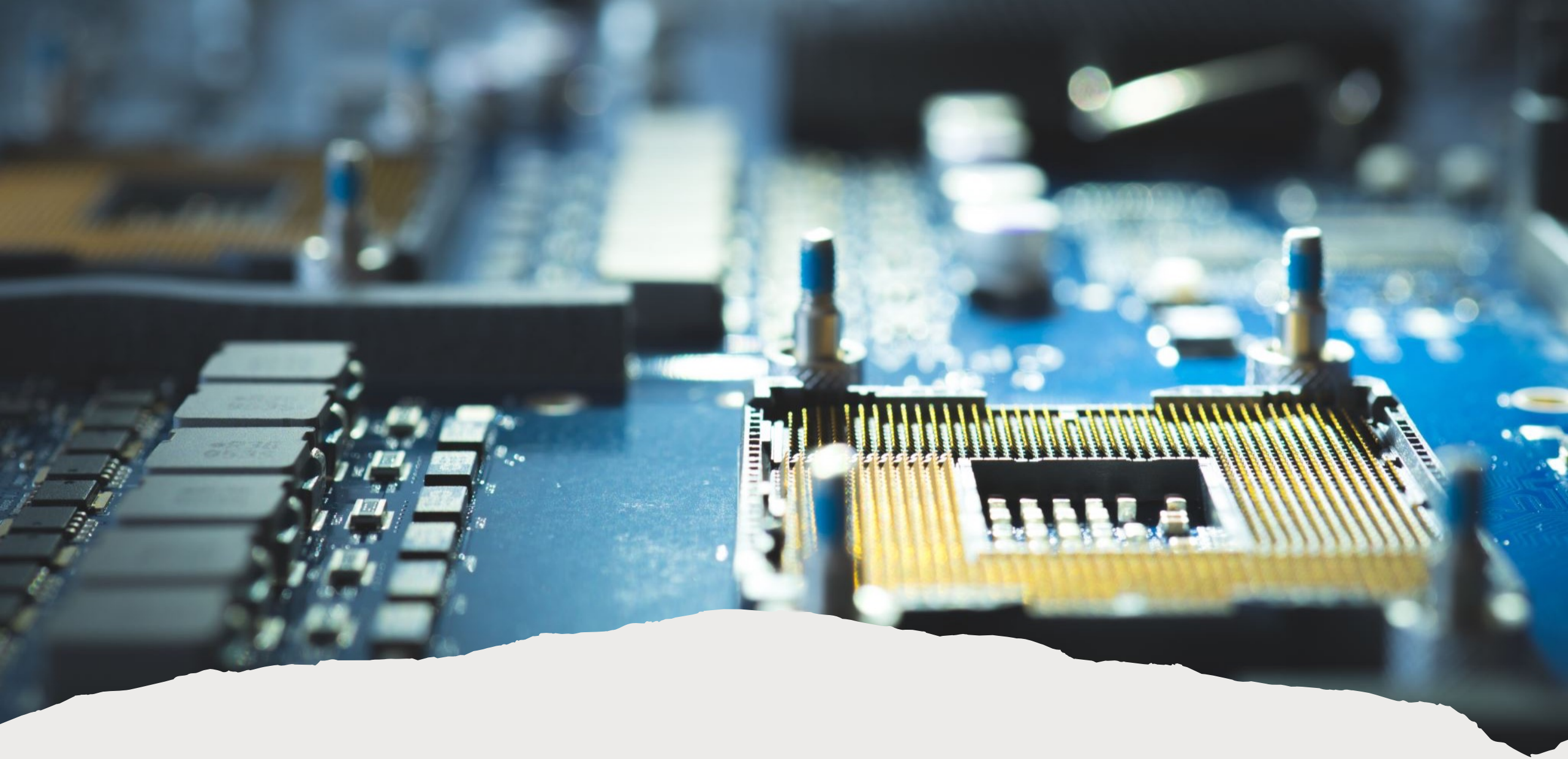
# PRODUCT ENGINEERING MINDSET - WORKSHOP

