Architectural Views

February 23, 2026

Richard Thomas & Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Architectural Views

Software Architecture

February 23, 2026

Richard Thomas & Brae Webb

1 Introduction

Understanding software is hard. It is often claimed that reading code is harder than writing code[1]. This principle is used to explain a programmer's innate desire to constantly rewrite their code from scratch. If software is hard to understand, then understanding software architecture from the detail of the code is near impossible. Fortunately, architects have developed a number of techniques to manage this complexity.

A software architecture consists of many dimensions. Programming languages, communication protocols, the operating systems and hardware used, virtualisation used, and the code itself are a subset of the many dimensions which comprise a software architecture. Asking a programmer's monkey brain to understand, communicate, or document every dimension at once is needlessly cruel. This is where architectural views come in.

Architectural views, or architectural projections, are a representation of one or more related aspects of a software architecture. Views allow us to focus on a particular slice of our multi-dimensional software architecture, ignoring other irrelevant slices. For example, if we are interested in applying a security patch to our software then we are only interested in the view which tells us which software packages are used on each host machine.

The successful implementation of any architecture relies on the ability for the architectural views to be disseminated, understood, and implemented. For some organisations, the software is simple enough, or the team small enough, that the design can be communicated through word of mouth. As software becomes increasingly complex and developers number in the thousands, it is critical for design to be communicated as effectively as possible. In addition to facilitating communication, architectural views also enable architectural policies to be designed and implemented.

2 C4 Model

Simon Brown's C4 model provides a set of abstractions that describe the static structure of the software architecture [3]. The C4 model uses these abstractions in a hierarchical set of diagrams, each leading to finer levels of detail. The hierarchical structure is based on the idea that a software system is composed of containers, which are implemented by components, that are built using code.

Software System Something that delivers functional value to its users (human or other systems).

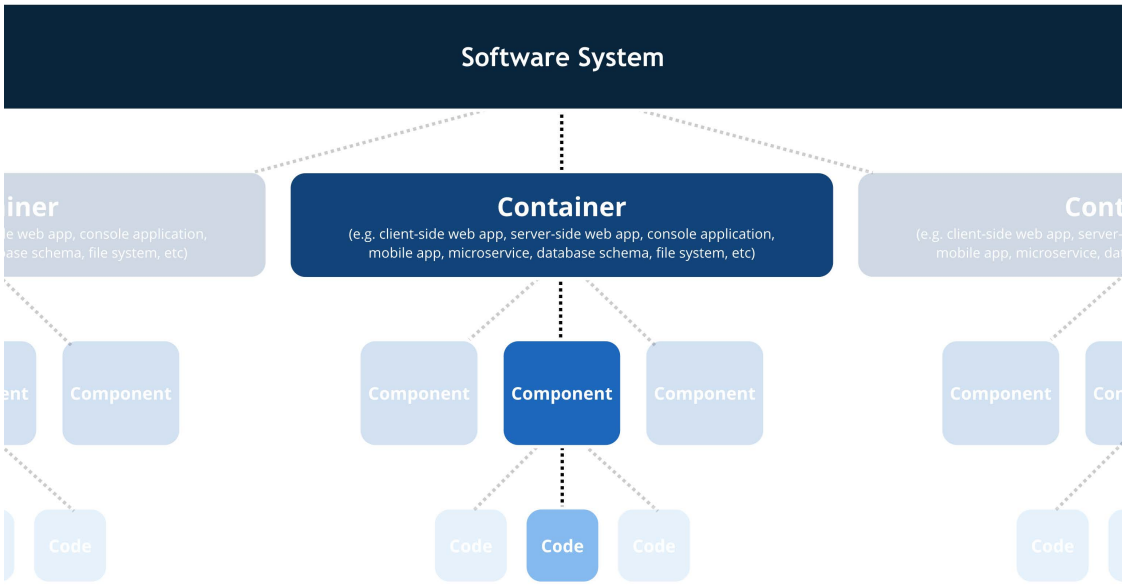
Containers Deployable 'block' of code or data that provides behaviour as part of the software system.

Components Encapsulate a group of related functionality, usually hidden behind a published interface.

Code Elements built from programming language constructs, e.g. classes, interfaces, functions, ...

The C4 model does not explicitly identify different architectural views, but views are implicit in the types of diagrams produced to describe an architecture.

[1] Though evidence suggests that an ability to read and reason about code is necessary to learn how to program well [1] [2].



A **software system** is made up of one or more **containers** (web applications, mobile apps, desktop applications, databases, file systems, etc), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (e.g. classes, interfaces, objects, functions, etc).

Figure 1: Levels within the C4 model (figure 2.1 from [4]).

The static *structural* view of software architecture is described through four levels of abstraction. Each level providing detail about parts of the previous level.

Context How the software system fits into the broader context around it.

Containers How the containers are connected to deliver system functionality.

Components How the components are structured to implement a container's behaviour.

Code How the code is structured to implement a component.

The *behavioural* view is described via a dynamic diagram that describes how containers and components interact to deliver system features.

The *infrastructure* view is described by a deployment diagram that describes how executable containers, software systems, and other infrastructure services are deployed on computing platforms to instantiate the system.

3 Sahara eCommerce Example

Sahara[2] eCommerce is an ambitious company who's prime business unit is an on-line store selling a wide range of products. Sahara provides both web and mobile applications to deliver the shopping experience to customers.

3.1 Architecturally Significant Requirements

Architecturally significant requirements (ASR) are functional or non-functional requirements, or constraints or principles, which influence the design of the system architecture. The structure of a software architecture

[2]Yes, that is intentionally a *dry* joke.

has to be designed to ensure that the ASRs can be delivered.

Not all requirements for a system will be architecturally significant, but those that are need to be identified. Once ASRs are identified, an architecture needs to be designed to deliver them. This may require some research, and experimentation with prototypes, to determine which options are appropriate. Tests should be designed to verify that the architecture is delivering the ASRs. Ideally, these should be part of an automated test suite. This may not be possible for all tests. Regardless, the ASR tests should be executed frequently during development to provide assurance that the system will deliver the ASRs.

Inevitably, some ASRs will be discovered later in the project. The existing architecture will need to be evaluated to determine if it can deliver the new ASRs. If it can, new tests need to be added to the ASR test suite to verify delivery of the new ASRs. If the architecture is no longer suitable due to the new ASRs, a major redesign needs to be done to create a new, more suitable, architecture.

