20 - Agile Methods & AOSD

- General
- XP
  - Extreme Programming Explained, Kent Beck Addison Wesley 1999
  - Refactoring: Improving the Design of Existing Code, Martin Fowler, Addison Wesley 1999
  - http://www.extremeprogramming.org
- AOP
  - Aspect-Oriented Programming with AspectJ™ Erik Hilsdale, Gregor Kiczales (with Bill Griswold, Jim Hugunin, Wes Isberg, Mik Kersten),

Agile Methods

- Agile methods
  - Adaptive Software Development (ASD), Agile Modeling, Crystal Methods, Dynamic System Development Methodology (DSDM), Extreme Programming (XP), Feature Driven Development, Lean Development, and Scrum
- Plan-Driven Methods include:
  - documented process procedures that involve tasks and milestone plans
  - product development strategies that involve requirements, designs, and architectural plans.
- Risk-driven spiral methods and frameworks
  - Rational UnifiedProcess (RUP), Model-Based Architecting and Software Engineering (MBASE), and the CMMI
Agile Methods

- emphasize these values:
  - individuals and interactions over processes and tools
  - working software over comprehensive documentation
  - customer collaboration over contract negotiation
  - responding to change over following a plan

- a method is agile when software development is
  - incremental (small releases with rapid cycles)
  - cooperative (customers and developers work together and communicate closely)
  - straightforward (easy to learn and modify)
  - adaptive (changes can be made at last minute)

XP Process (Abrahamsson, et al)

- 6 Phases: Exploration, Planning, Iteration, Productionizing, Maintenance, Death
Scrum

- empirical approach, applying ideas of industrial process control theory
- does not specify development techniques, but concentrates on team members
- includes three phases: pre-game, development and post-game

Crystal

- family of methodologies, select & tailor to fit to project
- "color coded" to represent the “heaviness” of the methodology
- Crystal Clear - small projects (D6)
- Crystal Orange - medium-size projects (D40)
Is Open Source (OSS) Agile?

- OSS development differs from agile development in philosophical, economic and team structure aspects (Cockburn 2002)
- however, there are some similarity
  - early and frequent releases
  - a lack of plan-based development features such as plans, system-level designs, schedules, defined processes
- Phases (Sharma 2002): problem discovery; finding volunteers; identifying solutions; code development and testing; code change review; code commit; release management

The changing OSS environment

- large companies (e.g., IBM) showing interest in OSS and beginning to serve as mediators (= managers?)
- new service industries (e.g., Red Hat) providing support
- repositories (e.g., SourceForge) provide hosting, version control, bug tracking, project management, backups, archives
- volunteer hierarchies
  - project leaders, developers (senior/core, peripheral, occasional contributors), community input (users, posters)
- market forces
  - developers have to find interest and attract volunteers in a very harsh and competitive OSS environment
Is RUP Agile?

- Contains extensive guidelines for process phases that may be irrelevant in an agile approach
- There are those that argue RUP can be made to be agile.
- As to the Agile principles, here are a few:
  - Active Stakeholder Participation
    - RUP includes project stakeholders, such as users and customers, but to be agile project stakeholders need to take on modeling roles
  - Collective Ownership
    - RUP supports collective ownership with its strong focus on configuration management issues, although its change management processes may potentially get in your way
  - Prove it With Code
    - At the end of every iteration, except perhaps for the ones during the Inception phase, RUP specifically states that you should have a working prototype.

Rational Method Composer

- *out-of-the-box processes or delivery processes*
  - RUP for small projects, medium-sized projects, and large projects (Classic RUP)
  - RUP for COTS (Commercial Off-The-Shelf) or packaged application development
  - RUP for systems engineering
  - RUP for service-oriented architecture (SOA)
  - RUP for maintenance
  - RUP for J2EE
  - RUP for XP

*Will RMC make RUP agile?*
Agile vs. Plan-Based Methods

- plan-based methods not used in practice?
  - too mechanistic
  - too hard to learn
  - potential adopters are skeptical
- agile methods
  - began with agile manifesto
  - lack of empirical evidence
  - not applicable to large and/or critical projects?