Balaji Thiruvengadam

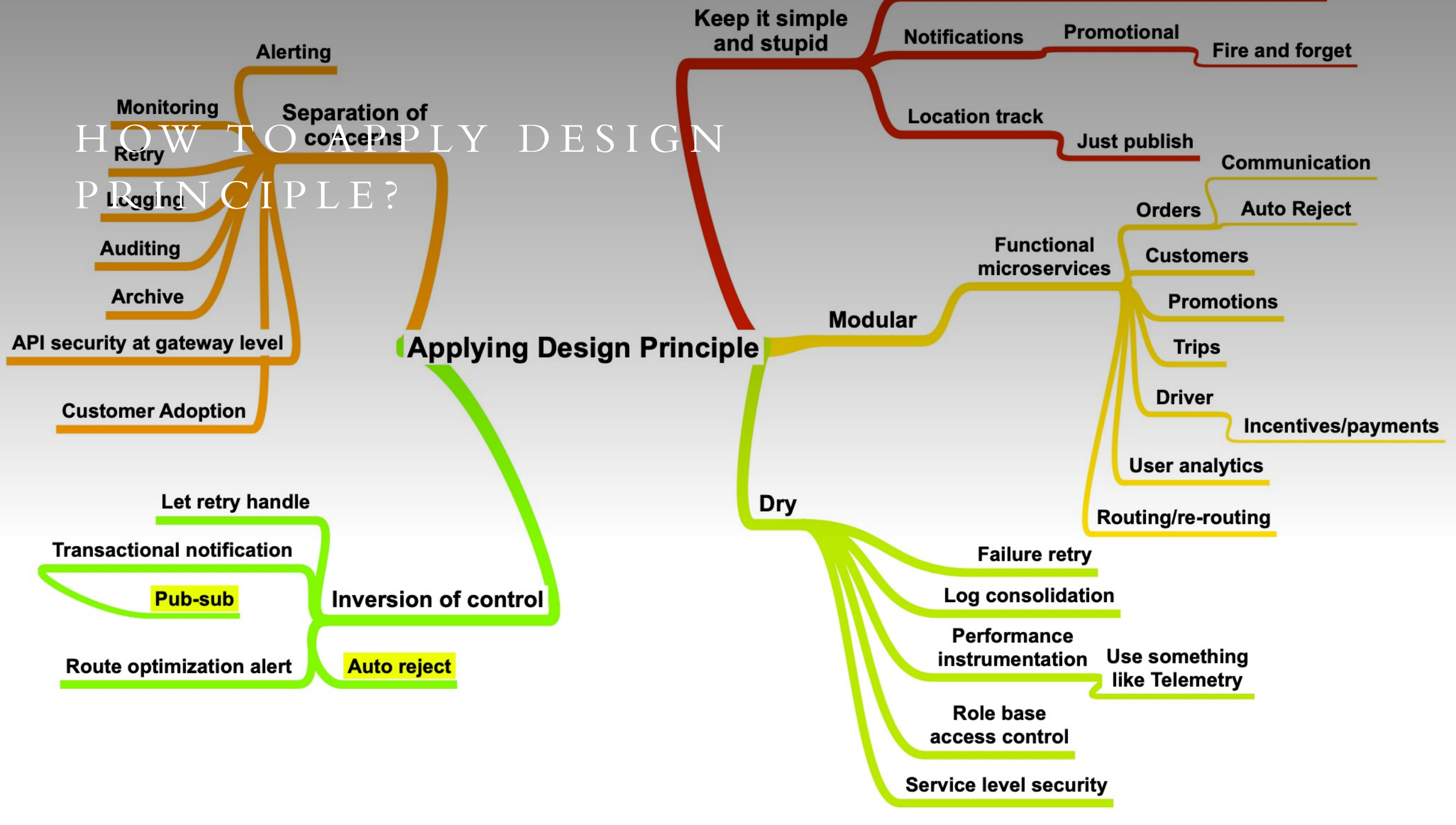
# DAY 6&7 - AGENDA

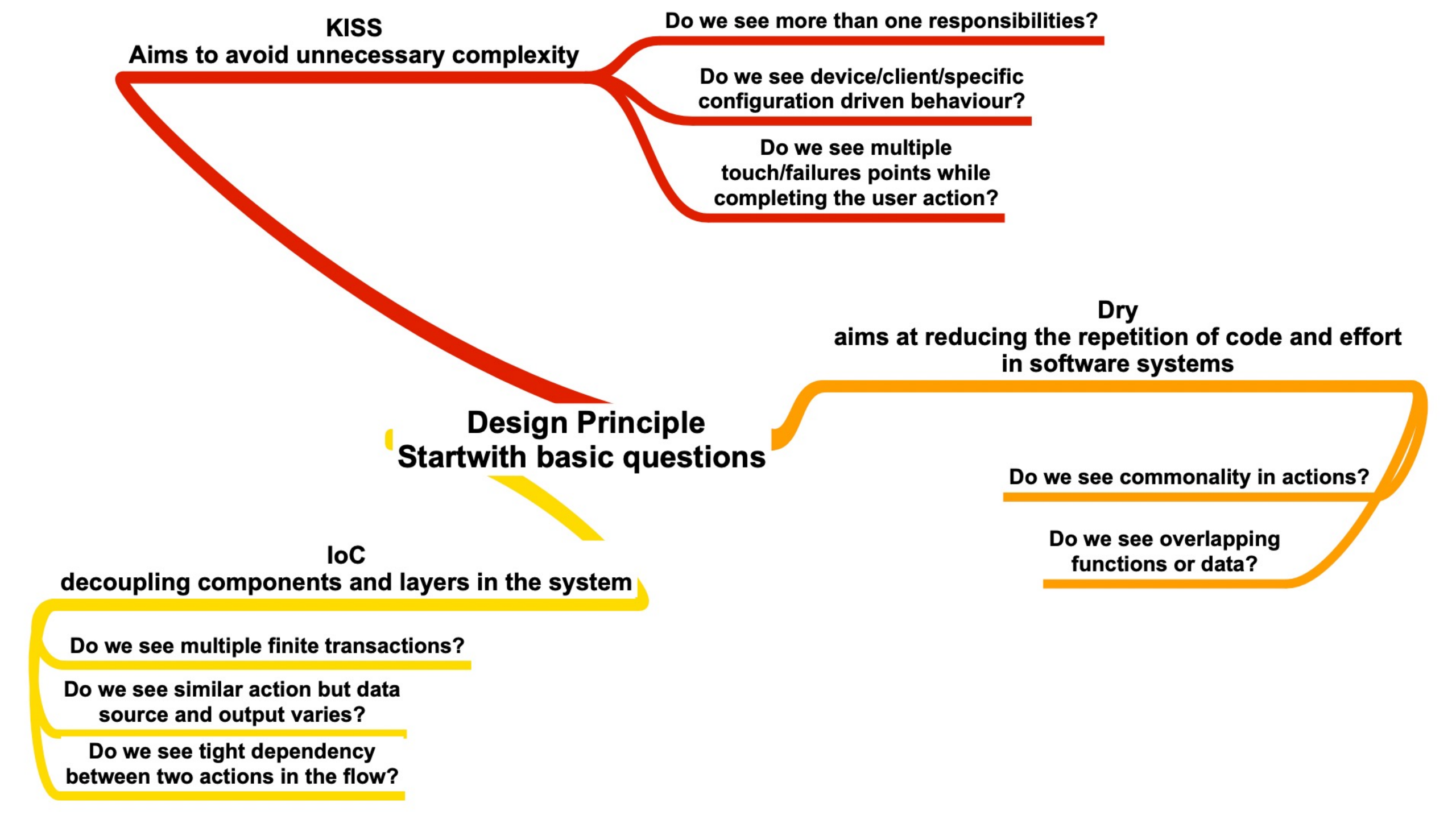
- Recap of day 5
- How to apply design principles into problems?
  Break-out on applying design principles
- Pair thinking
- Feature estimations
- Why defensive design & coding?
- Lets understand exception
- Product Engineering into Agile



# PRODUCT ENGINEERING – RECAP DAY5

# HOW TO APPLY DESIGN PRINCIPLE?




KISS
Aims to avoid unnecessary complexity
Do we see more than one responsibilities?
Do we see device/client/specific configuration driven behaviour?
Do we see multiple touch/failures points while completing the user action?
Dry
aims at reducing the repetition of code and effort in software systems
Design Principle
Startwith basic questions
Do we see commonality in actions?
Do we see overlapping functions or data?
IoC
decoupling components and layers in the system
Do we see multiple finite transactions?
Do we see similar action but data source and output varies?
Do we see tight dependency between two actions in the flow?

# WHY USER FLOW IS IMPORTANT BEFORE WE BUILD THE UX

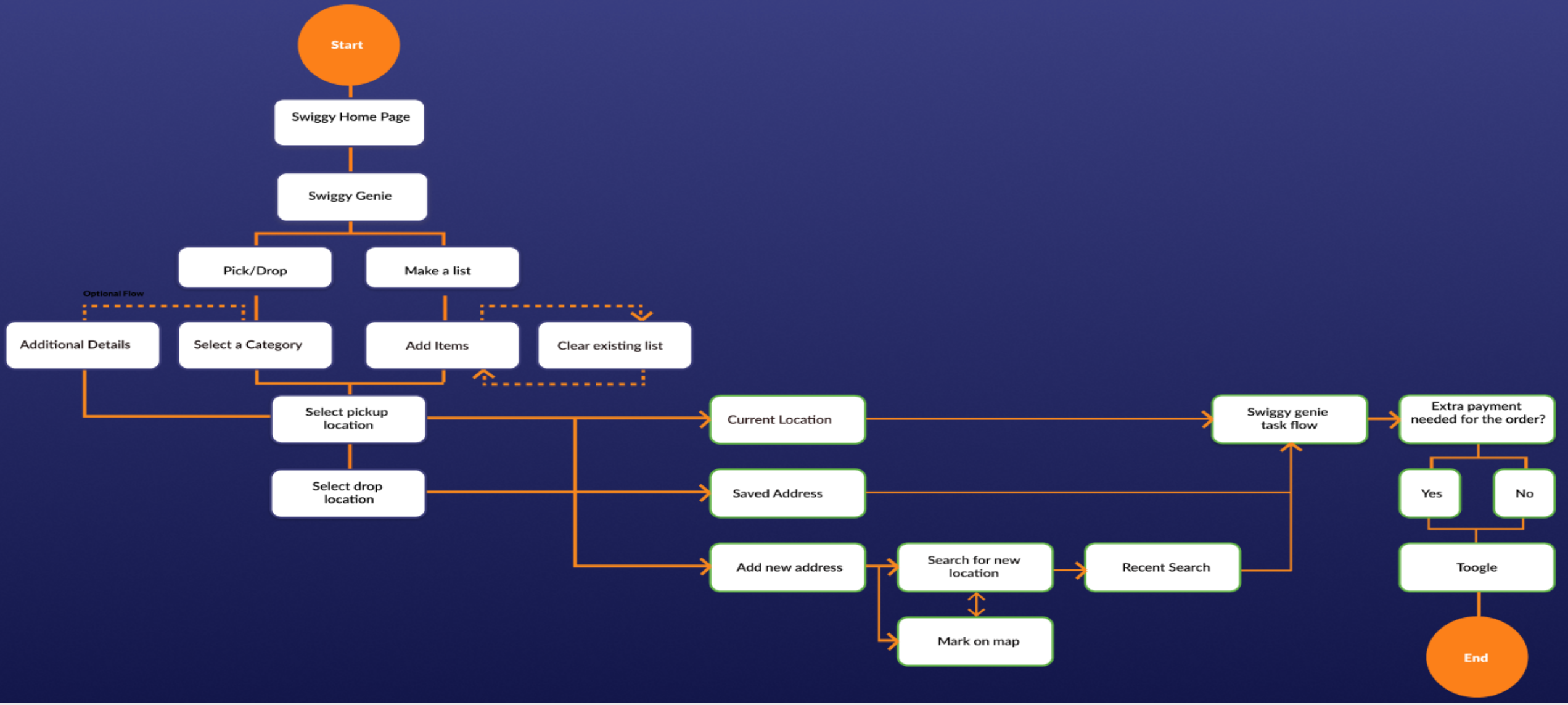
# SWIGGY GENIE

The idea of "Swiggy Genie" is to deliver anything at the doorstep of the user in the hyperlocal delivery space. They called it "Genie" because of the stories we heard from our childhood Genie is a wish-granting saviour.

The "Swiggy Genie" is classified, into two major categories:

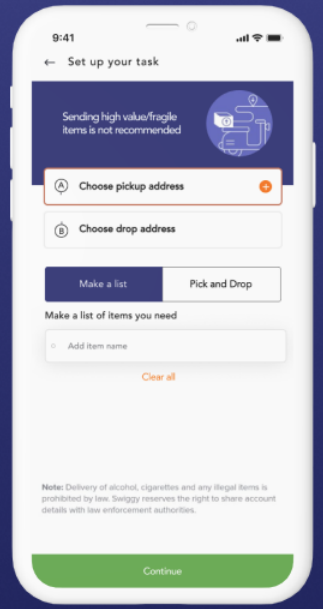
- ***Pickup & Drop*** — Users can get anything transferred from point A to point B, without leaving their home.
- ***Buy from any store*** — Users can make a custom list of items that are delivered & purchased by a Swiggy delivery partner for them.

# User Flow




SWIGGY GENIE

# Make a list task flow

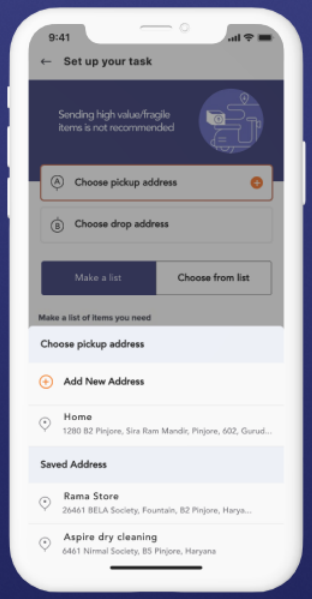



1

1. This is the home screen or the default screen of swiggy genie where the user wants to buy some groceries from the store.

