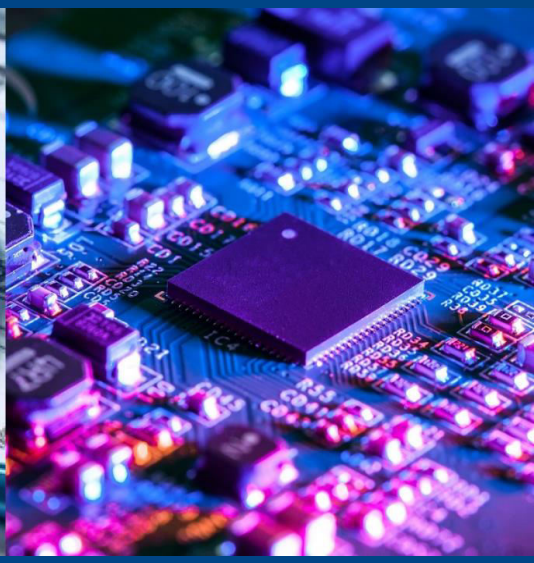
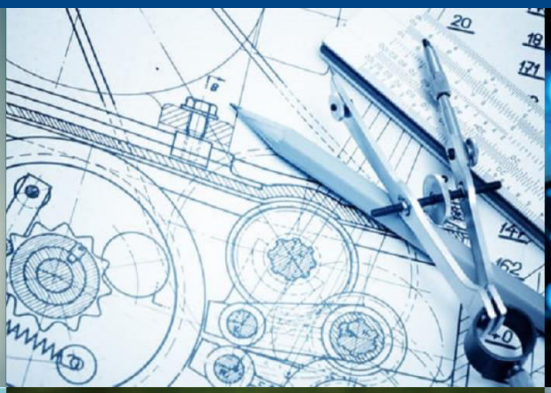
ISSN: 2582-7219

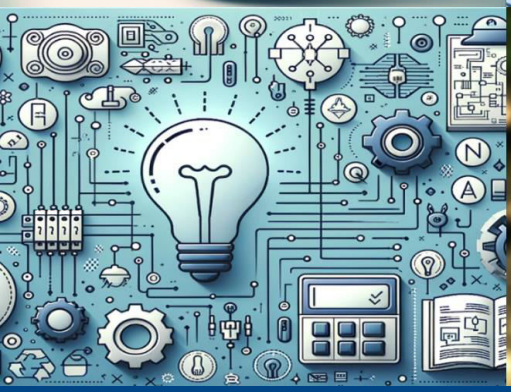




International Journal of Multidisciplinary Research in Science, Engineering and Technology

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 8.206 **Volume 9, Issue 3, March 2026**

© 2026 IJMRSET | Volume 9, Issue 3, March 2026 DOI:10.15680/IJMRSET.2026.0903202

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Kisan Sarathi Tractor Service System using Object Oriented Programming

Sameer Sanap[1], Prof. Punashri Patil[2], Ardhya Kumbhar[3], Om Katewal[4]

Assistant Professor, Dept. of Information Technology, AISSMS Institute of Information Technology, Pune, India[2]

UG Students, Dept. of Information Technology, AISSMS Institute of Information Technology, Pune, India[134]

ABSTRACT: Tractor availability remains one of the sharpest pain points in smallholder farming across rural India. Most farmers cannot afford to own a tractor, so they rent one — but finding a free tractor during peak season often means calling around for hours, relying on personal connections, and still ending up with nothing confirmed. Kisan Sarathi is a mobile application built specifically for this problem. It gives tractor owners a place to list their machines, set an hourly rate, and mark their availability. It gives farmers a way to search what is near them, check who is free, and book a tractor for a specific date and time — all without a middleman. Digital payment support through UPI, QR code, and card means money is settled cleanly with a record on both sides. The app runs on Android and was written in Java. The backend architecture uses object-oriented programming, divided into six classes: User, Farmer, Owner, Tractor, Booking, and Payment. Testing showed the system handled simultaneous booking requests correctly, maintained transparent pricing, and produced reliable payment records where none existed before.