Microsoft’s S&S Process

- S&S = “Synchronize and Stabilize”
- Strategy
  - Parallel development -- team structure parallels code structure
  - Continually test
  - Metric data for milestone completion
  - Common developer language
  - Divide large projects into multiple milestone cycles
  - Use a vision statement and outline specification of features to guide projects

Sawyer & Guinan 1998
**Microsoft S&S Process**

- **Vision Statement**
- **Specification Document**
- **Schedule and Feature Team Formation**
  - 1 prog. mgr, 3–8 developers, and 3–8 testers (who work in parallel 1:1 with developers).
- **Feature development in 3 or 4 sequential subprojects**
  - Subproject I First 1/3 of features (Most critical features and shared components)
  - Subproject II Second 1/3 of features
  - Subproject III Final 1/3 of features (Least critical)
- **Internal Testing**
- **External Testing (OEMs, ISVs, & end users)**
- **Release preparation**

*Cusumano & Selby 1997*

---

**S+S Issues**

- **Benefits**
  - Breaks down large products into manageable pieces
  - Large teams to work like small teams, proceeding in parallel but synchronizing continuously
  - Facilitates competition on customer input, product features, and short development times
  - Allows a product team to be very responsive to events in the marketplace by “always” having a product ready to ship

- **Problems**
  - Doesn’t work for teams that are not co-located
  - Only works for feature-driven development
  - No built-in maintenance or support
  - Planned as future versions of product
  - Must have customer base to drive development decisions

*Cusumano & Selby 1997*
Process Model Comparison

### User involvement
- Over the fence: XP, SA, SP, C

### Lifetime involvement
- Code & release: XP, SA, SP, C
- Throughout ownership

### Locality of developers
- Same location: XP, SA, SP, C
- Distributed

### Talent assumption
- Mostly tacit: XP, SA, SP, C
- Requires mostly knowledge

### Management involvement
- Management at arm's length: XP, SA, SP, C

### Sequentiality of development
- Design first: XP, SA, SP, C
- Proteus evolutionarily

### IP control
- Proprietary: XP, SA, SP, C
- Best publish code

Mary Shaw 2001

---

Agile and Other Methods

- **Agile**
  - Adaptive Software Development (ASD), Agile Modeling, Crystal Methods, Dynamic System Development Methodology (DSDM), Feature Driven Development, Lean Development

- **Other “hot” technology**

- **Other Approaches**
  - Design by Contract
  - Aspect-Oriented Software Development
Design by Contract

- Originated by Bertram Meyer (Eiffel)
  - Incorporated in others methods (e.g., JML) and tools (e.g. Parasoft)
  - Methodology for evolving code together with its specification.
  - Classes define their responsibility precisely.
  - Class invariants, method preconditions and postconditions.
  - Compiler instruments code to monitor.

Preconditions and postconditions

- Class to client: “If you promise to call r with pre satisfied, then I promise to deliver a final state in which post is satisfied…”
- A method’s precondition
  - Says what must be true to call it, i.e., what must hold upon entry to method
  - Binds the client.
- A method’s normal postcondition
  - Method guarantees it will hold upon exit
  - What is true when it returns normally (i.e., without throwing an exception).
- A method’s exceptional postcondition
  - What is true when a method throws an exception.
  - //@ signals (IllegalArgumentException e) x < 0;
- Class Invariants
  - Global properties of a class.
  - Must be preserved by all exported routines.
The role of a contract
- may monitor at run time - debugging tool.
  - eiffel - by the compiler.
  - other languages - specific tools.
- conceptual tool for correctness and robustness.
- design aid.
- aid to understanding
- documentation

advantages
- clear responsibility for checking.
- run time violation shows a bug:
  - Precondition violation -- bug in client.
  - Postcondition violation -- bug in supplier.
- simplify code
  - method need not check precondition.
  - If precondition is not satisfied, do anything