2. The user will select the pick up address and the drop address from this home page.

3. The user will make a list of items to purchase from the store.

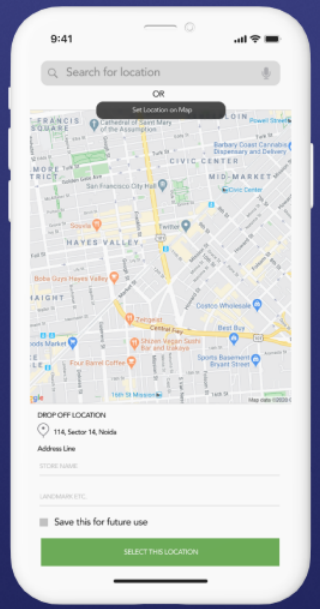



2

1. After clicking on the button of pickup/drop location this pop up appears.

2. The user can select the pickup/drop address from the saved address or from the home section.

3. When the user clicks on add new address this page is redirected to the map location.

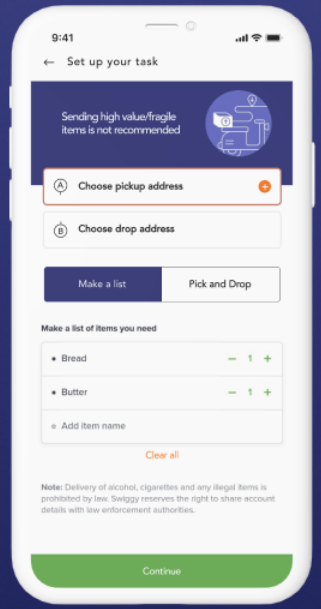



3

1. The user can select the address from the search bar.

2. The user can locate on the map directly by dragging it to the desired location.

3. The user can add the address manually by typing and can save it for further use.

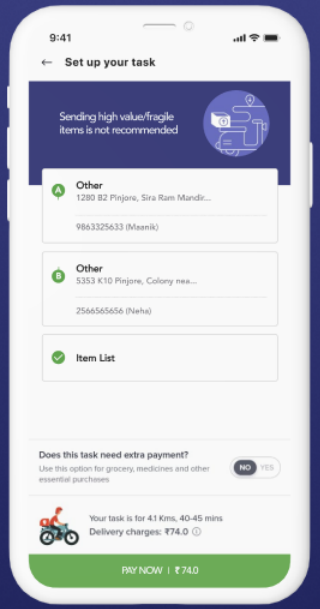



4

1. The user can add list items to be purchased and can aslo check their quantity.

2. The user can add maximum upto 10 items to purchase at once.

3. On clicking the clear all button the user can able to clear all the input items at once.



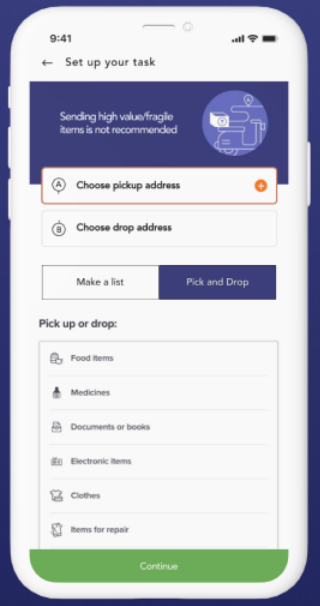

5

1. In this summary screen the user get the summary of pickup address and drop address.

2. Get the details of items which are added.

3. There is a toogle button where the user can add whether the task needs a extra payment.

# Pick/drop task flow

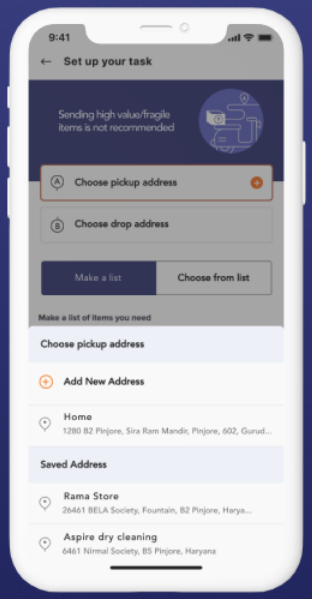



1

1. This is the home screen of pick and drop where the user can select from the list of categories mentioned below.

2. The user will select the pick up address and the drop address from this home page.

3. The user will make a list of items to purchase from the store.

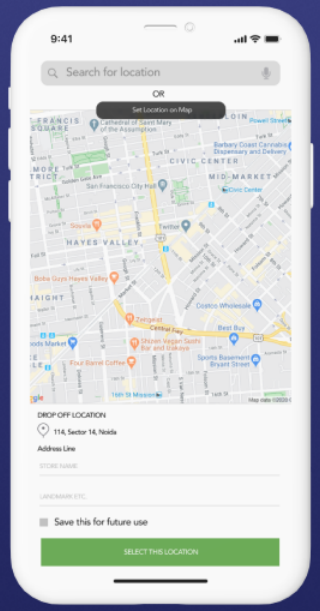



2

1. After clicking on the button of pickup/drop location this pop up appears.

2. The user can select the pickup/drop address from the saved address or from the home section.

3. When the user clicks on add new address this page is redirected to the map location.

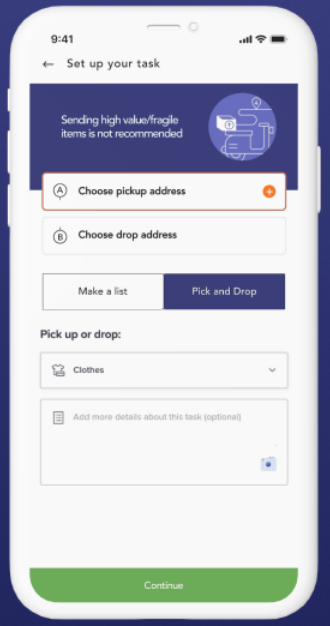



3

1. The user can select the address from the search bar.

2. The user can locate on the map directly by dragging it to the desired location.

3. The user can add the address manually by typing and can save it for further use.

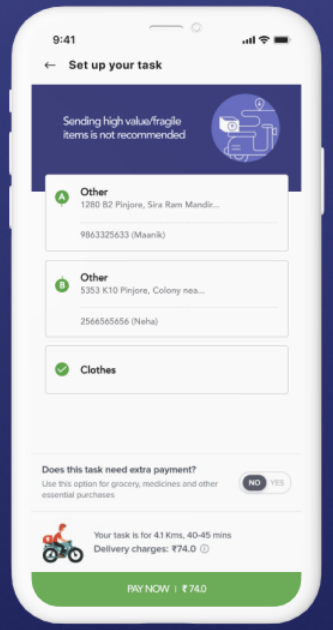



4

1. The user can add list items to be purchased and can aslo check their quantity.

2. The user can add image of the item which is to be picked up and also some description about it

3. The user can click on the dropdown button to select different options.




5

1. In this summary screen the user get the summary of pickup address and drop address.

2. Get the details of the task which is added.

3. There is a toogle button where the user can add whether the task needs a extra payment.

# PAIR THINKING / PROGRAMMING

**Unstructured**

In unstructured pair programming, the developers can trade off who takes the lead, and should discuss decisions about the code.

**Driver/Navigator**

In the driver/navigator approach to pair programming, one developer sets the architectural or strategic direction, and the other implements these decisions as code.

**Ping-pong**

Ping-pong pair programming shifts rapidly back-and-forth between the two developers, like a game of ping pong, where the software is the ball.

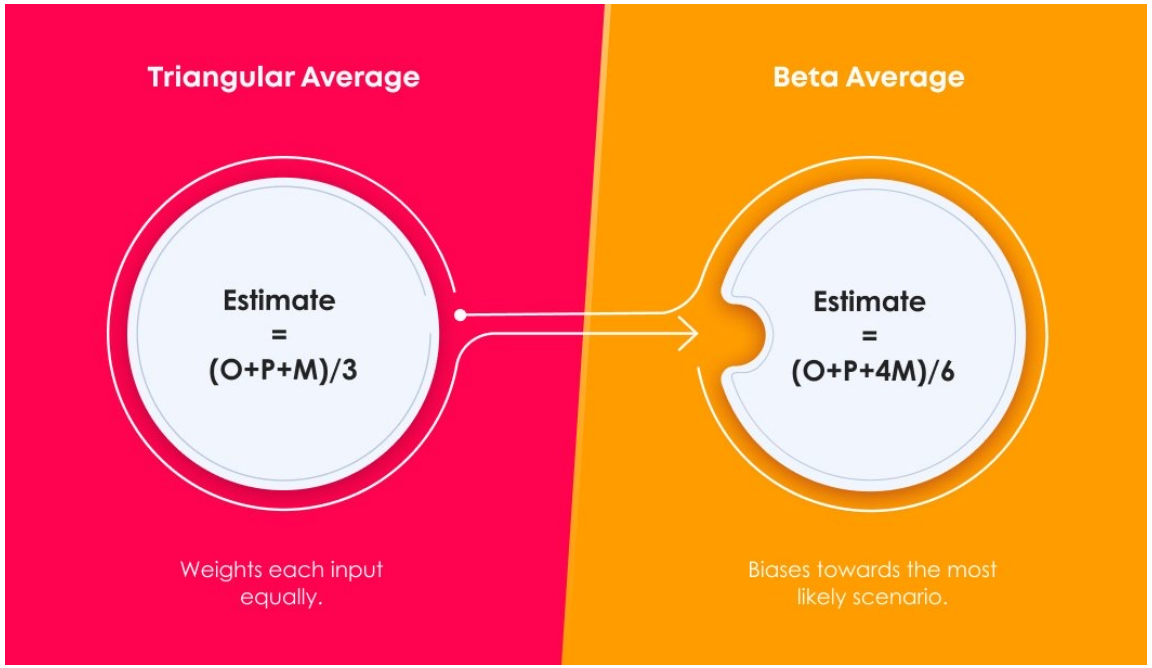
# FEATURE ESTIMATIONS - AGILE

# THREE-POINT METHOD

This method loops in the best-case scenario, the worst-case scenario, and the most likely scenario. The average of all these estimates is then calculated to give us the final estimate.

In this method, the team needs to measure time/effort based on the following parameters:

- ***Optimistic Value (O):*** How much time/effort will it take if everything is on track?
- ***Pessimist Value (P):*** How much time/effort will it take if things fall apart or there are impediments on the way?
- ***Most Likely Value (M):*** What is the most likely and practical estimate to complete the task?