The architecturally significant requirements for the Sahara eCommerce system are:

- Customers can start shopping on one device and continue on another device. (e.g. Add a product to their shopping cart while browsing on their computer when they are bored at school. Checkout their shopping cart from their mobile phone on their way home on the bus.)
- The system must be scalable. It must cater for peaks in demand (e.g. Cyber Monday and Singles Day). It must cater for an unknown distribution of customers accessing the on-line store through web or mobile applications.
- The system must be robust. The system must continue to operate if a server fails. It must be able to recover quickly from sub-system failures.
- The system must have high availability. Target availability is "four nines"[3] up time.

The following sections will describe the physical and software architecture for this system, and demonstrate how it delivers these ASRs.

3.2 System Context

The system context provides the 'big picture' perspective of the software system. It describes the key purpose of the system, who uses it, and with which other systems it interacts. The context diagram is usually a simple block diagram. The software system being designed typically sits in the centre of the diagram surrounded by users and other systems. The intent is to set the context for thinking about the software system's architecture. It can also be used to communicate basic structural ideas to non-technical stakeholders. Figure 2 is a context diagram for the Sahara eCommerce system. The overall eCommerce system is delivered through two interacting software systems, the on-line store and the data mining service.

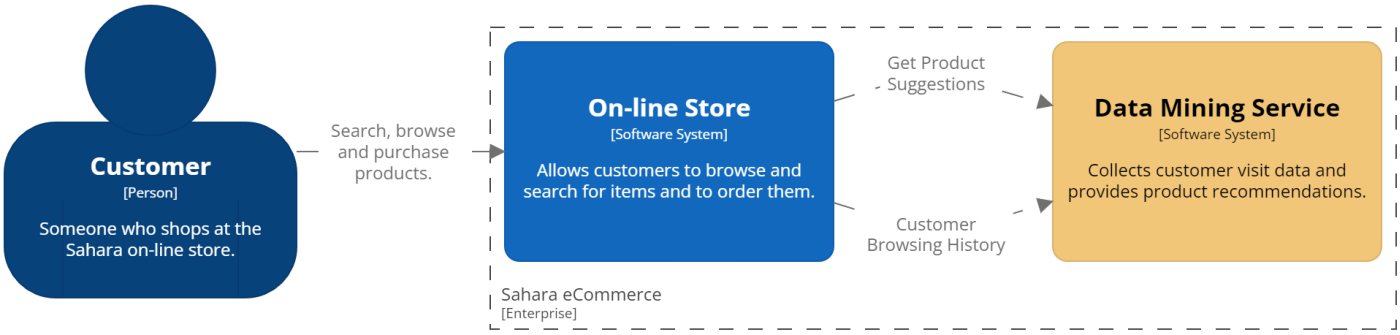


Figure 2: Context diagram for the Saraha eCommerce on-line store.

[3] A number of nines (e.g. four nines) is a common way to measure availability. It represents the percentage of time the system is "up". Four nines means the system is available 99.99% of the time, or it is not available for less than one hour per year.

Figure 3 is the key to help interpret the context diagram. A key is important for C4 diagrams, as they have a fairly informal syntax and specification.

Figure 3: Context diagram key.

The context diagram situates the on-line store software system in the environment in which it will be used. There are customers who shop at the on-line store, which is part of Sahara eCommerce's software ecosystem. The on-line store uses a data mining service that is also implemented by the company. The two key relationships between the on-line store and the data mining service are that the on-line store sends customer browsing data to the service, and that the on-line store requests the data mining service to recommend products for a customer.

In C4, arrows are used to indicate the main direction of the relationship, not the flow of data. So, in this example, the arrow points from the on-line store to the data mining service as it is the store that manages the communication.

3.3 Containers

Container diagrams provide an overview of the software architecture. They describe the main structure of the software system and the technologies selected to implement these aspects of the system. Containers are 'blocks' of code that can be independently deployed and executed. Examples of containers are web or mobile applications, databases, message buses, ... It is important to note that containers may be deployed on the same computing infrastructure or on different devices.

Container diagrams focus on the connections between containers and, to an extent, how they communicate. They do not explicitly show computing infrastructure. Decisions about how containers are connected and communicate have major implications for how the components and code will be designed and deployed. Figure 4 is a container diagram for the on-line store. For simplicity, containers managing load balancing and fail-over are not shown in this example.

To provide a link to the context diagram, a container diagram usually shows which containers communicate with which external elements. The text inside the square brackets in a container, and on a relationship, indicates the technology used to implement that container or relationship.

Customers access the on-line store through either web or mobile applications. The `Web Application` and `Interactive Web Pages` containers indicate that the web application's behaviour is delivered by two different specialised containers. The `Web Application` container indicates that it runs in a J2EE[4] server. This implements the presentation layer of the web application in Java. It handles browser requests from customers, using the HTTPS protocol over the Internet.

The `Interactive Web Pages` are JSF[5] pages and JavaScript code that implement more sophisticated user interactions on web pages. These run within the users' browsers.

The `Web Application` container uses RMI[6] to invoke functional behaviour in the `Application Backend`

[4]Jakarta Enterprise Edition [https://jakarta.ee/]

[5]Jakarta Faces [https://jakarta.ee/specifications/faces/]

[6]Remote Method Invocation [https://www.oracle.com/java/technologies/jpl1-remote-method-invocation.html

