Design and Implementation of an Invoice Management System for Very Good Building & Development (VGB)

Shelton Bumhe
sbumhe2@unl.edu
Defi Kapaba
dkapaba2@unl.edu
University of Nebraska—Lincoln

Spring 2025
Version 5.0

This document outlines the design and implementation of a Java-based invoice management system using object-oriented principles. The system automates billing processes, manages invoice generation, and integrates with a relational database for data storage. The system handles different types of transactions accurately using structured queries. It's built to grow easily, stay easy to update, and follow best practices for efficient invoicing and financial compliance.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Shelton Bumhe and Defi Kapaba	2025/01/03
2.0	Class Hierarchy	Shelton Bumhe and Defi Kapaba	2025/14/03
3.0	Database Design	Shelton Bumhe and Defi Kapaba	2025/28/03
4.0	Database Connectivity– Data Retrieval	Shelton Bumhe and Defi Kapaba	2025/11/04
5.0	Database Connectivity- Data Manipulation	Shelton Bumhe and Defi Kapaba	2025/25/04

Table of Contents

1. Introduction

The Very Good Building & Development Company (VGB), led by Ron Swanson, specializes in construction contracting, equipment sales, leasing, rentals, material sales, and subcontractor management. VGB currently manages its invoices and billing using spreadsheets and paper records, which limits its ability to scale, increases the risk of human error, and reduces operational efficiency. To address these challenges and modernize its financial operations, VGB has commissioned the development of a robust Invoice Management System (IMS).

This system will be designed as an object-oriented Java application, backed by a relational database for persistent storage, data integrity, and efficient query support. It will automate the entire invoicing process, from generating invoices and calculating taxes to summarizing financial reports and maintaining accurate customer and transaction records. The IMS will follow modern software engineering principles to ensure scalability, maintainability, and compliance with business and regulatory requirements.

At the heart of the system are the precise business rules that determine how costs and taxes are calculated for each type of transaction. Equipment transactions can occur as purchases, leases, or rentals. Equipment purchases are billed at the full retail price with an additional 5.25% use tax. Equipment leases calculate the cost by prorating the retail price over five years, applying a 50% markup, and then determining tax based on the total lease cost: small leases are untaxed, mid-range leases incur a flat $500 tax, and large leases are taxed at $1,500. Equipment rentals are billed by the hour at 0.1% of the retail price per hour, with a 4.38% sales tax applied to the total rental charge.

Material sales are handled by multiplying the unit price by the purchased quantity (sold only in whole units), with a 7.15% sales tax applied to the subtotal. Contract transactions, managed between VGB and trusted subcontractors, are billed at the agreed contract amount and are tax-exempt. Across all item types, the system consistently rounds monetary values to the nearest cent to ensure accuracy and compliance.

By embedding these detailed business rules into the system, VGB can guarantee that all invoices reflect correct pricing, appropriate taxes, and complete financial tracking. This ensures that customers are billed accurately, taxes are properly accounted for, and management can confidently rely on the system's outputs.

The Invoice Management System will provide automated invoice generation, detailed tax and cost calculations, and seamless data integration for financial reporting. It will maintain a comprehensive record of customers, transactions, and invoices, supporting the company's move from manual to digital operations. The system is designed with modular, reusable components,

following principles of inheritance, abstraction, encapsulation, and polymorphism to ensure code quality and ease of maintenance.

Reports generated by the system will offer valuable insights for stakeholders, allowing them to track revenue, monitor customer activity, and make informed business decisions. Designed for scalability, the IMS can grow with VGB's operations, supporting future enhancements and integration with other business subsystems.

1.1 Purpose of this Document

The primary goal of the VGB Invoice Management System is to automate and modernize the invoicing and billing process for the Very Good Building & Development Company (VGB). By replacing the current manual and spreadsheet-based system, the new solution improves efficiency, accuracy, and scalability while ensuring compliance with financial regulations.

The system is designed to:

- Generate and manage invoices for various transaction types, including equipment sales, leases, rentals, material purchases, and subcontractor contracts.
- Ensure accurate pricing and tax calculations based on VGB's business rules.
- Store and organize invoice data in a structured relational database for persistence and retrieval.
- Provide automated financial reports to help track revenue and expenses.
- Improve data integrity and security by centralizing billing information in a well-structured, maintainable system.