**Three-Point Method use cases:**

- The team is new to Agile estimation
- Running later-stage estimations

# PLANNING POKER

- Number-coded playing cards are used to estimate an item. The cards are distributed across the team (sized 2-10), with each of the cards representing a valid estimate.
- The reading on the cards could be something such as — 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100. Now, the product owner or the analyst describes the user story to the team, and the team can ask any related queries.
- Each team member secretly selects a card number for an estimate, which is revealed when all the cards are turned over. The card with the most voting is the finalized estimate for the item under discussion.

# ANALOGY

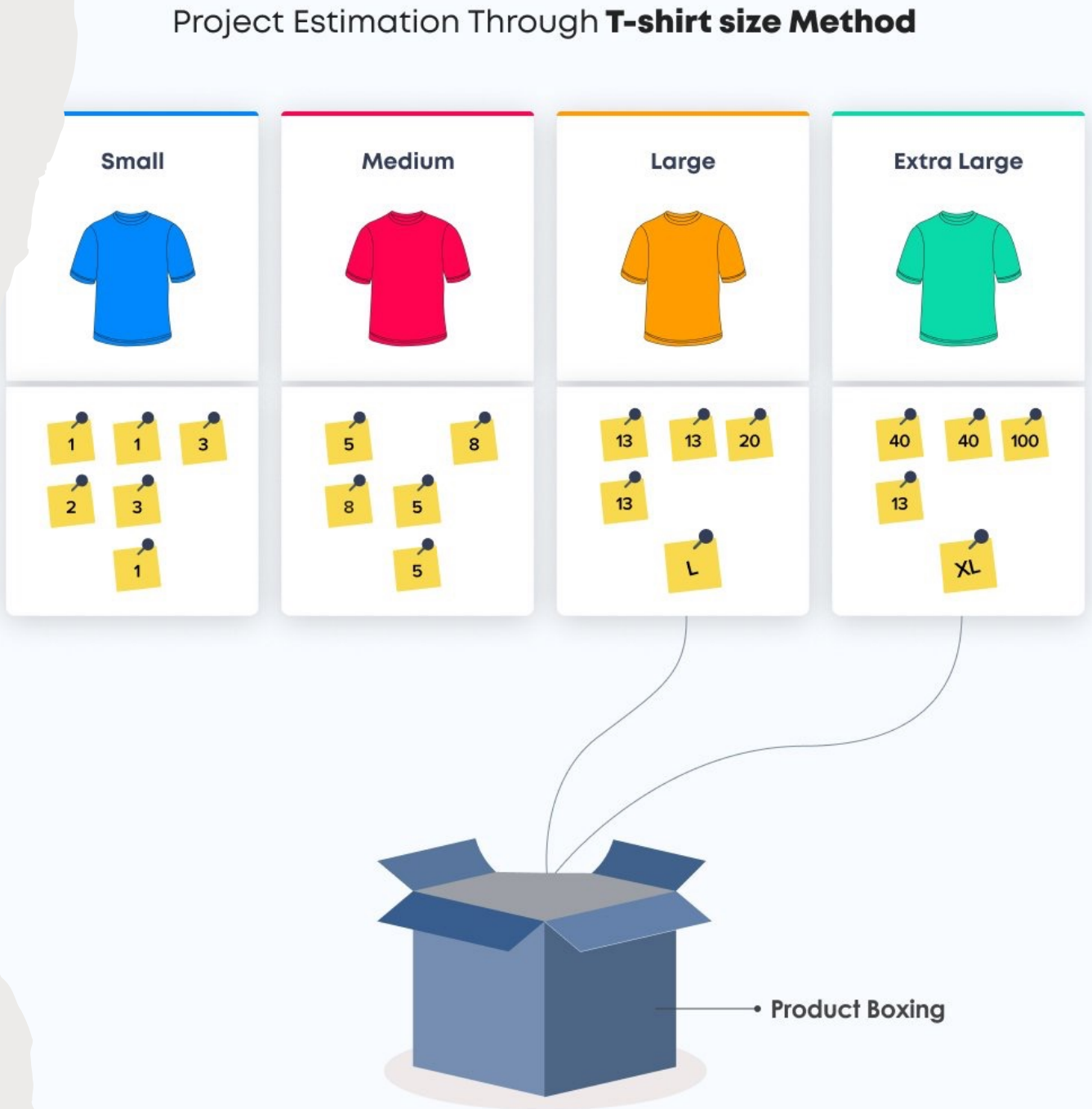
- With estimation by analogy in Agile, story sizes are compared with other stories. This relative sizing approach is helpful when making assumptions relevant to agile estimations.
- For instance, a company already estimated user story A for two weeks. Now, if they come across a user story B that is twice as large as user story A, they will assign it a larger estimation number.
- For effective Agile estimation using analogy, the triangulation method is widely used. According to the triangulation method, the user story is estimated against similar intent user stories that have already been estimated.
- For example, if the story is bigger than the story estimated at six-story points and smaller than the story estimated at 10 — estimating it at eight will be a good strategy.
- ***Analogy use cases:***
  - If retrospectives are a part of the process
  - Among teams that have a good mutual understanding
  - Among highly experienced teams

# T-SHIRT SIZE ESTIMATION

In this t-shirt sizing Agile estimation technique, the items are estimated in standard t-shirt sizes (i.e., XS, S, M, L, and XL). This is more of an informal but creative technique, and numbers can be assigned to each user story categorized under different t-shirt sizes for better understanding.

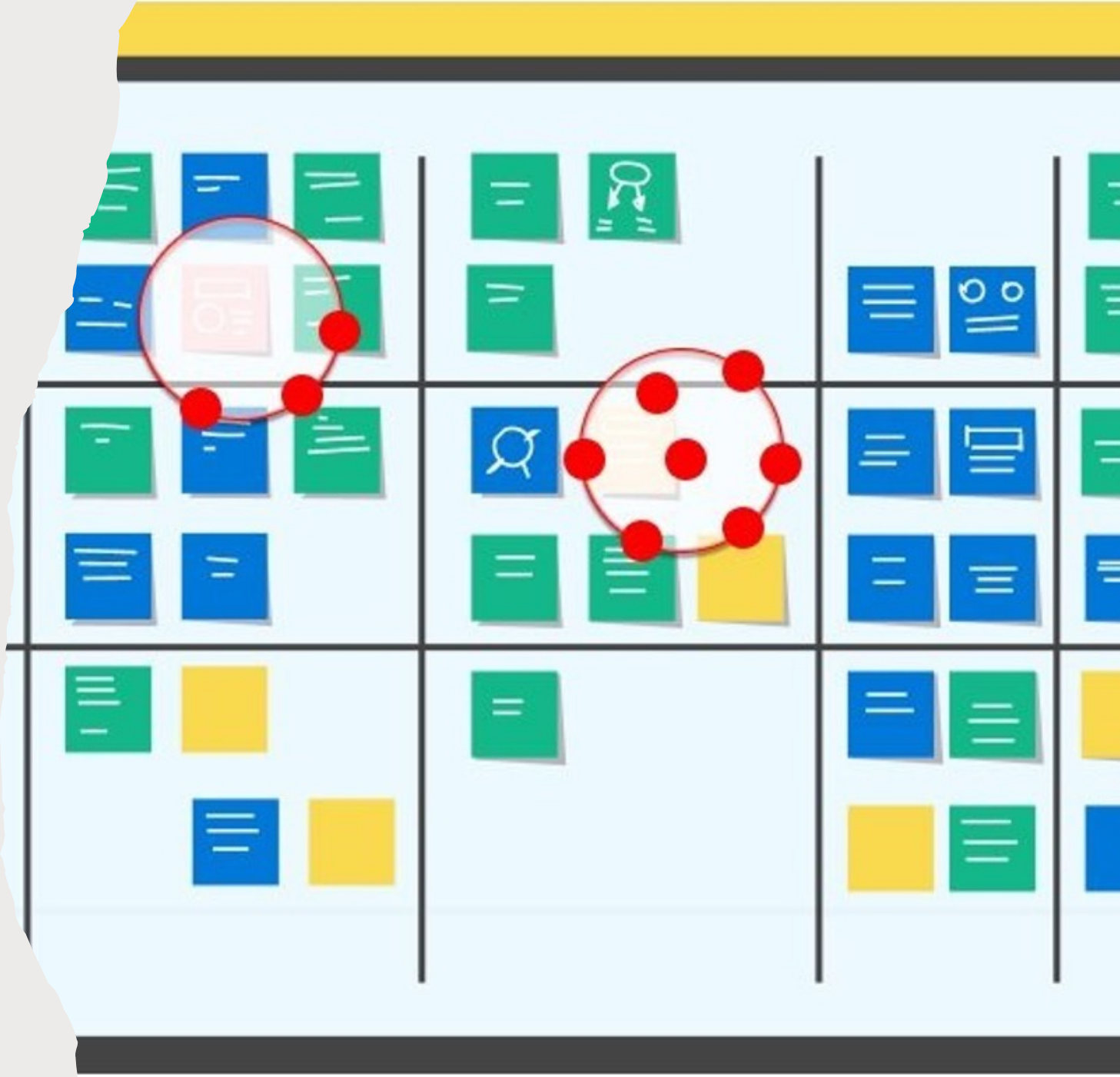
***T-Shirt size estimation use cases:***

- Running rough estimations
- The team is new to Agile estimation
- There are large backlogs
- Running early-stage estimations