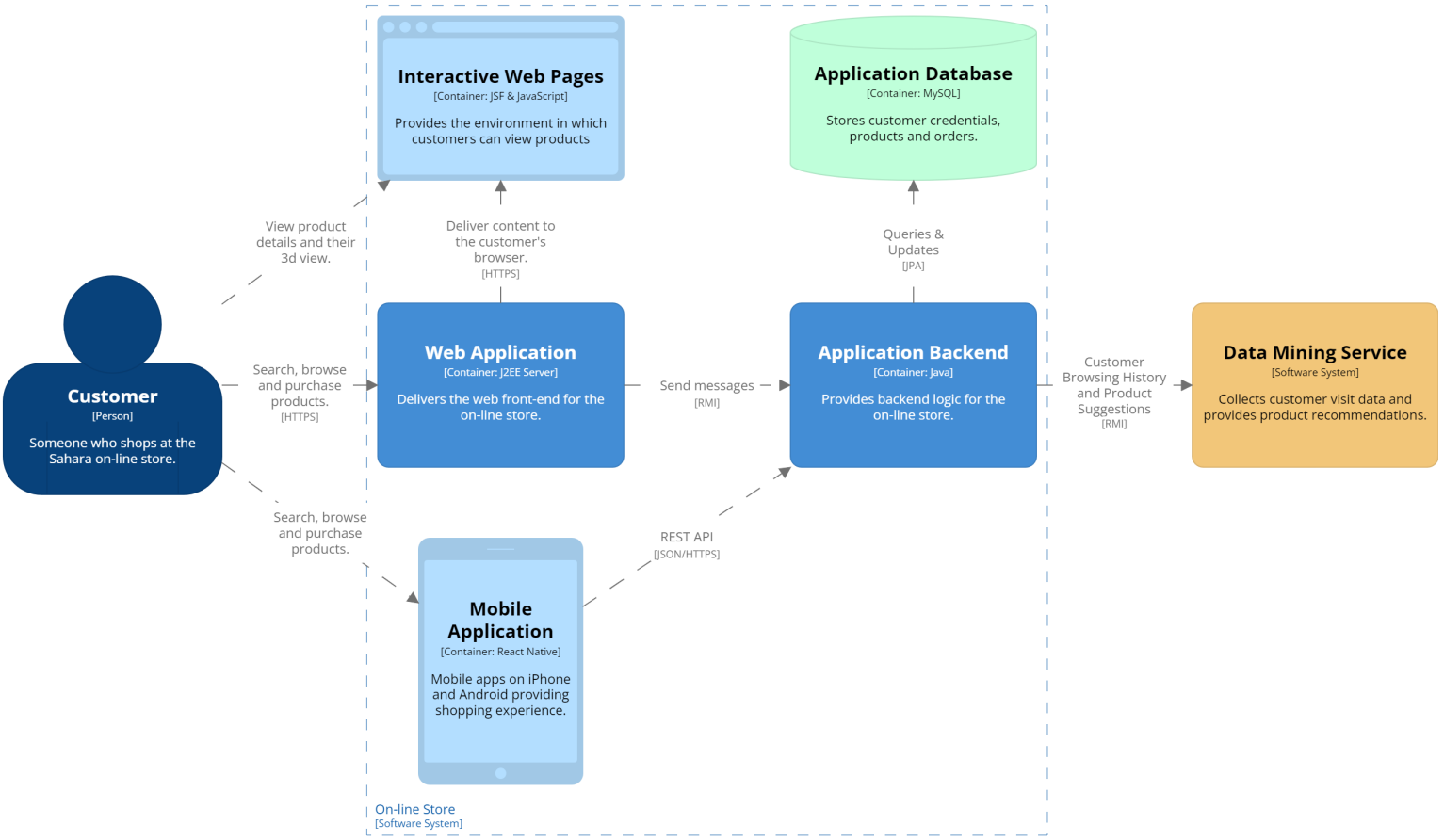


Figure 4: Container diagram for the on-line store software system.

container. It provides the shared logic of the on-line store. This supports implementing the functional requirement that a customer can start shopping on one device and continue on another.

The mobile application communicates with the application backend via a REST API[7]. The application backend uses JPA[8] to manage persistent data in the application's database. The application backend uses RMI to communicate with the data mining service, sending customer browsing and search history to be used for data mining and receiving product suggestions to present to customers.

While a container diagram does not explicitly show computing infrastructure, some of it can be implied by the types of containers in the diagram. Clearly, the mobile app and the code running in the interactive web pages have to be separate computing platforms to the rest of the on-line store's software system.

Colours and icons can be used to provide further information in the diagrams. The diagram key in figure 5 explains the purpose of each icon and colour.

The data mining service software system (figure 6) has three main containers.

`Data Mining Interface` provides the interface used to interact with the data mining service. It accepts data to store for future data mining. It returns suggestions based on requests from external systems, such as the on-line store.

`Data Mining Process` performs the data mining logic.

`Data Warehouse` stores the data and provides an SQL-based query system to manipulate the data. The data mining interface and process containers use JDBC[9] to work with the data in the data warehouse.

[7] https://www.ibm.com/topics/rest-apis

[8] Jakarta Persistence API [https://jakarta.ee/specifications/persistence/]

[9] Java DataBase Connectivity [https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html]

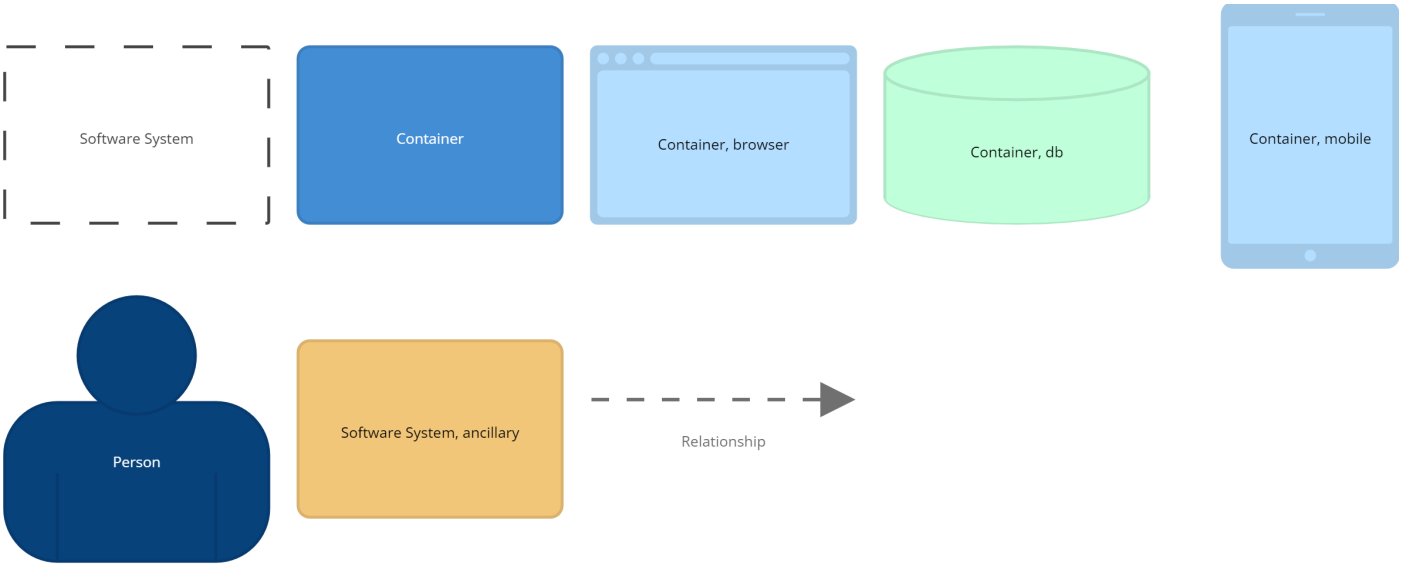


Figure 5: Container diagram key.