KEYWORDS: Tractor Rental, Digital Booking, Rural Farming, OOP in Java, Android Application, Agricultural Technology, Smallholder Farmers

I. INTRODUCTION

Walk into any farming village in Maharashtra during sowing season and the same scene plays out: farmers on their phones, calling tractor owners one by one, trying to figure out who has a machine free and when. It is not a system anyone designed — it just grew out of necessity, and everyone involved knows how badly it breaks down when multiple farmers need a tractor at the same time. The owner gets pulled in different directions, the farmer who calls second loses out, and the whole thing runs on memory and personal relationships with no written record of anything.

The interesting thing we found early in this project was that the infrastructure for a digital solution already exists. Rural smartphone adoption has grown steadily across Maharashtra in recent years. People pay with UPI, use WhatsApp to coordinate everything, and search Google for information they need. What was missing was not technology literacy or device access — it was a focused tool built for this one specific transaction. Nobody had made a tractor booking app for rural India.

Kisan Sarathi is that tool. It is not an all-in-one farming platform. It does one thing: it connects farmers who need a tractor with owners who have one available. The owner registers their machine, sets a price per hour, and marks when it is free. The farmer opens the app, finds tractors nearby, sees the hourly rate upfront, and books a time slot. When the booking is confirmed, both sides receive a notification and the slot is locked so no one else can take it. Payment is handled digitally, and every transaction is stored with a timestamp. That is the whole scope of the app — and that narrow focus is intentional.

On the development side, the application was built as an Android app in Java, with the architecture organized using object-oriented programming. Six classes — User, Farmer, Owner, Tractor, Booking, and Payment — each handle their own piece of the system, which made the code much easier to build, test, and change without breaking anything unexpectedly.

© 2026 IJMRSET | Volume 9, Issue 3, March 2026 DOI:10.15680/IJMRSET.2026.0903202

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

II. LITERATURE REVIEW

Before writing a single line of code, we went through the published research on three connected topics: how tractor access affects actual farming outcomes, how mobile apps have been adopted in rural agricultural settings, and what makes digital rental coordination work well. What we found shaped most of the key decisions we made.

The starting point was Jha and Singh [1], who studied agricultural mechanization and farm productivity across rural India and found that the timing of when a tractor was available — not just whether one was available — had a measurable effect on how well crops performed in a given season. A tractor that arrives two days late during soil preparation is not a partial solution; it is a missed window. Reddy and Naik [13] reinforced this from a different angle. Their focus was on why smallholder farmers consistently struggle to maintain productivity, and their conclusion pointed squarely at equipment access as one of the primary bottlenecks — not a matter of skill or knowledge, but of not being able to get the right machine at the right time. Both papers together made a strong case that the problem we were addressing had genuine consequences for farming outcomes, not just convenience.

The second area we looked at was mobile application adoption in rural contexts. Kumar and Patel [2] reviewed a range of farm service apps deployed across India and found that the ones with real, sustained adoption among rural users were almost always narrow in their scope — they solved one clearly defined problem well rather than trying to bundle multiple features together. This directly informed our decision to keep Kisan Sarathi focused exclusively on tractor rental rather than building a broader agricultural platform. Yadav and Verma [11] pushed this thinking further by studying technology uptake among farmers with limited digital experience. Their finding was striking: the single biggest predictor of whether a farmer kept using an app after the first few sessions was interface simplicity. Complexity, even when it meant more features, drove abandonment. We kept this in mind throughout the design process — every time we considered adding something, we asked whether it would make the experience harder for someone who is not comfortable with smartphones.

Singh and Mishra [5] examined smartphone-based solutions for rural agricultural development more broadly and highlighted both the opportunities and the practical challenges of building for this user base — connectivity gaps, varying device quality, and uneven digital literacy. Their observations shaped how we thought about fallback behavior and notification reliability in our implementation. Mishra and Tripathi [12] specifically examined real-time data management strategies in agricultural mobile apps and argued that reliable availability status — keeping the displayed data synchronized with actual ground reality — is the technical foundation everything else depends on. If a farmer sees a tractor listed as available and then gets told it is already booked, trust in the system collapses quickly.