# DOT VOTING

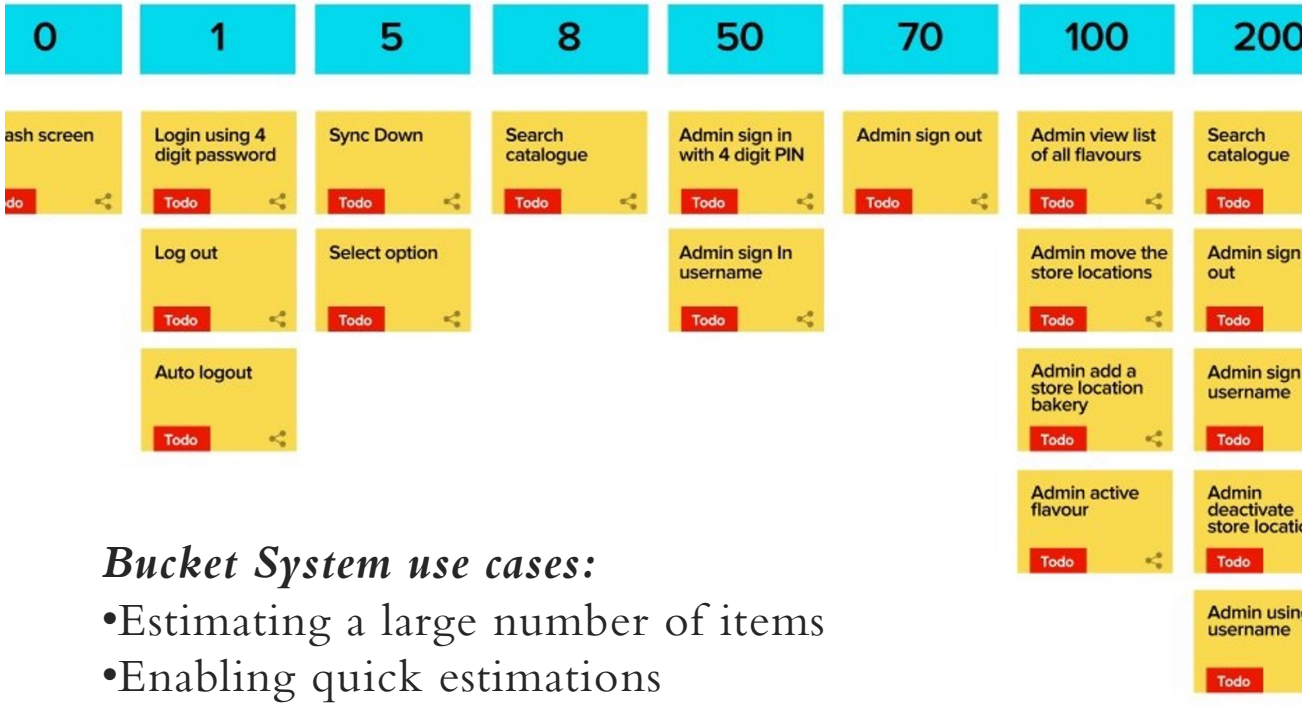
- Dot voting is a useful Agile estimation technique that works well for a small number of user stories. It is easy to implement and is effective as well.
- Here, all user stories (including descriptions) are written on post-its and placed on the wall or the board to receive votes from the team. They are given four to five dots in the form of stickers (pens or markers may also be used to create dots), which they can use to vote for the user stories of their choice.



# BUCKET SYSTEM

This Agile estimation technique can be incorporated when estimating many items (50-500) and is better than planning poker. Here, different buckets (cards) are placed sequentially with values ranging from 0, 1, 2, 3, 4, 5, 8, 13, 20, 30, 50, 100, and 200 (and more, if required). Next, according to the estimators' discretion, the user stories (items) are placed within the buckets.

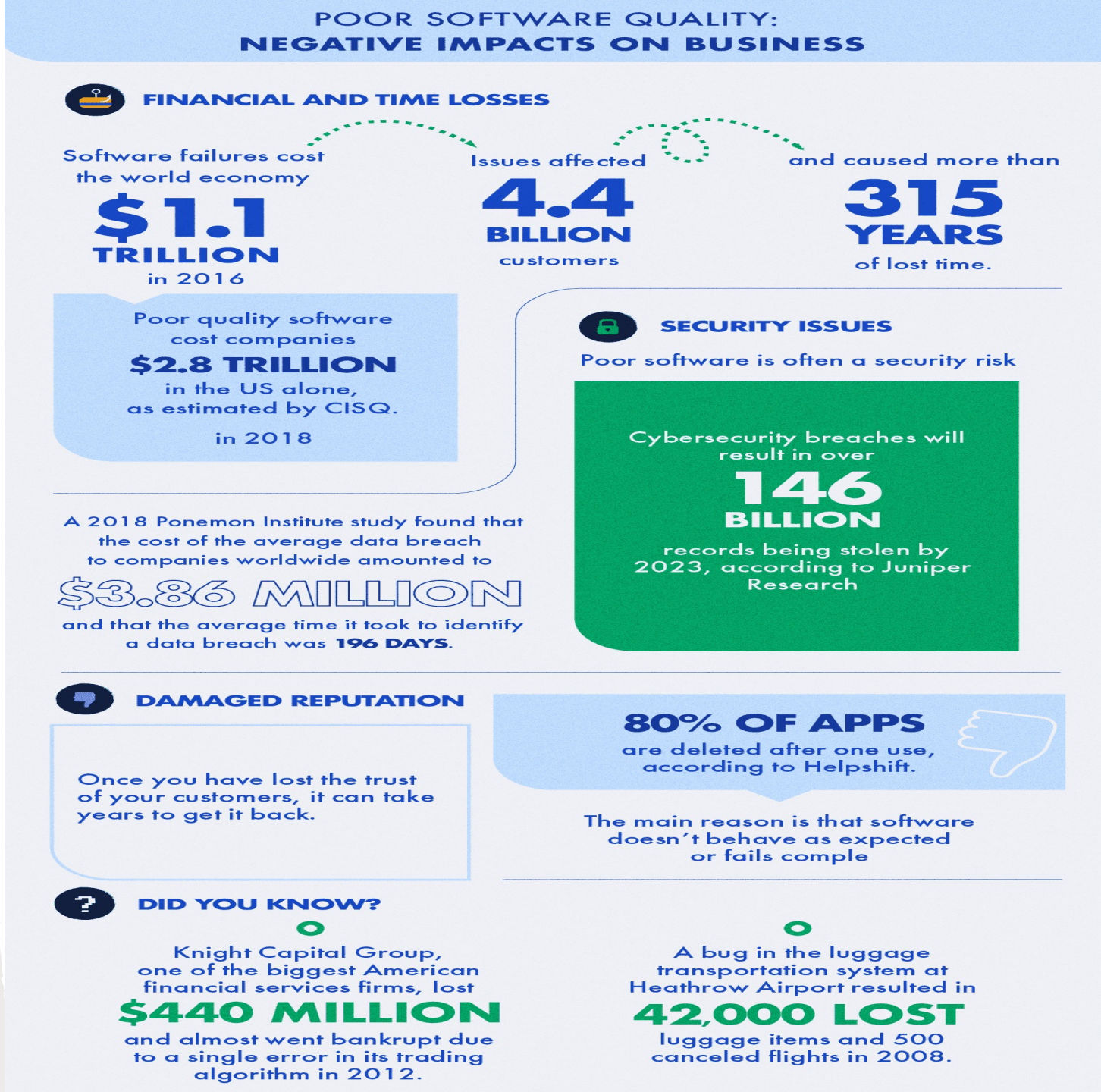
To start with, pick a random item and place it under a different bucket. Next, pick another user story, discuss all its features and requirements within the group, and place it in the bucket that suits the team's understanding.




***Bucket System use cases:***

- Estimating a large number of items
- Enabling quick estimations
- The team is new to Agile estimation
- Estimating long-term projects

# QUALITY


POOR SOFTWARE QUALITY:
NEGATIVE IMPACTS ON BUSINESS
FINANCIAL AND TIME LOSSES
Software failures cost the world economy
$1.1 TRILLION
in 2016
Issues affected
4.4 BILLION
customers
and caused more than
315 YEARS
of lost time.
Poor quality software cost companies
$2.8 TRILLION
in the US alone, as estimated by CISQ.
in 2018
SECURITY ISSUES
Poor software is often a security risk
Cybersecurity breaches will result in over
146 BILLION
records being stolen by 2023, according to Juniper Research
A 2018 Ponemon Institute study found that the cost of the average data breach to companies worldwide amounted to
$3.86 MILLION
and that the average time it took to identify a data breach was 196 DAYS.
DAMAGED REPUTATION
Once you have lost the trust of your customers, it can take years to get it back.
80% OF APPS
are deleted after one use, according to Helpshift.
The main reason is that software doesn't behave as expected or fails comple
DID YOU KNOW?
Knight Capital Group, one of the biggest American financial services firms, lost
$440 MILLION
and almost went bankrupt due to a single error in its trading algorithm in 2012.
A bug in the luggage transportation system at Heathrow Airport resulted in
42,000 LOST
luggage items and 500 canceled flights in 2008.

# HOW TO IMPROVE QUALITY

Defensive programming

Offensive programming

# DEFENSIVE DESIGN & CODING

The purpose of defensive design is to ensure that a ***program*** runs correctly and continues to run no matter what actions a user takes. This is done through planning for all possibilities (contingencies) and thinking about what a user may do that the program does not expect.