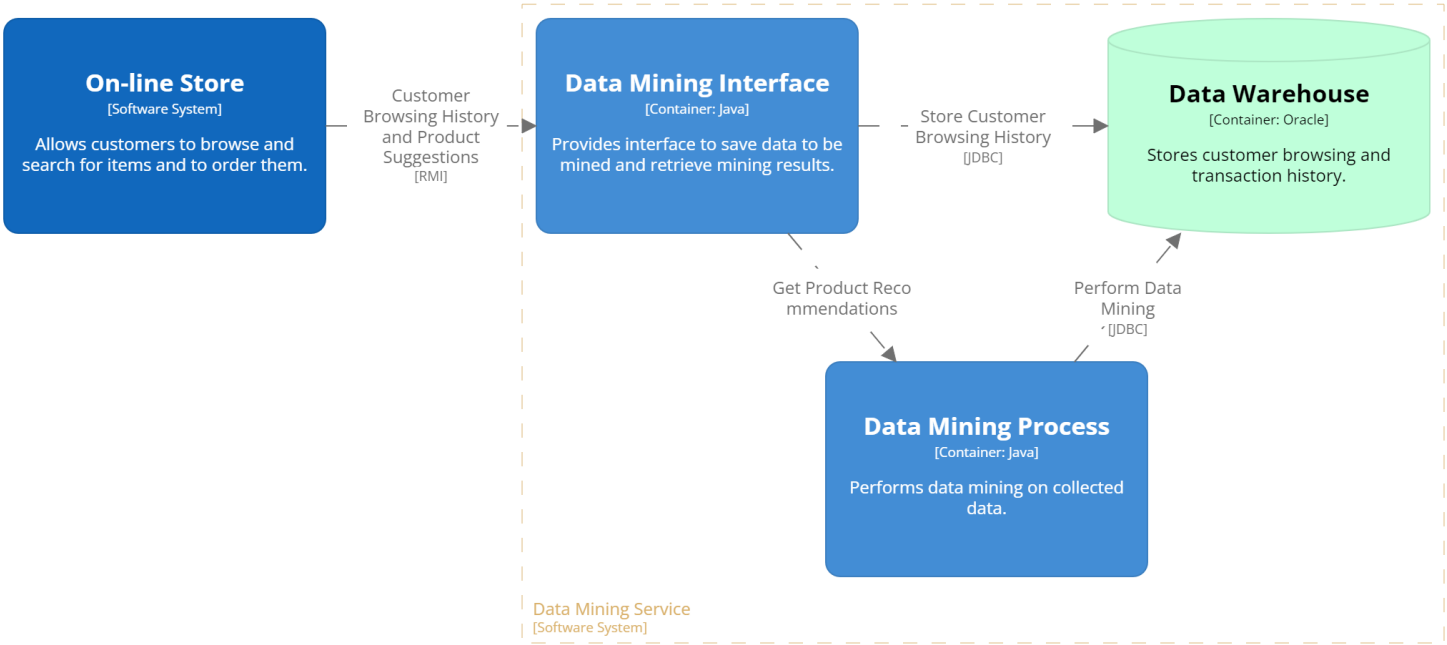


Figure 6: Container diagram for the data mining service software system.

3.4 Components

Component diagrams describe the major parts of containers and how they are connected. Components should describe their important responsibilities and the technology used to implement them (e.g. using React to implement a web frontend component). Like a container diagram, a component diagram can include elements from higher level diagrams to provide context of how the components interact with elements outside of the container.

In figure 7, the application backend is divided into five main components. The `Shopping Cart` component provides the backend logic of implementing a shopping cart. This includes storing the state of the cart in the application database so that it is available in a later shopping session. The `Products` component provides information about products in the application database. The `Customers` component retrieves customer credentials from the application database. The `Orders` component stores finalised orders in the application database.