For the rental coordination side, Sharma and Verma [3] studied digital booking systems in transportation and equipment leasing and identified three features that consistently improved outcomes across the systems they examined: real-time availability visibility, a confirmed booking record that both parties can refer to, and traceable payment logs. This matched closely with what we were hearing from farmers and owners during our early conversations. Gupta and Rao [4] studied ICT tool usage among rural farmers and found that having access to accurate, current information allowed farmers to make significantly better planning decisions compared to those who had to rely on informal word-of-mouth. The difference was not subtle — farmers who could see actual availability planned earlier and experienced fewer last-minute scrambles.

From the software engineering side, Patel and Shah [6] provide a grounding in object-oriented design principles for mobile applications that informed how we structured the six modules in our codebase. Pressman [8] and the Deitel textbook [10] were our core references for implementation practice — particularly for Java-specific OOP design, class hierarchy, and Android development conventions. Booch, Rumbaugh, and Jacobson [9] guided our UML modeling phase before implementation began. The World Bank's 2020 report on digital agriculture [7] and the Android and Firebase documentation [14, 15] supported deployment decisions and the technical infrastructure for user authentication and real-time database management.

Looking across all of this, the gap in the literature was clear. There is meaningful work on farm advisory apps, on digital rental platforms for urban equipment, and on ICT adoption in rural settings generally — but almost nothing that specifically addresses a booking system designed around rural tractor hire. The seasonal demand patterns, the low level

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

of pre-existing market organization, and the specific user constraints of this context are different enough that existing platforms and research do not transfer directly. That is the space this project occupies.

III. SYSTEM DESIGN AND METHODOLOGY

The design process started from the problems, not from the architecture. We mapped out every point where the current informal rental process breaks down — missed availability, conflicting bookings, no pricing record, no payment trail — and asked what the system would need to do to address each one. Once that list was clear, the structure almost wrote itself.

A. Why Object-Oriented Programming

The reason OOP made sense here is that the problem domain is naturally made of things. A farmer is a thing. A tractor is a thing. A booking is a thing. Each of them carries its own specific data and has its own specific actions it needs to be able to perform. Trying to manage all of that in a flat, unstructured codebase would have meant constantly passing data around, duplicating logic, and making changes in one place that silently broke something elsewhere. Organizing into classes gave each piece of the system a clear boundary and a clear responsibility.

B. The Six Modules

The application is built around six OOP classes that work together to handle the full rental workflow:

1. User — handles account registration, login, and the shared identity layer for both types of users.
2. Farmer — manages the farmer-facing experience: searching tractors by location, viewing availability, and creating bookings.
3. Owner — handles the owner-facing experience: listing tractors, setting hourly rates, updating availability.
4. Tractor — stores all tractor-specific data: brand name, horsepower rating, current status, and the owner it belongs to.
5. Booking — handles the full reservation flow: time slot selection, conflict checking, status updates after confirmation.
6. Payment — processes transactions and stores a full record for both the farmer and the owner.

C. Development Steps

We started with requirement analysis — going through the failure points in the current system and building a list of what each type of user actually needs. From there we did UML class and workflow diagramming before touching any code, which turned out to be useful because it surfaced a few design conflicts early. Then we built each module separately and tested it before integrating. The final phase was integration testing, with a stress scenario where multiple farmers tried to book the same tractor at the same time — the case we were most nervous about.

IV. RESULTS AND DISCUSSION

A. Booking Conflicts

In the traditional system, double bookings happen because the owner is keeping track of everything in their head with no shared record. Our system solves this at the data layer. The moment a booking is confirmed, the tractor's status in the database updates and that time slot disappears from everyone else's search results. During testing, when two farmers submitted requests for the same tractor at the same time, the system confirmed the first one and sent a rejection to the second — automatically, every time, without any manual step in between. This was the most critical thing to get right, and it held up under concurrent load.

B. Pricing Transparency

A common complaint we heard during the design phase was about rates changing between the booking conversation and payment time. With the app, the owner sets a fixed price per hour when they register the tractor. When a farmer selects a time window, the app calculates the total cost and shows it clearly before the farmer confirms. Both sides are looking at the same number, and it does not change. This removes the main source of the pricing disputes that currently happen.

© 2026 IJMRSET | Volume 9, Issue 3, March 2026 DOI:10.15680/IJMRSET.2026.0903202

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

C. Payment Records

Before this app, there was no record of anything. Cash changed hands, and if there was a disagreement later about hours worked or rate agreed, neither side had documentation. Every payment processed through the app is saved with the amount, the timestamp, the payment method used, and links to both the booking entry and the two user accounts. Beyond dispute resolution, this also means farmers and owners start building a transaction history over time, which creates a basis for trust between people who may not have an existing relationship.