Contracts and inheritance
Methodological implications of contracts on inheritance:
- Invariants and postconditions may only be strengthened;
- Preconditions may only be weakened.
- Eiffel enforces this principle.
Aspect-oriented programming

- AOP compliments O-O programming, but doesn’t replace it
- the problem
  - some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code
- examples:
  - logging (tracking program behavior to a file)
  - profiling (determining where a program spends its time)
  - tracing (determining what methods are called when)
  - session tracking, session expiration
  - special security management
- the result is crosscutting code—the necessary code “cuts across” many different classes and methods

The following AOP/AOSD slides adapted from Aspect-Oriented Programming with AspectJ™ Erik Hilsdale, Gregor Kiczales (with Bill Griswold, Jim Hugunin, Wes Isberg, Mik Kersten).

Some examples

```java
public class SomeBusinessClass extends OtherBusinessClass {
    // Core data members

    // Override methods in the base class
    public void performSomeOperation(OperationInformation info) {
        .
        // ==== Perform the core operation ====
    }
}
```

Breaking modularity

- Got the picture?
- Non-modularization due to client-server nature of OOP
- Current popular solution: EJB, servlets, dynamic proxies

modularity

intuitive definition

a concern is implemented\(^1\) in a modular way if the code for the concern is:
- localized and
- has a clear interface with the rest of the system

\(^1\) coded, designed, modeled …

- xml parsing
- url pattern matching
- logging

\(\text{code from org.apache.tomcat}\)
session expiration is not modularized...

problems like...

session tracking is not modularized

HTTPRequest

getRequestURI()

getSession()

getRequestSessionId()

...

SessionInterceptor

requestMap(request)

beforeBody(req, resp)

...

Session

getAttribute(name)

setAttribute(name, val)

invalidate()

...

HTTPResponse

getSession()

setContentType(contentType)

getOutputStream()

setSessionId(id)

...

Servlet
### Crosscutting Concerns

#### Diagram

- **HTTPRequest**
  - `getCookies()`
  - `getRequestURI()`
  - `getSession()`
  - `getRequestedSessionId()`
  - ...

- **SessionInterceptor**
  - `requestMap(request)`
  - `beforeBody(req, resp)`
  - ...

- **HttpResponse**
  - `getRequest()`
  - `setContentType(contentType)`
  - `setOutputStream()`
  - `setSessionId(id)`
  - ...

- **Session**
  - `getAttribute(name)`
  - `setAttribute(name, val)`
  - `invalidate()`
  - ...

- **Servlet**

### Non-modularization

- **Symptoms**
  - Code tangling
  - Code scattering
    - Duplicated code blocks
    - Complementary code blocks
  - Duplicated code blocks
  - Complementary code blocks

- **Consequences**
  - Redundant code
    - Same fragment of code in many places
  - Difficult to reason about
    - Non-explicit structure
    - The big picture of the tangling isn’t clear
  - Difficult to change
    - Have to find all the code involved...
    - …and be sure to change it consistently
    - …and be sure not to break it by accident
  - Inefficient when crosscutting code is not needed
the AOP idea

- crosscutting is inherent in complex systems
- crosscutting concerns
  - have a clear purpose
  - have a natural structure
    - defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow...
- so, let’s capture the structure of crosscutting concerns explicitly...
  - in a modular way
  - with linguistic and tool support
- aspects are
  - well-modularized crosscutting concerns

If we just could...
How does AOP do it?

AOP development stages

AspectJ™

- AspectJ is a small, well-integrated extension to Java
  - Based on the 1997 PhD thesis by Christina Lopes, *A Language Framework for Distributed Programming*
- AspectJ modularizes crosscutting concerns
  - That is, code for one *aspect* of the program (such as tracing) is collected together in one place
- The AspectJ compiler is free and open source
- AspectJ works with JBuilder, Forté, Eclipse, probably others
What is a Concern?