Defensive design encompasses three areas:

- protection against unexpected user ***inputs*** or actions, such as a user entering a letter where a number was expected
- maintainability – ensuring code is readable and understandable
- minimising/removing ***bugs***

This anticipation and protection is done through:

- **validation**
- **sanitisation**
- **authentication**
- **maintenance**
- **testing**

# DEFENSIVE PROGRAMMING

*Defensive programming is when a programmer anticipates problems and writes code to deal with them.*

- ***Check all data from external sources***: when obtaining data from files, networks, or external sources, check the value of the data obtained to ensure that it is within the date range. ***Example***: When we request user's age, It should be a positive number, and maybe not larger then 150?
- ***Check the parameters values for the methods***: ***Example***: Train Arrival time should be a Datetime type, not null or random integer numbers.
- ***Decide how to deal with incorrect data***: discovering an incorrect parameter, how do you deal with it? Depending on the situation, you can choose the mode that suits you, either by asserting or maybe throwing exceptions?

# OFFENSIVE PROGRAMMING

Normally, defensive programming emphasizes total ***Fault Tolerance***. A situation where the software will continue working normally in the event of failure or an error arising within its components. Offensive Programming, however, takes a different approach on that. According to offensive programming, the errors that should be handled defensively should come from outside the applications, such as user input. Errors arising from within the program itself should be exempted from total Fault Tolerance. There are two methodologies applied in Offensive Programming:

- Trusting internal data validity
- Trusting Software components

# DEFENSIVE CODING TECHNIQUES

Defensive coding allows the software to behave in a correct manner, despite incorrect input.

***Guard Clauses — checking preconditions***

- These one-liners are one of the absolute cornerstones of defensive coding. They sit at the top of your methods making sure the methods only continue executing when valid input is provided.
- They're precondition checks. Here's a very simple example, but nonetheless a real one.

```
public Author? GetAuthorOrDefaultById(string id) {
  if (string.IsNullOrEmpty(id)) return null;

  // rest of the implementation...
}
```

```
public Author GetAuthorBySpecification(AuthorSpecification specification) {
  if (!specification.IsValid()) return null;

  // rest of the implementation
}
```

```
public string DetermineGender(int input) {
  if (input == 0) return "woman";
  if (input == 1) return "man";

  return "unknown";
}
```

```
public string DetermineGender(int input) {
  if (input < 0) throw new ArgumentException();
  if (input > 1) throw new ArgumentException();

    return input == 0 ? "woman" : "man";
}
```

# APPLYING GUARDED CLAUSE

# REPLACING LOGICAL STATEMENTS WITH TABLE DRIVEN METHODS

```
// After refactoring



public string GetMonthName(int month) ⇒
  monthDictionary[month];
```

```
// Before refactoring
public string GetMonthName(int month, string language) {
  if (month == 1 && language == "EN") {
    return "January";
  } else if (month == 1 && language == "DA") {
    return "Januar";
  } else if (month == 2 && language == "EN") {
    return "February";
  } else if (month == 2 && language == "DA") {
    return "Februar";
  } else {
    return "Unknown";
  }
}
```

Logical statements, like if-else, are clumsy.

You often need to replace the growing mess with table driven design.

1. Generate a lookup table or dictionary in this case.
2. Instantiate the object using the configurations needed

```
// Read all the langauge configurations from somewhere.
// Could be from a database, json file, csv, or whatever.
var langConfig = new Dictionary<string, Dictionary<int, string>> {
  {"EN", new Dictionary<int, string> {
    {1, "January"},
    {2, "February"},
  }},
  {"DA", new Dictionary<int, string> {
    {1, "Januar"},
    {2, "Februar"},
  }},
};

// Instantiate two MonthOverview objects with different configs
var englishMonths = new MonthOverview(langConfig["EN"]);
var danishMonths = new MonthOverview(langConfig["DA"]);

// Get values
string english = englishMonths.MonthName(1); // → January
string danish = danishMonths.MonthName(1);   // → Januar
```

# DEFENSIVE CODING TECHNIQUES

***Assertions inside methods***

- You're quite familiar with assertions. Those statements at the end of unit tests. In defensive coding, they're not restricted to testing.
- You'll likely be calling other class methods and even methods provided by external libraries. In such cases, we'd like to check if our assumptions about what those methods do or return are true, before continuing with our method execution.

```
public OpertionStatus SaveAuthor(Author author) {
  if (author is null) return OpertionStatus.Failed;
  if (string.IsNullOrEmpty(author.Name)) return OpertionStatus.Failed;

  OpertionStatus status = database.Save(author);
  if (status == OpertionStatus.Failed) {
    // log statements and whatever else you'd want to do
    return OpertionStatus.Failed;
  }

  return OpertionStatus.Succeeded;
}
```

