You have a code fragment that can be grouped together

⇓

Turn the fragment into a method whose name explains the purpose of the method

Extract Method –Theoretical Example

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

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

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

Extract Method - Motivation

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

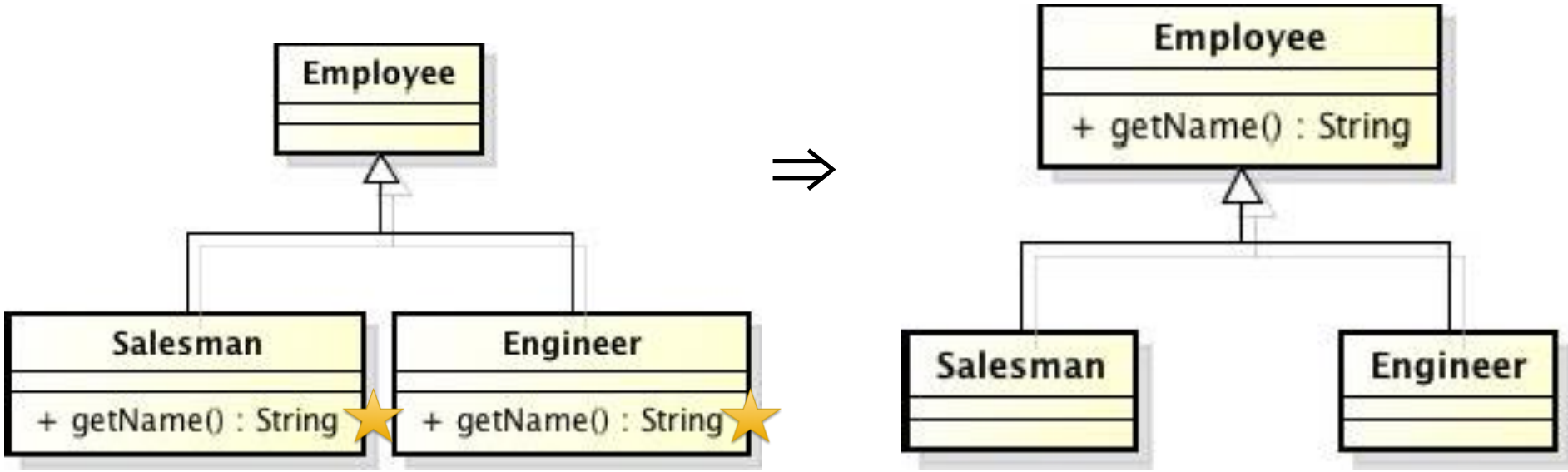
Refactoring - Pull Up Method

You have methods with identical results on subclasses

⇓

Move them to the superclass

Pull Up Method - Theoretical Example



Pull Up Method - Motivation

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

Refactoring - Move Method

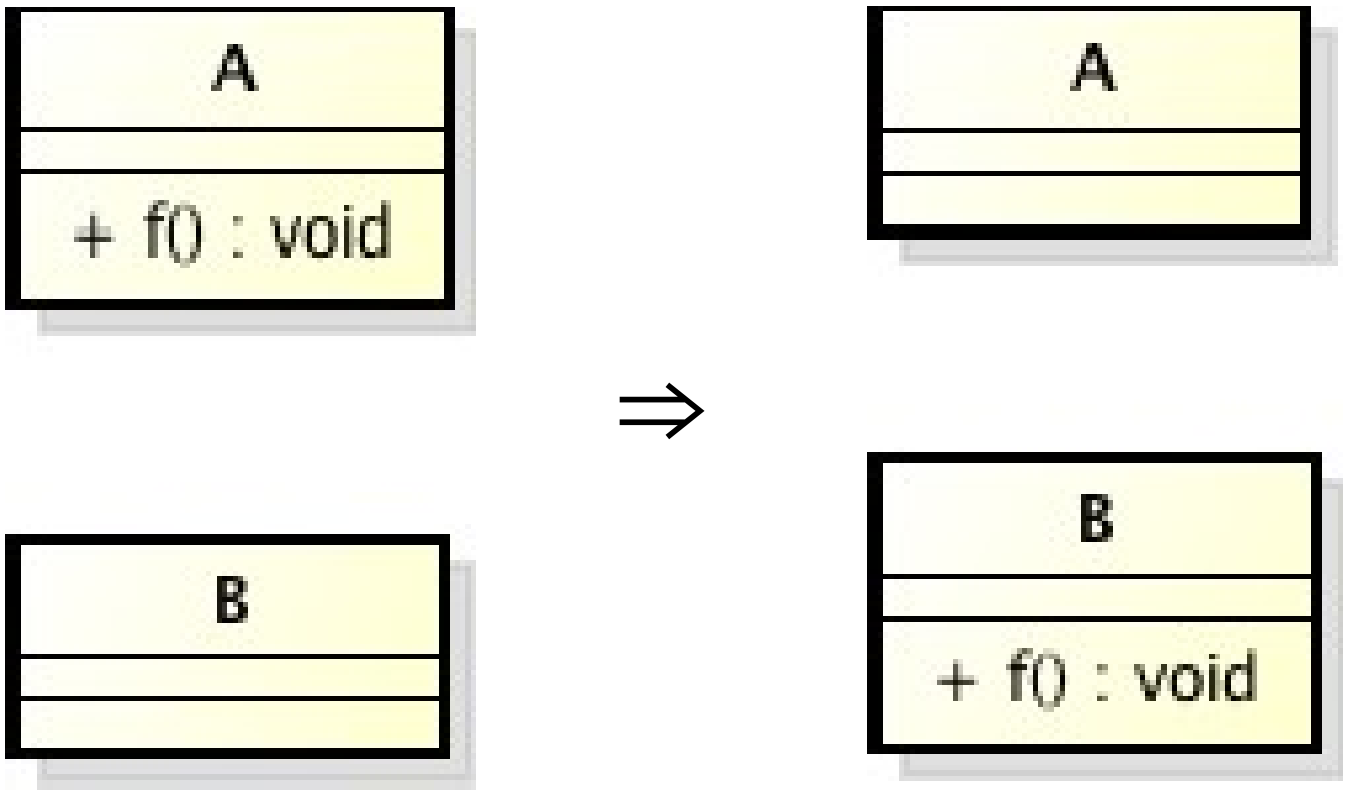
A method is, or will be, using or used by more features of another class than the class on which it is defined