- a concern is a particular goal, concept, or area of interest
- concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts

Terminology

- A join point is a well-defined point in the program flow
- A pointcut is a group of join points
- Advice is code that is executed at a pointcut
- Introduction modifies the members of a class and the relationships between classes
- An aspect is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable
a simple figure editor

Display

Figure

makePoint(..)
makeLine(..)

FigureElement

moveBy(int, int)

Point

2

getX()
getY()
setX(int)
setY(int)
moveBy(int, int)

Line

getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)

factory methods

operations that move elements

UNIVERSITY OF MASSACHUSETTS AMHERST - DEPARTMENT OF COMPUTER SCIENCE - CPSCi520/620 Fall 2005

join points

imagine l.moveBy(2, 2)

key points in dynamic call graph

a Line

dispatch

a method execution returning or throwing

a method call returning or throwing

a Point

dispatch

a method execution returning or throwing

UNIVERSITY OF MASSACHUSETTS AMHERST - DEPARTMENT OF COMPUTER SCIENCE - CPSCi520/620 Fall 2005

© Rick Adrion 2005 (except as noted)
Join points

- A join point is a well-defined point in the program flow
  - We want to execute some code ("advice") each time a join point is reached
  - We do not want to clutter up the code with explicit indicators saying "This is a join point"
  - AspectJ provides a syntax for indicating these join points "from outside" the actual code
- A join point is a point in the program flow "where something happens"
  - Examples:
    - When a method is called
    - When an exception is thrown
    - When a variable is accessed

Join point terminology

- several kinds of join points
  - method & constructor call
  - method & constructor execution
  - field get & set
  - exception handler execution
  - static & dynamic initialization
join point terminology

- All join points on this slide are within the control flow of this join point

Pointcuts

- Pointcut definitions consist of a left-hand side and a right-hand side, separated by a colon
- The left-hand side consists of the pointcut name and the pointcut parameters (i.e. the data available when the events happen)
- The right-hand side consists of the pointcut itself
- Example pointcut:
  ```
  pointcut setter(): call(void setX(int));
  ```
Example pointcut designators

- When a particular method body executes:
  - execution(void Point.setX(int))
- When a method is called:
  - call(void Point.setX(int))
- When an exception handler executes:
  - handler(ArrayOutOfBoundsException)
- When the object currently executing (i.e. this) is of type SomeType:
  - this(SomeType)
- When the target object is of type SomeType
  - target(SomeType)
- When the executing code belongs to class MyClass
  - within(MyClass)
- When the join point is in the control flow of a call to a Test's no-argument main method
  - cflow(call(void Test.main()))

Pointcut designator wildcards

- It is possible to use wildcards to declare pointcuts:
  - execution(* *(..))
    - Chooses the execution of any method regardless of return or parameter types
  - call(* set(..))
    - Chooses the call to any method named set regardless of return or parameter type
    - In case of overloading there may be more than one such set method; this pointcut picks out calls to all of them
pointcut composition

- A pointcut is a kind of predicate on join points that:
  - Can match or not match any given join point and
  - Optionally, can pull out some of the values at that join point
- Pointcuts compose like predicates, using &&, || and !
  - A "void Line.setP1(Point)" call
  - Call (void Line.setP1(Point)) || or call (void Line.setP2(Point));
  - A "void Line.setP2(Point)" call

Whenever a Line receives a "void setP1(Point)" or "void setP2(Point)" method call.

Crosscutting Concern - Example

Notify ScreenManager if a figure element moves
Figure Editor

```java
class Line {
    private Point p1, p2;
    Point getPl() { return p1; }
    Point getP2() { return p2; }
    void setPl(Point p1) {
        this.p1 = p1;
        Display.update();
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update();
    }
}
class Point {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) {
        this.x = x;
        Display.update();
    }
    void setY(int y) {
        this.y = y;
        Display.update();
    }
}
```

- no locus of “display updating”
- evolution is cumbersome
- changes in all classes
- have to track & change all callers