# BUILD PRODUCT FOR UNEXPECTED

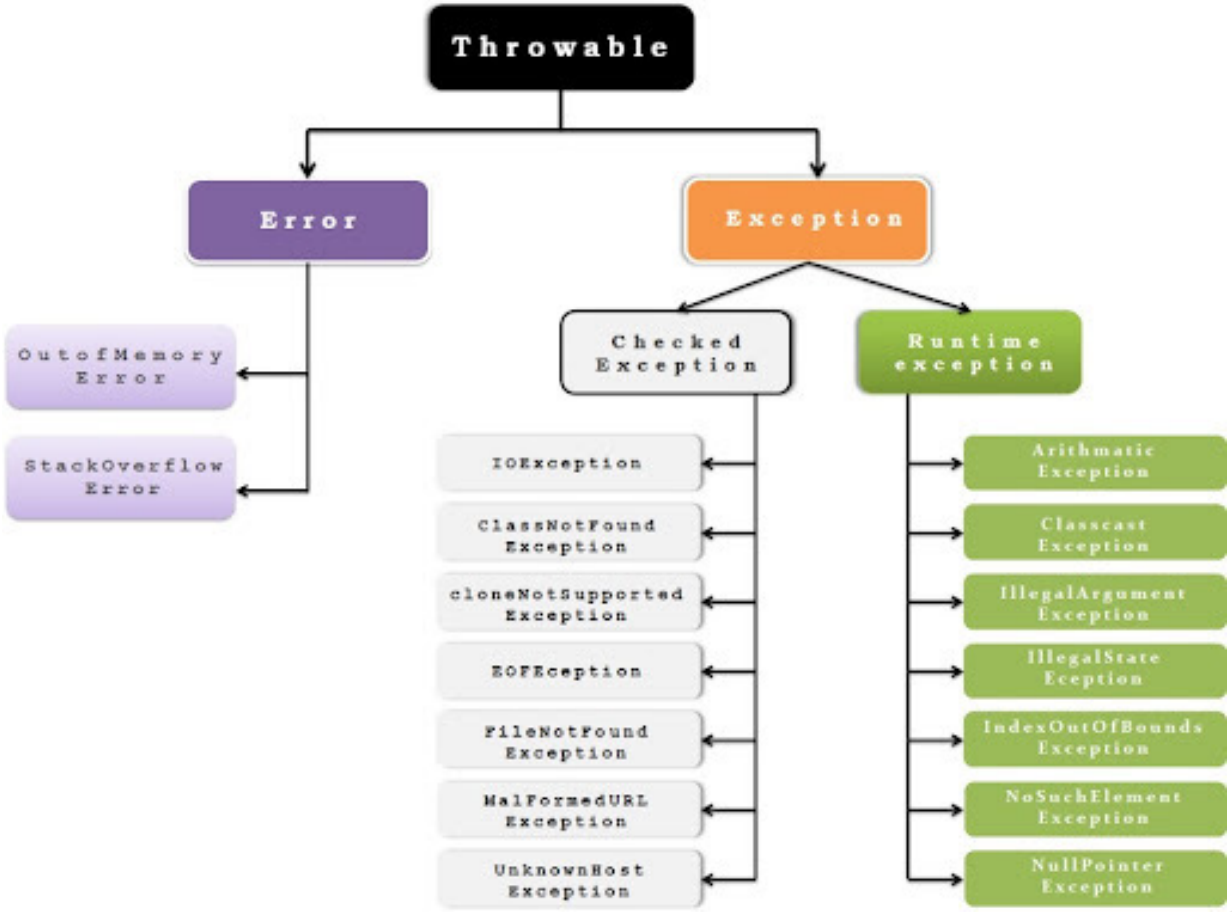
# ERRORS VS EXCEPTIONS

***ERROR***

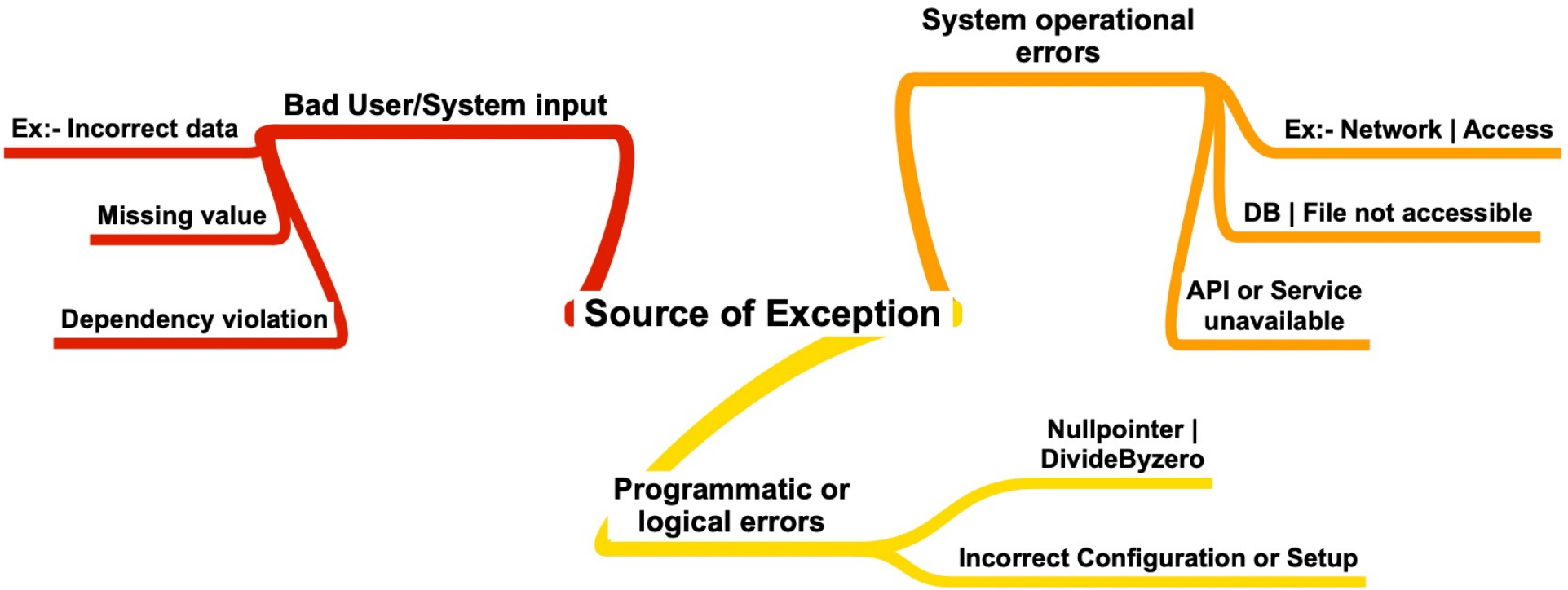
Programming errors where there is no way to recover/continue gracefully and usually need a programmer to step into and change the code to make the fix. Errors can sometimes be turned into exceptions so that they can be handled within the code.

***EXCEPTION***

Take advantage of language specific semantics and represent when something exceptional has happened. Exceptions are thrown and caught so the code can recover and handle the situation and not enter an error state. Exceptions can be thrown and caught so the application can recover or continue gracefully.



# SOURCES OF ERRORS/EXCEPTIONS



# *ARCHITECTING YOUR ERRORS*

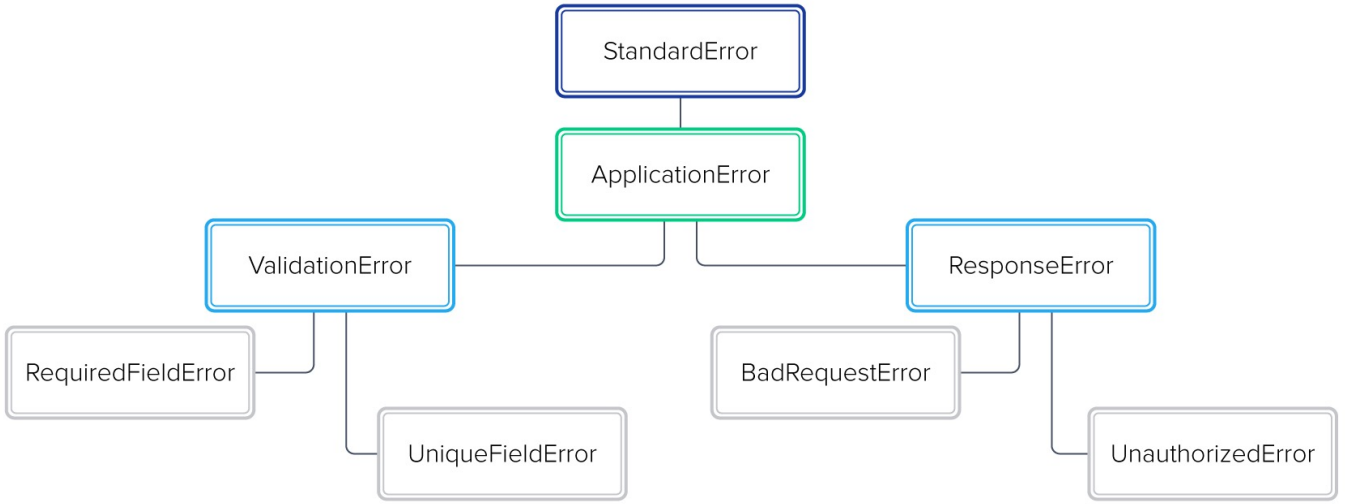
Embracing the architecting process all at once will be easier than rolling out changes one at a time. With that in mind, we have ***five tips for developing your architecting system***.

- Establish naming conventions and standard error messages across the team. Align error names with business logic and naming conventions from other systems. Naming should not be unique to any one developer;
- Always include version information and details about releases or commits in your errors;
- Identify the feature and/or unit test associated with the error;
- Create a standard for exception handling and clearly communicate this to the entire team; and
- Leverage error monitoring tools to quickly spot and respond to errors.

# BEST PRACTICES OF EXCEPTION HANDLING

- Always create your own ***ApplicationError*** hierarchy
- Never ***catch-all*** Exception
- Never ***generalise*** exceptions

```
// Bad code
try
{
    // Code that will throw exception
}
catch (Exception ex)
{
}
```

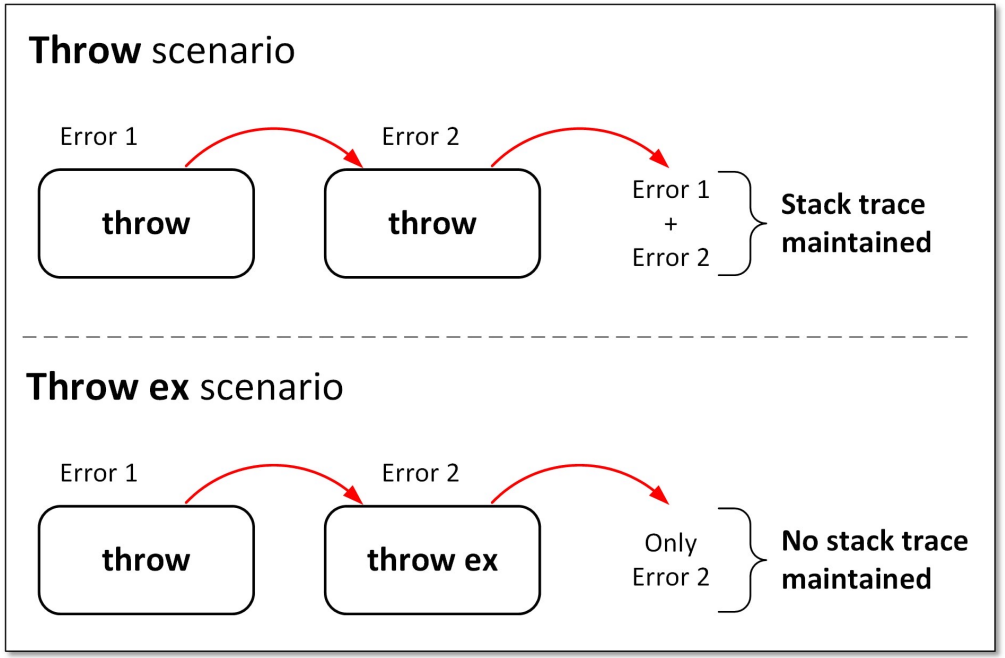


# BEST PRACTICES OF EXCEPTION HANDLING

- ***Use throw instead of throw ex***
- ***Log the exception object while logging exceptions***

```
// Good code
catch (DivideByZeroException ex)
{
    this.logger.LogError(ex, "Divide By Zero Exception occurred");
}
```

- ***Avoid catch block that just rethrows it***



# AGILE

- File upload, faster search & results
- Methods to understand -> what to be done?
- Planning Meeting -> 4 => 6 hours

FRs, NFRs, Driver-Navigator
Prioritized -> 2 weeks sprint
Estimation -> Three-point estimation
80% prioritized list & aligned ppl

***Daily Standup [4 devlopers]***

Developer A -> Validate file extension, Data ->

```
10 line code
// No pagination
fun checkExtension(){
        FOR/WHILE { method, static variable } -> CPU + Memory
        Calculation -> CPU
}
```