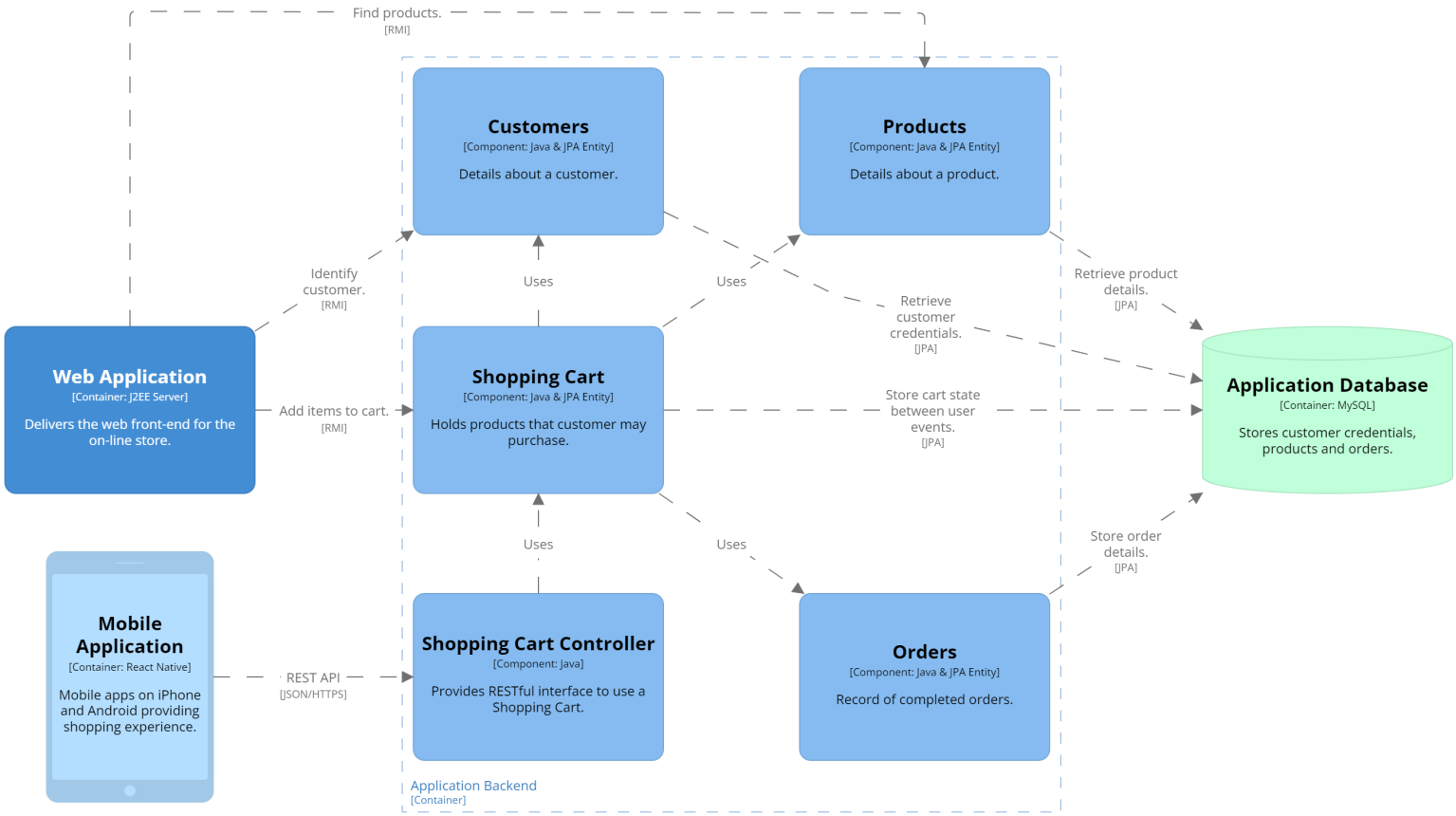


Figure 7: Component diagram for the application backend container.

The web application interacts with the `Customers`, `Shopping Cart` and `Products` components via RMI. They would provide interfaces for this interaction. `Shopping Cart` uses the `Products`, `Orders` and `Customers` components to deliver its behaviour. Together they deliver the logical behaviour of providing information about products, tracking what is in the shopping cart, and placing orders.

The application backend is implemented in Java and uses JPA Entities[10] to manage persistent data in the application database.

The mobile applications use a REST API to communicate with the application backend. An example of this is shown by the `Shopping Cart Controller` component which uses the `Shopping Cart` to deliver that behaviour to the mobile applications. Other controllers implementing REST APIs are not shown to reduce the complexity of the diagram.

Figure 8, shows the icons and colours used to represent different elements in the component diagrams.

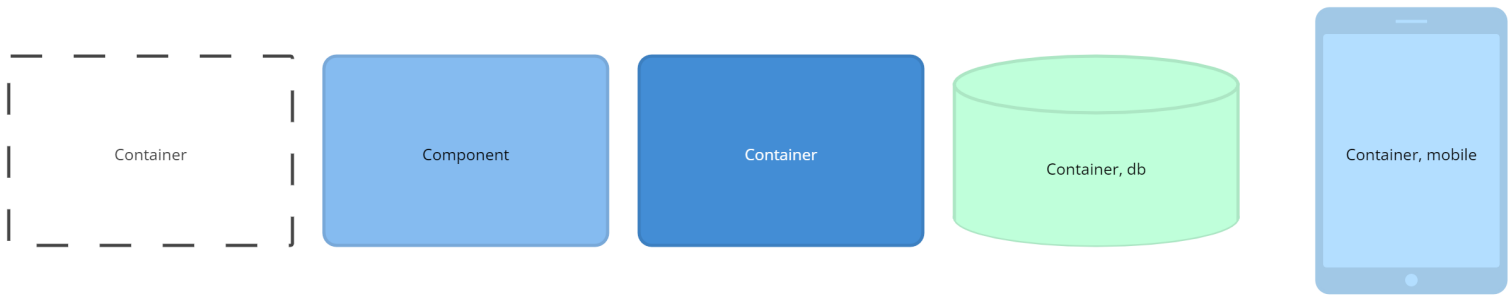


Figure 8: Component diagram key.

[10] https://jakarta.ee/specifications/persistence/3.2/jakarta-persistence-spec-3.2.html#entities

Figure 9 shows the components that provide the frontend behaviour, in the web application, of browsing for products, adding them to the shopping cart, and purchasing them.

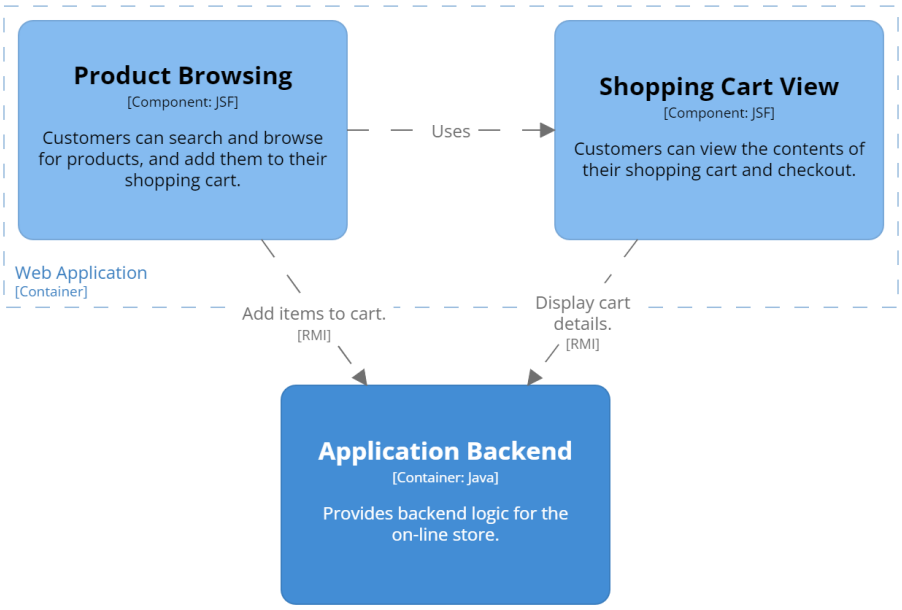


Figure 9: Component diagram for the web application container.