Example: Display Tracking

```java
class DisplayTracker {
    static void updatePoint(Point p) {
        this.display(p);
    }...
    static void updateLine(Line l) {
        this.display(l);
    }...
}
class Point {
    void setX(int x) {
        DisplayTracker.updatePoint(this);
        this.x = x;
    }
}
class Line {
    void setPl(Point p1) {
        DisplayTracker.updateLine(this);
        this.pl = p1;
    }
}
```
after advice

action to take after computation under join points

define pointcut and aspects

aspect DisplayUpdating {
  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)));

  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}

Figure Editor with AspectJ

class Line {
  private Point p1, p2;
  Point getP1() { return p1; } Point getP2() { return p2; }
  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
  private int x = 0, y = 0;
  int getX() { return x; } int getY() { return y; }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}

aspect DisplayUpdating {
  pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}

- clear display updating module
  - all changes in single aspect
  - evolution is modular

© Rick Adrion 2005 (except as noted)
AspectJ advice

- **Before**
  - advice runs as a join point is reached, before the program proceeds with the join point

- **After**
  - advice on a particular join point runs after the program proceeds with that join point
  - returning
    - advice is executed after a method returns normally
  - throwing
    - advice is executed after a method returns by throwing an exception
  - any
    - advice is executed after a method returns, regardless of whether it returns normally or by throwing an exception

- **Around**
  - advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point

Concluding remarks

- Aspect-oriented programming (AOP) is a new paradigm—a new way to think about programming
- AOP is somewhat similar to event handling, where the “events” are defined outside the code itself
- AspectJ is not itself a complete programming language, but an adjunct to Java
- AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code
- AspectJ is free, open source software
- Like all new technologies, AOP may—or may not—catch on in a big way
Myths and realities of AOP

- The program flow in an AOP-based system is hard to follow: True
- AOP doesn’t solve any new problems: True
- AOP promotes sloppy design: False
- AOP is nice, but a nice abstract OOP interface is all you need: False
- AOP compiler simply patches the core implementation: True, but..
- AOP breaks the encapsulation: False

some AOP languages

<table>
<thead>
<tr>
<th>Join Point Model</th>
<th>join points</th>
<th>means of specifying semantics at join points</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ</td>
<td>points in execution call, get, set...</td>
<td>signatures w/ patterns, static &amp; dynamic props of JPs</td>
</tr>
<tr>
<td>static JPM</td>
<td>class members</td>
<td>signatures</td>
</tr>
<tr>
<td>Composition Filters</td>
<td>message sends &amp; receptions</td>
<td>signature &amp; property based object queries</td>
</tr>
<tr>
<td>Hyper/J</td>
<td>members</td>
<td>signatures w/ patterns, whole class ops</td>
</tr>
<tr>
<td>Demeter traversals</td>
<td>when traversal reaches object or edge</td>
<td>class &amp; edge names</td>
</tr>
</tbody>
</table>

Hyper/J, Demeter traversals: 
- Some AOP languages include: AspectJ, Composition Filters, Hyper/J, Demeter traversals.

- Hyper/J:
  - Supports both declarative (filters) and imperative (~ advice) approaches.
  - Enables the addition, composition (and removal) of methods.

- Demeter traversals:
  - Provides means to define visit methods during traversal.

Note: The table above summarizes various AOP languages and their means of specifying semantics at join points.
Software Processes

- Software processes are:
  - the set of activities, methods, and practices that are used in the production and evolution of software
  - devices for creating, modifying, analyzing, understanding software artifacts and products

-Hypothesis: Processes are software

- Improve quality by improving processes
  - Build in quality in, don’t “test in” quality (manufacturing)
  - Use processes to manage complex activities
  - Many observed “process errors”

- Proposed approach
  - Use computers to help perform processes
  - Analyze processes to determine and eliminate defects
  - Use demonstrably superior processes to identify risks, mitigate their consequences, demonstrate quality

Software Process as Software

- Software processes should be developed using a (Software development process) development process

- Process Requirements
  - Key to designing suitable process
  - Basis for evaluation and improvement of process

- Process Specification/Modeling/Design
  - Helps conceptualization, communication, visualization
  - Can be management aid