1.2 Scope of the Project

The VGB Invoice Management System automates invoicing and billing for equipment sales, leases, rentals, material purchases, and subcontractor contracts. It ensures accurate pricing, tax calculations, and customer transaction tracking while storing data in a structured relational database. While the system supports robust invoice and item management, report generation, and JSON/XML data exchange, it does not include user authentication, a graphical user interface, real-time cloud integration, or a live EDI messaging system. All operations are performed locally, and data exchange must be manually initiated

1.3 Definitions, Acronyms, Abbreviations

- **UID :** A 128-bit identifier used to uniquely identify items..
- **EDI:** The electronic transfer of structured data between organizations, typically in a standardized format such as XML or JSON.

- **JDBC:** A Java API that enables Java applications to connect to and interact with relational databases.
- **JAR:** A file format used to bundle Java class files, metadata, and resources into a single compressed file for distribution.
- **JUnit:** A widely used testing framework for Java applications, which helps in performing unit testing of code.
- **Tax Rate:** A percentage of the price of an item or service that is added to the total cost for tax purposes. Different tax rates are applied based on the type of product or service.

1.3.1 Definitions

- **Invoice:** A document issued by a business to a customer that lists the items or services provided, along with the total amount due, including taxes and other charges. It serves as a request for payment.
- **Equipment:** Refers to construction machinery, both heavy (e.g., bulldozers, backhoes) and small (e.g., concrete mixers, compactors)
- , available for purchase, leasing, or renting. Each piece of equipment is uniquely identified by a model number and retail price.
- **Materials:** Construction supplies
- **Contract:** A formal agreement between VGB and a subcontractor for the performance of certain services related to a construction project. It includes the terms of payment and specific work to be performed.
- **Tax:** A financial charge imposed by a government on products or services. The invoicing system applies various tax rates based on the type of product (e.g., equipment, materials) and the nature of the transaction (e.g., purchase, lease, rental).
- **Customer:** A business entity that purchases goods or services from VGB. The customer is represented in the system by their company name, address, and primary contact.
- **Lease:** An agreement between VGB and a customer in which the customer pays for the use of equipment for a specified period, with costs amortized over a 5-year period, prorated to the lease term.
- **Rental:** A transaction where a customer rents equipment for a certain number of hours. The hourly rate is calculated as a percentage of the equipment's retail price.
- **UUID (Universally Unique Identifier):** A 128-bit identifier that uniquely identifies each item in the system, such as equipment, materials, and invoices. It is used to ensure that each item can be distinctly recognized.
- **Relational Database:** A database structured to recognize relationships between stored data items. Data is stored in tables, and the relationships between different types of data (e.g., customers, items, invoices) are established via keys.
- **JAR (Java Archive):** A file format that packages multiple Java classes, metadata, and other resources (like images or configuration files) into a single compressed file. It simplifies the distribution and execution of Java applications.

- **API (Application Programming Interface):** A set of protocols and tools that allow different software applications to interact with each other. In this project, the API used to interface between the application and the database.
- **Tax Rates:** The percentages applied to equipment purchases, leases, rentals, and materials to determine the applicable tax amount. These rates vary depending on the type of product or service provided.
- **Inventory System:** A system that manages the records of materials and equipment available for sale, lease, or rent, including tracking their quantities and prices.
- **Invoice Management System:** A software system designed to manage all aspects of the invoicing process for VGB, including generating invoices, applying taxes, managing customer records, and providing reports.

1.3.2 Abbreviations & Acronyms

UUID - Universally Unique Identifier
JAR - Java Archive
API - Application Programming Interface
EDI - Electronic Data Interchange
VGB - Very Good Building & Development Company
B2B - Business-to-Business
DB - Database
SQL - Structured Query Language
CRUD - Create, Read, Update, Delete
JDBC - Java Database Connectivity

2. Overall Design Description

This project involves creating a Java-based invoice management system using object-oriented design principles. The system loads data from CSV files, convert them into objects, and serialize the data into XML or JSON format.