Figure 10, shows that the `Product Animator` component is downloaded from the web application to the customer's browser and that it is implemented in JavaScript. This component provides the code that allows customers to interact with 3D views of products.

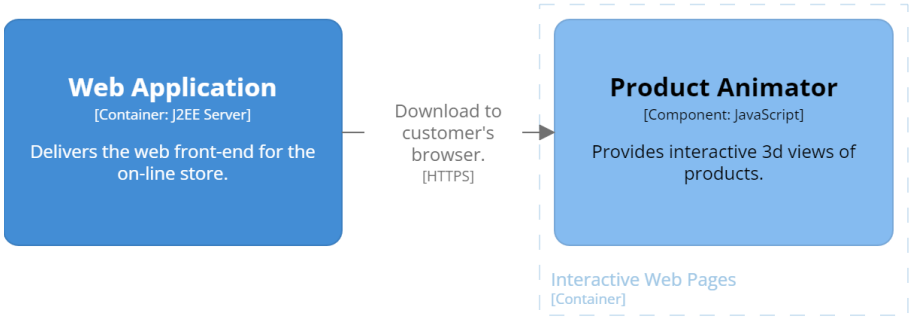


Figure 10: Component diagram for the interactive web pages container.

To keep the example manageable, only the components related to implementing the shopping cart are shown at this level of detail.

3.4.1 Component Diagram Detail

There may be some components that are important parts of the software design, but which may not necessarily be included in component diagrams. For example, a logging component is an important part of many software systems. But, due to the nature of a logging component, most other components will use it. Adding it to component diagrams will usually clutter the diagrams without adding much useful information. Usually it is better to add a note indicating which logging component is used in the system. If it is helpful to indicate which components use the logging component, it may be better to colour code these components or use an icon to represent that they use the logging component.

3.5 Code

Code-level diagrams describe the structure of the code that implements a component. The intent is to provide a visual representation of the important aspects of the code's structure.

C4 does not provide a code level diagram. It suggests using diagrams appropriate to your programming paradigm. Assuming the implementation is in an object-oriented language, a UML diagrams[11] class diagram would be an appropriate way to model the design of the code.

Figure 11 is a UML class diagram visualising the static structure of the classes that implement the `Shopping Cart` component. Usually only architecturally significant operations and attributes are shown. (e.g. Operations and attributes needed to understand relationships and behaviour.) Rarely do you need to provide all the detail that replicates the source code. (The source code could be considered a fifth level to the C4 model.) And for simplicity in this diagram, only the classes and interfaces related to adding items to a shopping cart and checking out are shown.

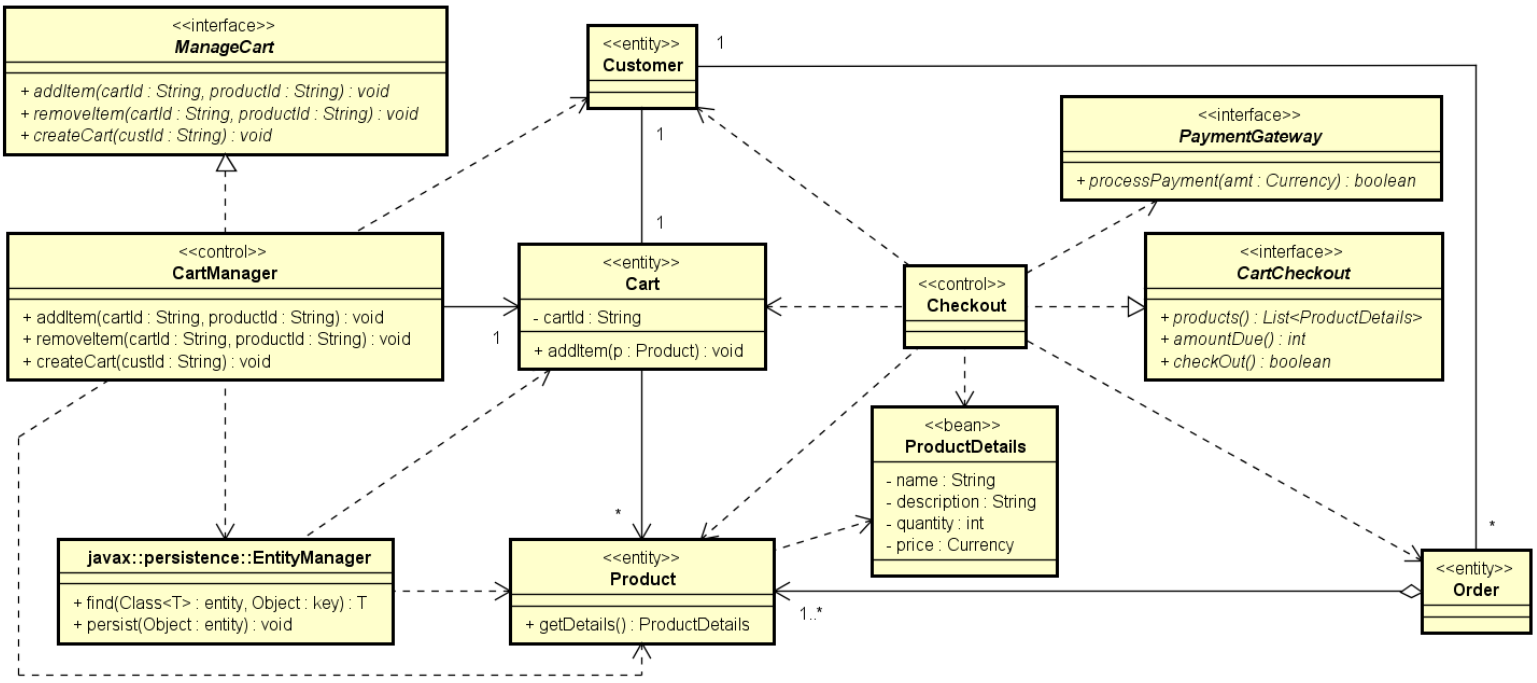


Figure 11: Example static structure for part of the shopping cart component.