D. OOP Structure in Practice

The modular structure paid off during development in a concrete way. There were two significant redesigns mid-project — one to the Payment module when we changed how refunds were handled, and one to the Booking confirmation logic. Both times, the changes stayed entirely inside their module. Nothing else broke. In a flat codebase, either of those changes would have required tracing dependencies across the whole system. The class boundaries made the work manageable.

V. OOP CONCEPTS IN JAVA: APPLIED TO KISAN SARATHI

J

ava was the language used to build the Android application for Kisan Sarathi. The following code shows how the core OOP concepts are implemented. A Tractor base class and a RentalTractor derived class together demonstrate all the key ideas. Before going through the code, it helps to understand what each concept actually does in Java — and where Java differs from C++, since those differences matter for how the code is structured.

Listing 1: Tractor.java — Base Class

```
class Tractor {
    protected String brand;
    protected int horsepower;
    protected double rentPerHour;

    // Constructor
    Tractor(String b, int hp, double rent) {
        this.brand = b;
        this.horsepower = hp;
        this.rentPerHour = rent;
    }

    // Method Overriding — overridden in RentalTractor
    void displayDetails() {
        System.out.println("Brand: " + brand);
        System.out.println("HP: " + horsepower);
        System.out.println("Rate/hr: Rs." + rentPerHour);
    }

    // Method Overloading — same name, two signatures
    double calculateRent(int hours) {
        return hours * rentPerHour;
    }
    double calculateRent(int hours, double discount) {
        return (hours * rentPerHour) - discount;
    }

    // Java uses Garbage Collector — finalize() replaces C++ destructor
    protected void finalize() {
        System.out.println("Record cleared.");
    }
}
```

© 2026 IJMRSET | Volume 9, Issue 3, March 2026 DOI:10.15680/IJMRSET.2026.0903202

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Listing 2: RentalTractor.java — Derived Class

```
class RentalTractor extends Tractor {
   private String ownerName;

   RentalTractor(String b, int hp, double rent, String owner) {
      super(b, hp, rent);   // calls parent constructor
      this.ownerName = owner;
   }

   @Override
   void displayDetails() {
      super.displayDetails();   // runs parent version first
      System.out.println("Owner: " + ownerName);
   }
}
```

Listing 3: Main.java — Creating and Testing Objects

```
public class Main {
   public static void main(String[] args) {
      RentalTractor t1 = new RentalTractor(
         "Mahindra", 50, 800, "Sameer");
      t1.displayDetails();
      System.out.println("Total (5 hrs): Rs."
         + t1.calculateRent(5));
      System.out.println("With discount: Rs."
         + t1.calculateRent(5, 200));
   }
}
```

A. Class

A class is a template — a description of what something is and what it can do, before any actual instance of it exists. The Tractor class in our code describes what every tractor object will contain: a brand name, a horsepower value, and an hourly rental rate. It also defines what a tractor can do: show its details and calculate a rental cost. None of this refers to a specific tractor yet. The class is just the blueprint. The real tractor comes into existence only when an object is created from it.

B. Object

An object is a specific instance created from a class. When we write new RentalTractor("Mahindra", 50, 800, "Sameer"), we are not working with the template anymore — we are creating an actual tractor record with real values. The object t1 represents a specific Mahindra tractor, 50 horsepower, rented at Rs.800 per hour, owned by Sameer. In the Kisan Sarathi app, a separate object like this exists for every tractor registered on the platform, each with its own data.

C. Constructor

A constructor is a special method that runs automatically the moment an object is created. Its job is to take the values you provide and assign them to the object's fields right away. In the Tractor class, the constructor takes three parameters — brand, horsepower, and rent per hour — and immediately stores them using the this keyword, which refers to the current object being created. In RentalTractor, the constructor calls super(b, hp, rent) first to hand the first three values up to the parent class's constructor, then stores the owner name itself. You never call the constructor directly; it fires automatically with new.