(*Feature Envy*)

⇓

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

Move Method - Theoretical Example



Move Method - Motivation

- Moving methods is the bread and butter of refactoring
  - classes with too much behavior
  - classes are collaborating too much (too highly coupled)

- By moving methods around:
  - make classes simpler
  - high cohesion (classes end up being a more crisp implementation of a set of responsibilities)

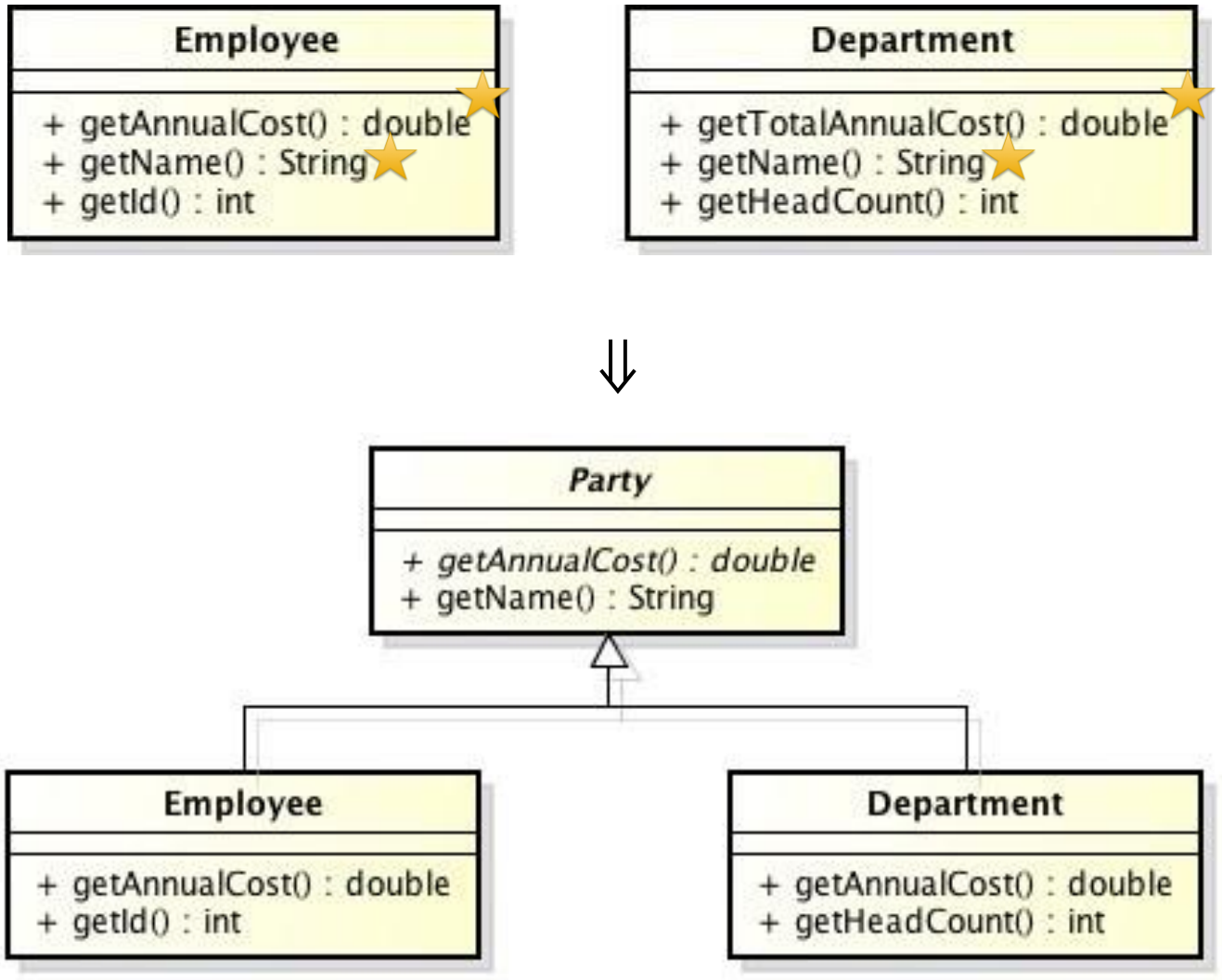
Refactoring - Extract Superclass

You have two classes with similar features

⇓

Create a superclass and move the common features to the superclass

Extract Superclass - Theoretical Example



Extract Superclass– Motivation

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


Do you have any
QUESTIONS?