The `CartManager` and `Checkout` control classes implement, respectively, the *`ManageCart`* and *`CartCheckout`* interfaces. These two classes implement the Façade design pattern and manage how adding products to a shopping cart and checking out are delivered by the classes in this package. Going back to the application backend and web application component diagrams (figures 7 and 9), when a customer, via their web browser, selects to add a product to their shopping cart, the `Product Browsing` component's logic uses the *`ManageCart`* interface's *`additem`* operation to send a message to the `Shopping Cart` component.

In the implementation of the `Shopping Cart` component, the `CartManager` class uses the `EntityManager` to load the product details and the customer's shopping cart from the application database. The `EntityManager` creates the `Product` and `Cart` entity objects, and `CartManager` adds the product to the cart. Once this is done the `EntityManager` is used to save the updated cart data into the database.

When a customer wants to checkout the products in their shopping cart, the `Shopping Cart View` component uses the *`CartCheckout`* interface's *`products`* operation to get a list of the product details to be displayed in the shopping cart. The `ProductDetails` class is a Java bean that is used to pass the data about each product to the `Shopping Cart View`. Once a customer decides to buy the products in their

[11] Unified Modeling Language [https://www.uml.org/]

shopping cart, the `Shopping Cart View` sends the `checkOut` message to the `Shopping Cart`. `Checkout` uses the *`PaymentGateway`* interface to process the payment.

3.5.1 Class Diagram Notation

Formally in UML, rectangles represent *classifiers*. A *class* is one type of classifier. In a class diagram, a rectangle represents a class, unless a keyword is used to indicate that it is a different type of classifier. Classifier rectangles have three compartments. The top compartment contains its name and optionally includes a keyword, stereotypes and properties for the classifier. The middle compartment contains *attributes*. The bottom compartment contains *operations*.

Solid lines represent *associations*, which may optionally have an arrow indicating the direction of the relationship. An association indicates a structural relationship between classes. Typically this means that the target of an association will be an implicit attribute of the class. The end of an association can use *multiplicity* to indicate the number of objects of the class that may take part in the relationship.

A diamond on the end of an association indicates an *aggregate* relationship. The diamond is on the end that is the aggregate, and the other end is the part. The diamond may be filled or not. A filled diamond represents *composition*. This indicates 'ownership', where the aggregate controls the lifespan of the part. A hollow diamond, as in the relationship between `Order` and `Product`, indicates *aggregation*. This is a weaker relationship than composition, as the aggregate does not control the lifespan of the part, but it still indicates a strong relationship between the classes.

A dashed line with an open arrowhead (e.g. from `CartManager` to `Product`) indicates that one classifier *depends* on (or uses) another. This is meant to indicate a transient relationship.

A dashed lines with a closed and hollow arrowhead (e.g. from `Checkout` to *`CartCheckout`*) indicates that the class is *realising* (or implementing) that interface.

Italicised names indicate an abstract classifier. Keywords are used to indicate the type of a classifier. In this example, the keyword «interface» indicates that the classifier is an interface. Stereotypes use the same notation as keywords. Three standard stereotypes for classes in UML are:

«entity» Represents a concept (*entity*) from the problem domain.

«control» Provides logical behaviour from the solution domain.

«boundary» Communicates with something outside of the system. (Not shown in diagram.)

An additional stereotype «bean» is used to indicate that the class is a Java bean.

3.6 Dynamic

Dynamic diagrams in C4 show how different parts of the model collaborate to deliver architecturally significant requirements. Normally dynamic diagrams are used to describe behaviour that is not clear from other diagrams and descriptions. This can include complex interactions between modules, complex concurrency, real-time constraints, or latency constraints.

For example, when Boeing was upgrading the combat control system of the F-111[12] for the Australian Airforce, they designed a software architecture that used CORBA[13] as middleware. The implementation of the middleware caused a fixed delay in sending messages between components. From an architectural design perspective, it was important to document this delay and enforce a maximum delay on the time taken to complete any process. This type of constraint can be documented with a dynamic diagram and supporting documentation.

Figure 12 provides an overview of how the `Product Browsing` component in the `Web Application` container collaborates with the `Shopping Cart` component in the `Application Backend` container to

[12] https://www.youtube.com/watch?v=xUcpZJE050s

[13] Common Object Request Broker Architecture [https://www.ibm.com/docs/en/integration-bus/9.0.0?topic=corba-common-object-request-broker-architecture]

deliver the behaviour of a customer adding a product to their shopping cart. It also shows the communication between the `Shopping Cart` component and the application database.

Figure 12: Dynamic diagram for adding a product to the customer's shopping cart.

A dynamic diagram should provide enough information to clarify any requirements or constraints on the design, but should not be so restrictive as to limit valid implementation choices.

UML provides two types of interaction diagrams that are similar to dynamic diagrams. A communication diagram is very similar to a dynamic diagram. A sequence diagram focusses on the time or ordered sequence of events that occur in a scenario. In documenting a software architecture, there may be some types of constraints that are more clearly expressed through a sequence diagram.