D. Destructor and Garbage Collection in Java

Java does not have destructors in the C++ sense. In C++, you write a destructor method that runs when an object goes out of scope and you are responsible for freeing memory manually. Java removes this entirely. Instead, Java has a

© 2026 IJMRSET | Volume 9, Issue 3, March 2026 DOI:10.15680/IJMRSET.2026.0903202

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Garbage Collector — a background process that automatically detects when an object is no longer reachable in the program and frees its memory on its own. You do not need to do anything.

Java does provide a finalize() method which is the closest equivalent. It runs just before the Garbage Collector removes an object. In our code, finalize() prints a message to confirm the record is being cleared. In a production app, you would use it to close an open database connection or release a file handle before the object disappears — anything that needs to happen as a cleanup step. The important difference from C++ is that you cannot control when finalize() runs. The Garbage Collector decides the timing, not your code.

E. Encapsulation

Encapsulation means protecting an object's data from being changed directly by code outside the class. In our Tractor class, the three fields — brand, horsepower, and rentPerHour — are declared protected. This means they cannot be accessed or modified from outside the class hierarchy by just writing t1.brand = "something". Any change has to go through the methods the class provides. The idea is the same as a bank account: your balance is not a public variable anyone can edit freely. It is protected, and changes only happen through defined, controlled operations like deposits and withdrawals. This prevents accidental or unauthorized modification of data that other parts of the system depend on.

F. Inheritance

Inheritance lets one class build on top of another without rewriting what already exists. In Java, this is done with the extends keyword. RentalTractor extends Tractor, which means it automatically gets everything Tractor already has: the three data fields and the three methods. RentalTractor only adds the one thing that makes a rental tractor different from a generic tractor — the owner's name. The super(b, hp, rent) call in the constructor explicitly invokes the parent's constructor to initialize the inherited fields. Without this, those fields would not be set. This keeps the code clean: shared behaviour lives in one place, and the subclass only adds what is genuinely new.

G. Method Overloading

Overloading means having multiple versions of the same method that accept different inputs. In our Tractor class, calculateRent() exists in two forms: one that takes only the number of hours, and one that takes hours plus a discount amount. When you call the method, Java looks at what arguments you pass in and automatically picks the right version. If you pass one number, it uses the simpler formula. If you pass two, it applies the discount. The method name stays the same from the caller's perspective, which makes the code easier to read. You always know you are calculating rent — you just specify what kind.

H. Method Overriding

Overriding is when a subclass provides its own version of a method it inherited from the parent. RentalTractor overrides displayDetails(). In Java, you mark this with the @Override annotation directly above the method. This annotation is not strictly required — the code would work without it — but it is strongly recommended because the compiler will throw an error if you accidentally write the signature wrong. Inside the overriding method, calling super.displayDetails() first runs the parent's version and prints brand, horsepower, and rate. Then the subclass adds its own line to print the owner's name. The child is not replacing the parent's behaviour — it is extending it.

I. Polymorphism

Polymorphism means the same method call produces different behaviour depending on what kind of object is actually running it. Because RentalTractor overrides displayDetails(), calling that method on a plain Tractor object gives you three lines of output. Calling it on a RentalTractor object gives you four lines — the same three plus the owner's name. The calling code does not need to check what type it is dealing with. It just calls displayDetails() and gets the right output automatically, based on the actual runtime type of the object. This becomes especially useful when a list contains a mix of different object types that all share the same parent class — you can loop through and call the same method on each without writing separate cases.

J. Virtual Methods — Java vs C++

In C++, runtime polymorphism does not happen unless you explicitly declare a method as virtual in the parent class. If you forget the keyword, the parent's version runs regardless of the object's actual type, which is a common source of bugs. Java removes this requirement entirely. In Java, every non-static, non-private instance method is virtual by default. Runtime polymorphism is always active. If anything, Java reverses the model: instead of opting in to

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

polymorphic behaviour with virtual, you opt out of it by declaring a method final. A final method in Java cannot be overridden by any subclass. This design makes it much harder to accidentally get the wrong behaviour, and it is one of the reasons Java tends to be more beginner-friendly than C++ for object-oriented work.