It includes several major components:

- **Inheritance & Polymorphism**
  The class hierarchy is structured to promote code reuse and flexibility through inheritance. Shared attributes and methods, such as item identifiers, names, and base prices, are placed in abstract superclasses. Subclasses like Equipment, Material, and Contract inherit these features while implementing their own unique behaviors, such as tax calculation or cost computation based on purchase type. Polymorphism

allows the application to treat all items uniformly through a common interface, enabling consistent handling in invoices and reports, while ensuring that each item responds according to its type-specific logic.

- **Encapsulation & Abstraction**
  Each class is designed to protect its internal state by encapsulating data within private fields and exposing only the necessary methods for interaction. This prevents unintended manipulation of data and maintains consistency throughout the system. Abstraction is achieved by providing meaningful public methods that describe *what* an object does, not *how* it does it. For example, an invoice object provides a method to calculate its total amount, while hiding the logic that handles item subtotals, taxes, and formatting. This separation of concerns simplifies usage and reduces coupling between classes.
- **Composition & Relationships**
  The system models real-world relationships by building complex classes from simpler components. For instance, an Invoice object holds a list of Item objects, and it is linked to both a Customer and a Salesperson. Each relationship reflects a meaningful connection within the business domain. Composition enables this modular structure, allowing updates or extensions to one part of the system (e.g., adding a new item type) without affecting unrelated components. This approach supports scalability and aligns with best practices in object-oriented design.
- **Reusability & Maintainability**
  Classes are written to be self-contained and modular, making them easy to reuse in different contexts or future projects. The use of interfaces, abstract classes, and flexible constructors ensures that classes can evolve without breaking existing functionality. For example, an Item class can be reused in both file-based and database-driven implementations with minimal changes. This design allows the system to scale over time — from simple report generation to advanced features like GUI integration or cloud-based data storage — without requiring a full rewrite.

Technologies:

- **Java**: Primary programming language used for implementing object-oriented classes that represent real-world entities such as invoices, items, customers, and employees.
- **JUnit**: Used to create structured unit test suites that ensure each class functions correctly and follows OOP principles.
- **XStream / Gson**: Used for serializing objects into XML and JSON formats. These tools are essential for exporting invoice data and integrating with other systems through standardized formats.
- **MySQL**: While implemented in later phases, the current class design is made compatible with MySQL by modeling data relationships in a way that maps directly to database tables.
- **JDBC:** Used for connecting the Java application to the MySQL database. JDBC enables reading and writing invoices, items, customers, and employee data directly from the database by executing SQL queries through Java code.

- **Custom Sorted List ADT** replaces standard Java collections by providing a self-implemented sorted list data structure that maintains elements in a defined order using comparators. This data structure is critical for generating summary reports that rank invoices by total, by customer name, and by company-wide totals, all while maintaining performance and encapsulation without relying on built-in Java sorting or collection utilities.

Design Principles:

- Inheritance and polymorphism were used for shared behavior across entities.
- Encapsulation and abstraction were applied to control access to class data.

Outputs:

- Invoice and item data were exported to XML and JSON files.
- A MySQL database schema was created to store structured data.

Class Structure:

- Classes like Person, Company, Item, and Invoice were built using inheritance and specialization.

Database:

- Data was stored and retrieved from a MySQL database using JDBC connections.

- Testing: JUnit tests will verify functionality, focusing on entities and invoices.
- **Technologies:** Java, XStream/Gson for serialization, MySQL for database.

2.1 Alternative Design Options

- **Data Representation**: Considered using flat files or CSV over XML/JSON. Flat files are simpler but lack structure for complex data. XML/JSON were chosen for better scalability and hierarchical data handling.
- **Database Design**: Considered NoSQL (e.g., MongoDB) over MySQL. NoSQL offers flexibility but lacks support for relational data and complex queries. MySQL was chosen for its relational capabilities and data integrity.