- Process Code
  - Provides rigor and complete details
  - Basis for execution/tool support and integration
**Software Process Code**

- Provides details and elaborations upon process design
- Tries to include details omitted from model/design
- Supports more detailed, precise, definitive reasoning
- Vehicle for meshing process control with product data at arbitrarily low levels of detail
- Provides superior visibility enabling better control
- Basis for better predictability
- Basis for process enactment/execution
- Blueprint for tool integration

**Software Process as Software**

- Software processes should be developed using a (Software development process) development process
- Process Measurements and Evaluation
- Results of Static Analysis and Dynamic Measurement ⇒ Basis for Process Maintenance (i.e., Process Improvement)
Software Process Formalisms

- Techniques
  - Languages
    - procedural
    - rule-based
    - object-oriented

- Modeling formalisms
  - Data flow diagrams
  - Petri Nets

- Flow graphs

- Key considerations
  - concurrency
  - exception handling
  - resource specification
  - self-modification/long lifetime
  - constraint management
  - artifact specification/management
  - real-time
  - visualizability

Language-based Formalisms

- More traditional coding languages:
  - Procedural (Sutton's Appl/A)
  - Rule-based (Kaiser's Marvel)
  - Functional Hierarchy (Katayama's HFSP)
  - Law based (Minsky)
  - Object Oriented (schema definition languages)

- Key issue: developing abstractions to facilitate process definition
HFSP design model

(a) JSD(Real_World | Design_Spec) =>
  (1) Develop_Spec(Real_World_Desc | System_Spec_Diagram)
  (2) Develop_Impl(System_Spec_Diagram | System_Impl_Diagram)
  (3) Where Real_World_Desc = Interview(Users, Developers, Real_World)
  (4) Design_Spec = union(System_Spec_Diagram, System_Impl_Diagram)

(b) Develop_Spec(Real_World_Desc | System_Spec_Diagram) =>
  (1) Develop_System_Model(Real_World_Desc | Real_World_Model, Init_System_Spec_Diagram)
  (2) Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram)

(c) Develop_System_Model(Real_World_Desc | Real_World_Model, Init_System_Spec_Diagram) =>
  (1) Model_Reality(Real_World_Desc | Real_World_Model)
  (2) Model_System(Real_World_Model | Init_System_Spec_Diagram)

(d) Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram) =>
  (1) Define_Func(Real_World_Model, Init_System_Spec_Diagram | System_Function, Function_Process)
  (2) Define_Timing(Init_System_Spec_Diagram, System_Function | Timing)
Software Process Formalisms

- Techniques
  - Languages
    - procedural
    - rule-based
    - object-oriented
  - Modeling formalisms
    - Data flow diagrams
    - Petri Nets
    - Flow graphs

IDEF 0

- Integration definition for function modeling
- Adapted from SADT
- Very widespread usage in industry
- Overburdened DFD
- Overly constrained visualization
IDEF 0 Course Scheduling

Other “DFD”s

- Many different adaptations of the basic idea
- Add control flow in
- Add various annotations on
- Add timing information
- Etc.
**Petri Net-like representations**

- Particularly effective for showing concurrency in processes
- Weak in dealing with artifacts
- Weak in dealing with exception flow

**Petri net-like formalisms**

![Diagram showing the requirements specification process]

**Requirements specification process**
**Decomposition**

- RW_Desc ➔ Develop Spec ➔ Sys_Spec_Diag

- Model_System ➔ RW_Model

**Design Process Petri net**

- Sys_Spec_Diag ➔ Develop Implementation ➔ Sys_Impl_Diag + Sys_Spec_Diag

- BOOD ➔ Req_Spec ➔ Identify Object ➔ Objects ➔ States ➔ Identify Operations ➔ Operations ➔ Objects ➔ States

- Establish Visibility ➔ Operations ➔ Objects ➔ States ➔ Visibility ➔ Establish_Interface ➔ Interface ➔ Create_Implementation ➔ Implementation ➔ Create Design Spec ➔ Design Spec ➔ Design_Spec ➔ Req_Spec