Developer B -> US2, Search should be done on A,B,C,D, ETA <3 sec

10mins Pair thinking -> 2 pairs [Validation 10mins], 30 mins unit testing

***Cloud – Cost Optimization***

CPU -> Cost impact
Memory -> Scaling problem, Cost
IO -> Latency, Scaling

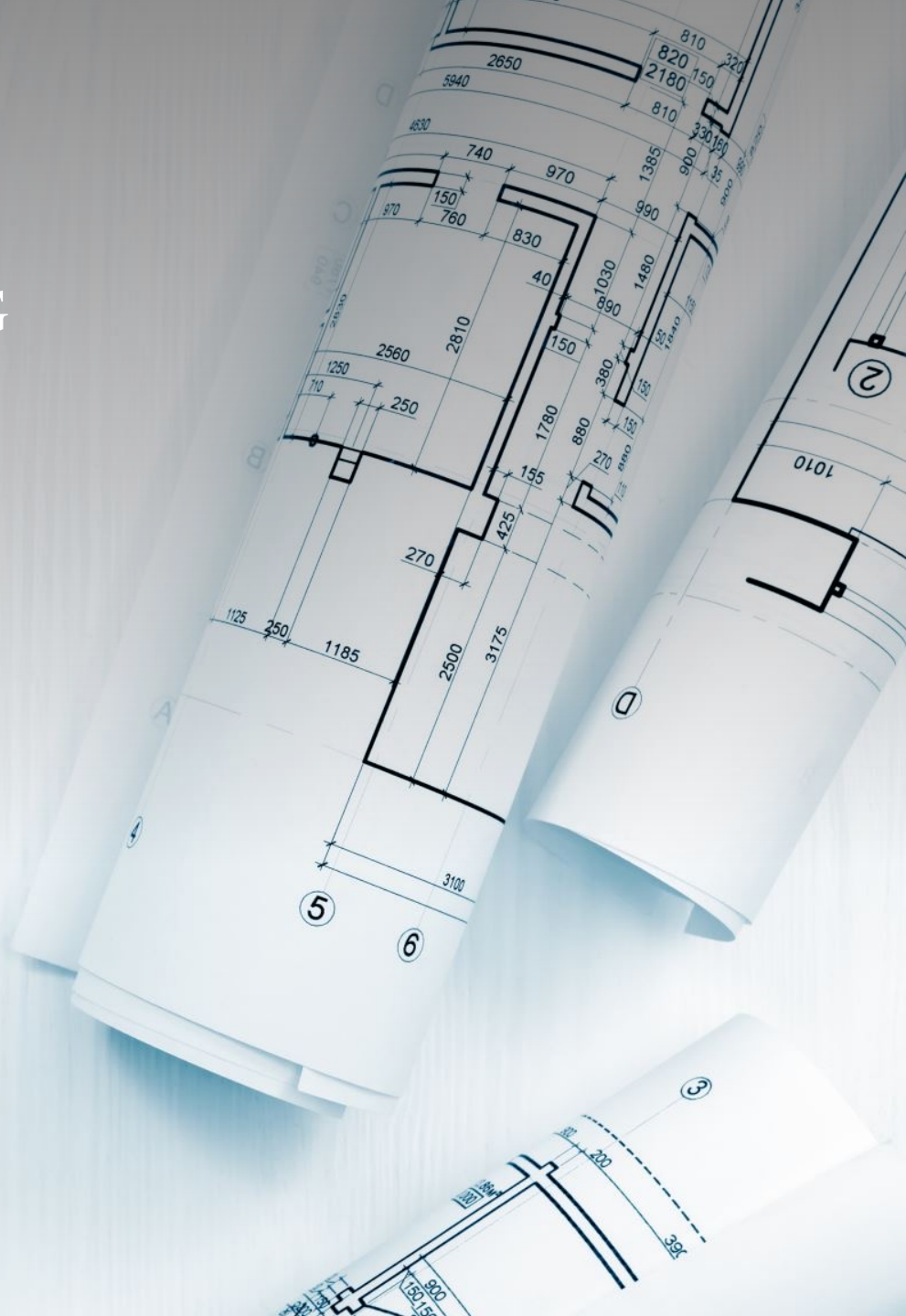
***Checklist***

- Design thinking -> planning
- Persona Analysis -> planning
- NFRs -> planning
- Pair thinking -> Daily standup

# COMMON FAILURE CONDITIONS SHOULD BE HANDLED WITHOUT THROWING EXCEPTIONS

- Close the connection only after checking if it's not already closed.
- Before opening a file, check if it exists using the File.Exists(path) method.
- Use fail-safe methods like - CreateTableIfNotExists while dealing with databases and tables.
- Before dividing, ensure the divisor is not 0.
- Check null before assigning value inside a null object.
- While dealing with parsing, consider using the TryParse methods.

# CONTINUOUS REFACTORING

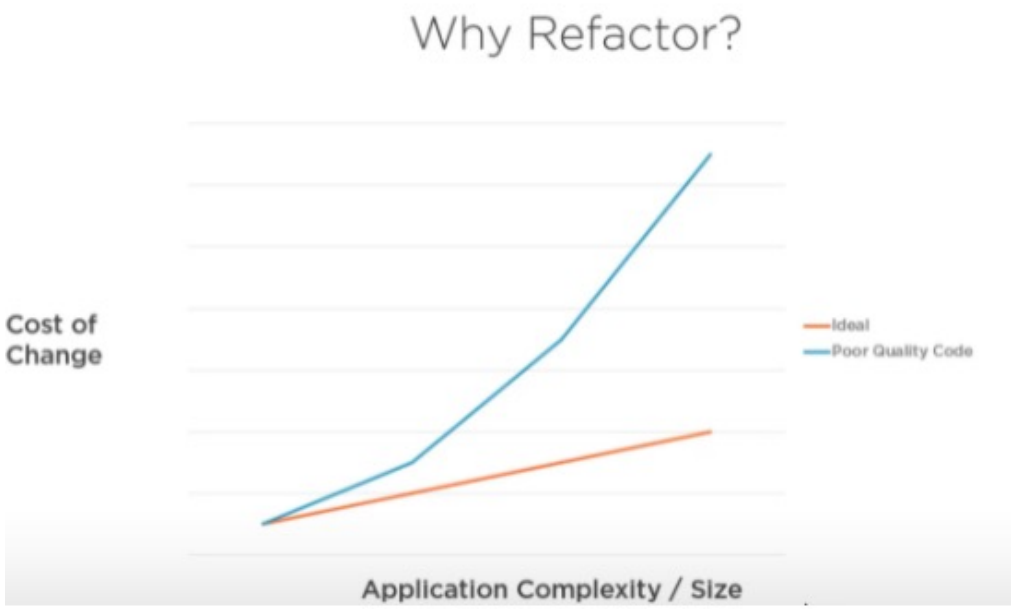


# WHY REFACTOR?

If the code works, isn't refactoring gold plating? A waste of time? A mental exercise to keep up your billing by the hour? Entertaining yourselves as you try and make your code the best from some purist standpoint, or is there real value in doing it?

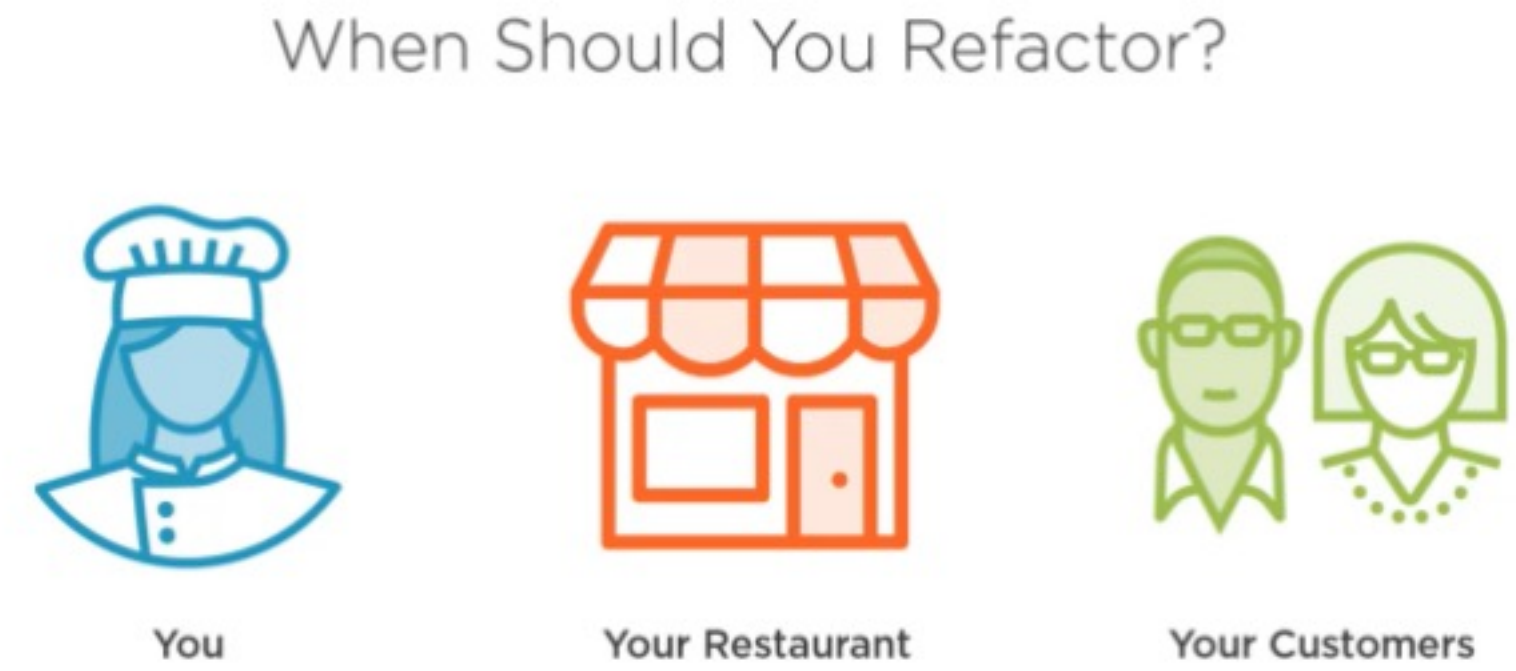
It turns out that refactoring is how you improve the design and quality of your code. And if you stop working on your code as soon as it seems to work, it's very likely that it is not well suited to future changes. And thus, future changes will be more expensive.

Without proper care, the cost of changing an application can increase exponentially in proportion to its size and complexity. Eventually, it may no longer be cost-effective to update the application further.



# WHEN SHOULD YOU REFACTOR?

If your system has much technical debt, should you stop working on new features and spend a few weeks refactoring? Sometimes this can make sense, but there are certain problems with this strategy.



# THE PROCESS OF REFACTORING


To minimize the likelihood that you'll accidentally introduce bugs as part of your refactoring, you want to follow a strict process.