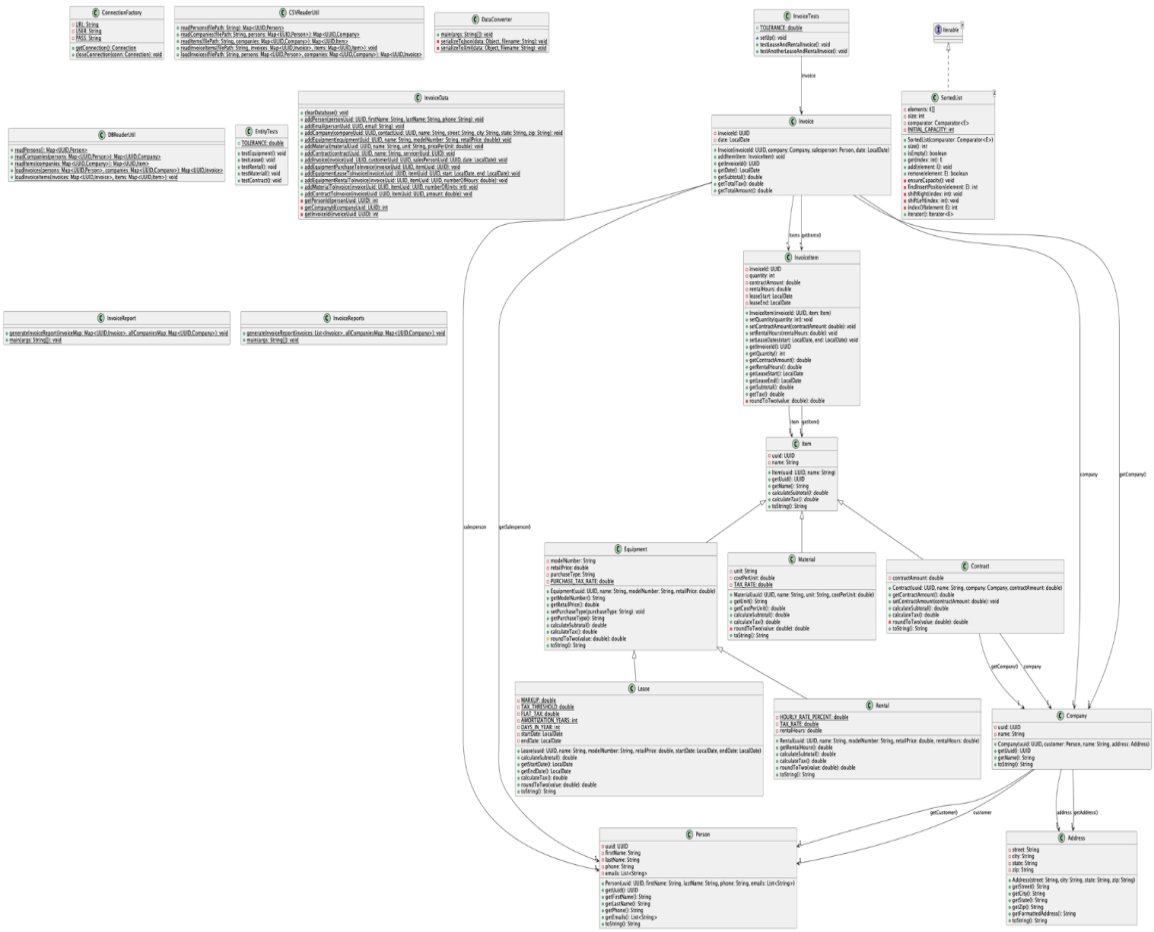
- **Sorted List ADT**: Considered using Java's ArrayList or TreeMap. These are simpler but less efficient for our custom sorting needs. A custom Sorted List ADT was chosen for better control and performance.
- **Database Connectivity**: Considered using Hibernate instead of JDBC. Hibernate simplifies queries but adds overhead. JDBC was chosen for better performance and direct query control.
- **Architecture**: Considered a monolithic architecture. It's simpler but harder to scale. A modular design was chosen for better scalability and easier integration with future subsystems.

3. Detailed Component Description

The ER diagram for the design is shown in Section 3.1. In section 3.2, the class model of the projects is demonstrated. Section 3.3 covers the database interface. Each section includes its own testing strategies.

3.1 Database Design

The database follows a normalized relational schema designed to efficiently manage invoicing data for the system. It models key entities such as customers, employees, invoices, and items using primary and foreign keys to enforce referential integrity. The design supports scalability, data accuracy, and ease of maintenance.



3.1.1 Component Testing Strategy

Approach:

Used unit testing for validating CRUD operations.

Integration testing for verifying entity relationships.

Edge case handling for missing, duplicate, or invalid data.

Test Cases:

Ensure invoices link correctly to customers and salespersons.

Validate aggregations (e.g., total sales per salesperson).