VI. CONCLUSION

What stands out most when we look back at this project is not the technology itself — it is how straightforward the underlying problem was. Farmers could not find tractors when they needed them. Payments got disputed because nothing was written down. Booking conflicts happened because no one had a shared view of availability. None of these are hard problems in isolation. They just needed someone to build the right tool for this specific situation.

Kisan Sarathi addresses each of those problems directly. Booking conflicts are stopped at the data level — the moment a booking confirms, the slot locks and no one else can take it. Pricing is shown upfront and agreed before confirmation, not negotiated after the work is done. Payments create a record that both the farmer and the owner can refer back to, which removes the main source of post-rental disputes.

The Java OOP structure served the project well during development. Having six separate classes, each responsible for its own piece of the system, meant that changes were predictable and contained. When the Payment module needed reworking or the booking confirmation logic changed, those modifications stayed inside their class. Nothing downstream broke unexpectedly.

There is clearly more to build. Real users will find interface problems and edge cases we did not think of. Local language support — Marathi first, then Hindi — would make a significant difference in adoption. Offline capability matters for areas where connectivity is patchy. A farmer's history of payments and past bookings could eventually serve as a trust signal when they are renting from someone they have never dealt with before. None of that is in the current version. But what is built works, it covers the core of the problem, and it is something worth extending.

AUTHOR CONTRIBUTIONS

Sameer Sanap led the concept development, wrote the methodology, and prepared the original draft. Ardhya Kumbhar handled the Android implementation and Firebase integration. Om Katewal carried out validation and review. Prof. Punashri Patil provided supervision and project administration throughout. No external funding was received. The authors declare no conflicts of interest. All data used in testing was simulated; no real personal data was collected or stored.

REFERENCES

[1] R. Jha and P. Singh, "Agricultural Mechanization and Farm Productivity in Rural India," International Journal of Agricultural Sciences, vol. 12, no. 4, pp. 210–218, 2020.
[2] A. Kumar and N. Patel, "Mobile Applications for Agricultural Service Delivery in Rural Communities," Journal of Rural Technology and Development, vol. 8, no. 2, pp. 45–53, 2019.
[3] V. Sharma and K. Verma, "Digital Rental Coordination and Equipment Sharing in Agricultural Markets," Journal of Agricultural Innovation, vol. 5, no. 1, pp. 78–89, 2021.
[4] S. Gupta and M. Rao, "ICT Tools and Decision-Making Among Rural Farmers: An Empirical Study," International Journal of Rural Development, vol. 9, no. 3, pp. 120–131, 2020.
[5] D. Singh and T. Mishra, "Smartphone-Based Solutions for Rural Agricultural Development: Opportunities and Challenges," International Journal of Digital Agriculture, vol. 3, no. 1, pp. 14–26, 2022.
[6] H. Patel and R. Shah, "Object-Oriented Design Principles in Mobile Application Development," Journal of Software Engineering and Applications, vol. 14, no. 6, pp. 301–315, 2021.
[7] World Bank, "Digital Agriculture: Technology Transforming Food Systems," World Bank Group, Washington, D.C., Tech. Rep., 2020.
[8] R. S. Pressman, Software Engineering: A Practitioner's Approach, 8th ed. New York, NY: McGraw-Hill Education, 2014.
[9] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, 2nd ed. Boston, MA: Addison-Wesley, 2005.
[10] P. Deitel and H. Deitel, Java How to Program, 11th ed. Hoboken, NJ: Pearson Education, 2017.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

[11] S. Yadav and R. Verma, "Mobile Technology Adoption Trajectories Among Rural Farming Communities," Journal of Rural and Community Development, vol. 16, no. 2, pp. 55–68, 2021.
[12] B. Mishra and S. Tripathi, "Real-Time Data Management Strategies in Agricultural Mobile Applications," International Journal of Computer Applications, vol. 174, no. 12, pp. 22–29, 2020.
[13] C. Reddy and G. Naik, "Equipment Access Barriers and Rental Model Recommendations for Smallholder Farmers," Journal of Agricultural Economics, vol. 70, no. 4, pp. 890–902, 2019.
[14] Google LLC, "Android Developer Documentation," [Online]. Available: https://developer.android.com [Accessed: 2024].
[15] Google LLC, "Firebase Documentation: Realtime Database and Authentication," [Online]. Available: https://firebase.google.com [Accessed: 2024].

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com