Detects invalid duplicate material purchases within invoices.

Test Data Generation:

To validate the system's database design and ensure robust functionality, we populated the database with realistic test data using carefully crafted SQL insert scripts and, where applicable, random data generators to simulate diverse and non-trivial cases. This ensured the database could handle multiple customers, invoices, items, and transactions across all supported business types. To thoroughly exercise the schema, we implemented a set of structured SQL queries, including retrieval of all people and their emails, fetching items on specific invoices, identifying total purchases by customer, counting invoices and sales by customer and salesperson, calculating material subtotals, and detecting data anomalies such as duplicate material entries or potential fraud cases where a salesperson transacts with their own company. These queries are organized in the queries.sql file, clearly numbered and commented, to demonstrate the database's ability to support both operational needs and compliance monitoring. Database interaction methods were tested using both real and mock data, and adjustments were made to improve error handling, optimize query performance, and ensure consistency with the business logic defined in the application layer.

3.2 Class/Entity Model

The system applies object-oriented principles, with distinct classes representing database entities.

- Class Diagram Overview:
- Person (Superclass) → Customer, Employee (Subclasses).
- Invoice links Customer (buyer) and Employee (salesperson).
- InvoiceItem as a junction table links Invoice and Item.
- Item (Abstract Class) → Contract, Equipment (Subclasses).
- Company linked to invoices for business transactions.
- Single Responsibility Principle:
- Each class focuses on one concern, ensuring maintainability and flexibility.

3.2.1 Component Testing Strategy

Unit Tests: Validate individual class behaviors and method outputs.

Integration Tests: Ensure proper relationships between entities.

Test Data: Used mock data generators.

Results: Identified redundant attributes, leading to table refinements.

3.3 Database Interface

- API Design:
- JDBC
- **API Design:**
- Encapsulates all database interactions for efficiency
- Provides CRUD operations for invoices, customers, employees, companies, and items
- Uses a `ConnectionFactory` class to manage and reuse database connections

Handling Edge Cases:

- SQL injection prevention via parameterized queries
- Defined NOT NULL, CHECK, and FOREIGN KEY constraints to prevent inconsistencies
- Null values are checked in Java and handled with defaults or logs to prevent crashes
- Performance Enhancements:

Performance Enhancement:

- Indexed frequently queried columns such as invoice_id, customer_id, salesperson_id, and company_id to speed up lookups and improve overall database query performance.
- Used join queries to efficiently combine related data (such as invoices with items or customers) in a single query, reducing repeated database calls and improving data retrieval speed.
- Implemented factory methods like loadInvoices(), loadCustomers(), and loadItemsByInvoiceID() to load database records into fully structured Java objects, making the data ready for report generation and further processing.
- Integrated a logging system using java.util.logging to track query execution, record system errors, and log key activity, supporting easier debugging and system monitoring.
- Expanded the database interface to support full data manipulation (CRUD), enabling the system not only to retrieve but also to add, update, and delete records efficiently.
- Developed methods like addPerson(), addCompany(), addInvoice(), addItem(), and addInvoiceItem() using JDBC and parameterized queries to ensure secure insertion and modification of data, protecting against SQL injection.

- Provided a clean API layer (via InvoiceData.java) that abstracts all low-level database interactions, so external tools and grading systems interact only through high-level, simplified methods.
- Tested JDBC operations systematically by inserting known test records, running retrieval queries to check that the data was loaded correctly into Java objects, and verifying that updates and deletions properly modified the database state.
- Designed test cases to deliberately trigger error scenarios (such as violating foreign key constraints or inserting duplicate records) to ensure that the system handles exceptions correctly and logs issues without crashing.
- Confirmed that after each add, update, or delete operation, follow-up queries accurately reflected the expected database state, validating that the JDBC methods worked as intended.
- Ensured that all CRUD operations remain fully abstracted from the user interface, exposing only clean, safe, and high-level methods while managing detailed database logic internally within the system.

3.3.1 Component Testing Strategy

To ensure the reliability and correctness of individual components, we designed and executed a structured component testing strategy. Each major module — including data retrieval, data manipulation, and sorting — was tested independently using controlled test cases. For the database interaction layer, we tested methods responsible for adding, retrieving, updating, and deleting records by inserting known test data and verifying that the database state matched expectations after each operation. We also designed test cases to trigger expected failures, such as foreign key constraint violations or duplicate entry attempts, to confirm that error handling worked correctly and that exceptions were properly logged.

For the sorting component, we created focused test cases to validate that invoices, customers, and summary reports were sorted correctly according to defined comparators (such as total amount, customer name, or date). We checked that the sorted list maintained its order as elements were added, that ties were consistently broken using UUIDs, and that retrieval operations returned the correct sorted sequence without needing additional post-processing.

Overall, the component testing strategy ensured that each module functioned correctly on its own, paving the way for smooth integration and reliable end-to-end system behavior.

3.4 Design & Integration of a Sorted List Data Structure

Design:

API Design (JDBC):

- Encapsulates all database interactions in a clean, abstracted API layer
- Provides full CRUD operations: Create, Retrieve, Update, Delete
- Uses a ConnectionFactory class to centralize and reuse database connections

Adding (Create):

- Methods like addPerson(), addCompany(), addItem(), addInvoice(), and addInvoiceItem() use INSERT statements with parameterized PreparedStatements
- Java-side input validation ensures data integrity before insertion
- Catches and logs exceptions during insertion to detect issues

Retrieving (Read):

- Factory methods such as loadPersons(), loadCompanies(), loadItems(), loadInvoices() use SELECT queries with JOINs to efficiently load related data
- Maps ResultSet data into Java objects reflecting database relationships
- Provides ready-to-use Java objects for the rest of the application

Updating (Update):

- Uses UPDATE statements with parameterized queries to modify existing records (e.g., updating email addresses or item details)
- Ensures only valid fields are updated and foreign keys are respected
- Logs any update errors and maintains consistent database state

Deleting (Delete):

- Methods like removeAllPersons(), removeAllCompanies(), removeAllInvoices() use DELETE statements with proper ordering to respect foreign key constraints
- Uses transactions when needed to prevent partial deletions and maintain integrity

Edge Case Handling:

- Uses parameterized queries to prevent SQL injection attacks
- Relies on NOT NULL, CHECK, and FOREIGN KEY constraints in the schema to enforce integrity
- Java methods check for null values and apply default values or log warnings to prevent crashes

- Implemented as a **linked list-based** sorted structure.

- Supports dynamic insertion, deletion, and sorting of invoices.

Interface & Performance:

- Optimized sorting logic for efficient retrieval by implementing a custom Sorted List Abstract Data Type (ADT) that maintains elements in sorted order as they are added, rather than sorting them later. This design ensures that invoices, customers, and company summaries are always available in the correct order for reports, eliminating the need for additional sorting steps during report generation. The sorted list is constructed using a comparator provided at instantiation, allowing the system to define flexible sorting rules such as ordering invoices by total amount, by customer name, or by company totals. Because the list maintains order internally, retrieval operations can directly iterate over the list in sorted sequence, reducing computational overhead and improving overall performance..

3.4.1 Component Testing Strategy

Component Testing for Sorting

- **Test Cases:**
  - We created test cases to validate that sorting logic correctly orders invoices based on total amount, customer name, and invoice date. These tests included verifying that the sorted list maintained correct ordering when new elements were added, that ties were properly resolved using UUIDs as secondary keys, and that retrieval operations produced the expected sorted sequence without additional adjustments.
- **Outcomes:**
  - Based on the test results, we refined the internal traversal and insertion logic within the sorted list implementation to minimize redundant operations and ensure that the list maintains order as efficiently as possible. These improvements reduced unnecessary shifting or comparisons during element addition and retrieval, enhancing overall system performance and ensuring the sorted data was always ready for use in summary and report generation.

4. Changes & Refactoring

Optimized API Calls: Reduced redundant queries for faster execution.

Refactored Entity Classes: Eliminated unnecessary attributes.

Enhanced Error Handling: Improved SQL transaction rollback mechanisms.

Extended database interface to support full CRUD operations by implementing the InvoiceData API.

Bibliography

[1] *APA 6 – Citing Online Sources*. (n.d.). Retrieved March 19, 2021, from https://media.easybib.com/guides/easybib-apa-web.pdf

[2] Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

[3] Internet Goons. Do I cast the result of malloc? http://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc. [Online; accessed September 27, 2015].