3.6.1 Detailed Behaviour

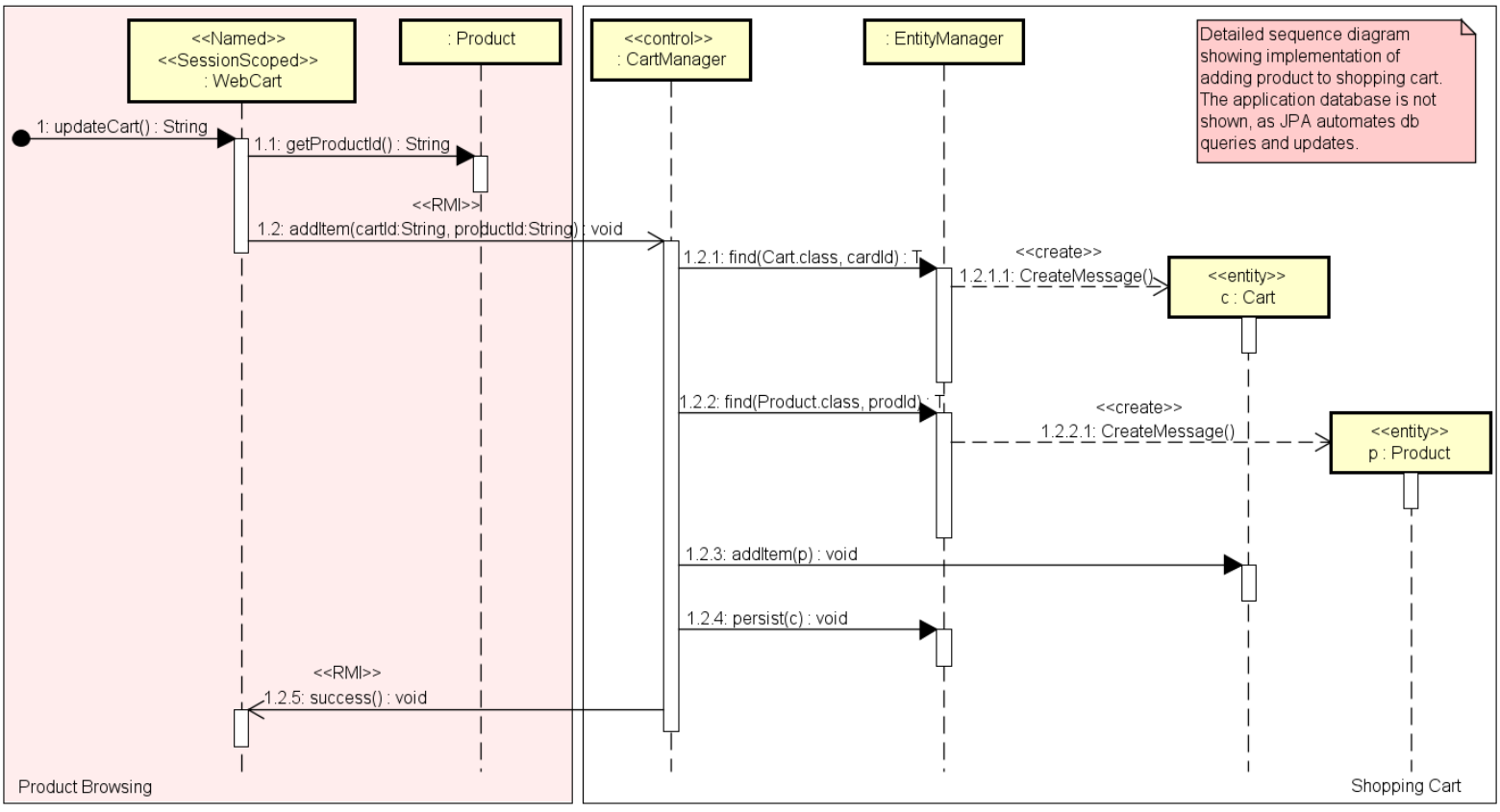


Figure 13: Example detailed sequence diagram showing the implementation of customer adding a product to their shopping cart.

Figure 13 is a detailed sequence showing how the class model in figure 11 implements the behaviour of a customer adding a product to their shopping cart. This is a more detailed, or code-level, view of the

scenario of adding a product to a customer's shopping cart shown in figure 12. You would only provide detailed behavioural diagrams to describe architecturally important details of the detailed design.

The scenario starts with the JSF session-scoped bean `WebCart` receiving the `updateCart` message from the browser. `WebCart` retrieves the product id and uses the `ManageCart` interface to send it, along with the cart id, to the `Shopping Cart` component in the application backend. The *`ManageCart`* interface is implemented by the `CartManager` class in the `Shopping Cart` component.

The `CartManager` uses the JPA `EntityManager` to retrieve the cart and product entities from the application database. Once the product is added to the cart, the `EntityManager` saves the updated cart details to the database. Upon successfully saving the updated cart, the `CartManager` notifies the `WebCart` object in the `Product Browsing` component in the web application.

3.6.2 Sequence Diagram Notation

Sequence diagrams are read from the top down. The top of the diagram represents the start of the scenario, and execution time progresses down the diagram. The bottom of the diagram is the end of the scenario. Duration constraints can be placed between messages indicating information like the maximum allowed time between the start and end of a message.

Rectangles with dashed lines descending from them are *lifelines*. They represent an instance of a participant in the scenario being described. The name at the top of a lifeline describes the participant. In figure 13, these are objects of classes from the detailed design. The class diagram in figure 11 shows the classes for the detailed design of the `Shopping Cart` component used in this scenario.

The horizontal lines are *messages* sent between participants. Messages use hierarchical numbers to indicate both nesting and sequence of messages. Message 1.1 is sent by message 1. Message 1.1.1 comes before message 1.1.2. Message 1 in figure 13 is a *found message*, meaning that the sender of the message is not shown in the diagram.

A closed arrowhead on a message (e.g. message 1.1) indicates that it is a synchronous message. An open arrowhead on a message (e.g. message 1.2) indicates that it is an asynchronous message. In figure 13, stereotypes have been placed on messages between containers (i.e. messages 1.2 and 1.2.5) to indicate the protocol used to send the message.

The vertical rectangles sitting on top of lifelines are *execution specifications*. They indicate when an instance is executing logic. For example, after the asynchronous message 1.2 is sent to the `CartManager` object, message 1 finishes executing. When the synchronous message 1.2.1 is sent to the `EntityManager`, message 1.2 is still active as it is waiting for message 1.2.1 to finish before message 1.2 can continue to the next part of the logic.

The «create» stereotype indicates when an instance is created. When an instance is created, its lifeline starts at the level of the sequence diagram that indicates the point in time when it is created. When an instance is destroyed, its lifeline finishes, with a large `X`. Lifelines that are at the top of the diagram indicate instances that existed before the start of the scenario. Lifelines that reach the bottom of the diagram indicate instances that still exist after the end of the scenario.

System boundary boxes in figure 13 indicate the components being implemented by the objects. The `Product Browsing` component is shaded in pink. The `Shopping Cart` component is white.

3.7 Deployment

While not one of the "four C's", deployment diagrams are important for most systems. They describe the physical architecture or infrastructure on which the system will be deployed. It shows which containers will run on which computing platforms (*deployment nodes*). Deployment nodes can be nested, as shown in figure 14. They may also contain infrastructure nodes or software system instances[14].

[14] https://docs.structurizr.com/dsl/language#deploymentnode

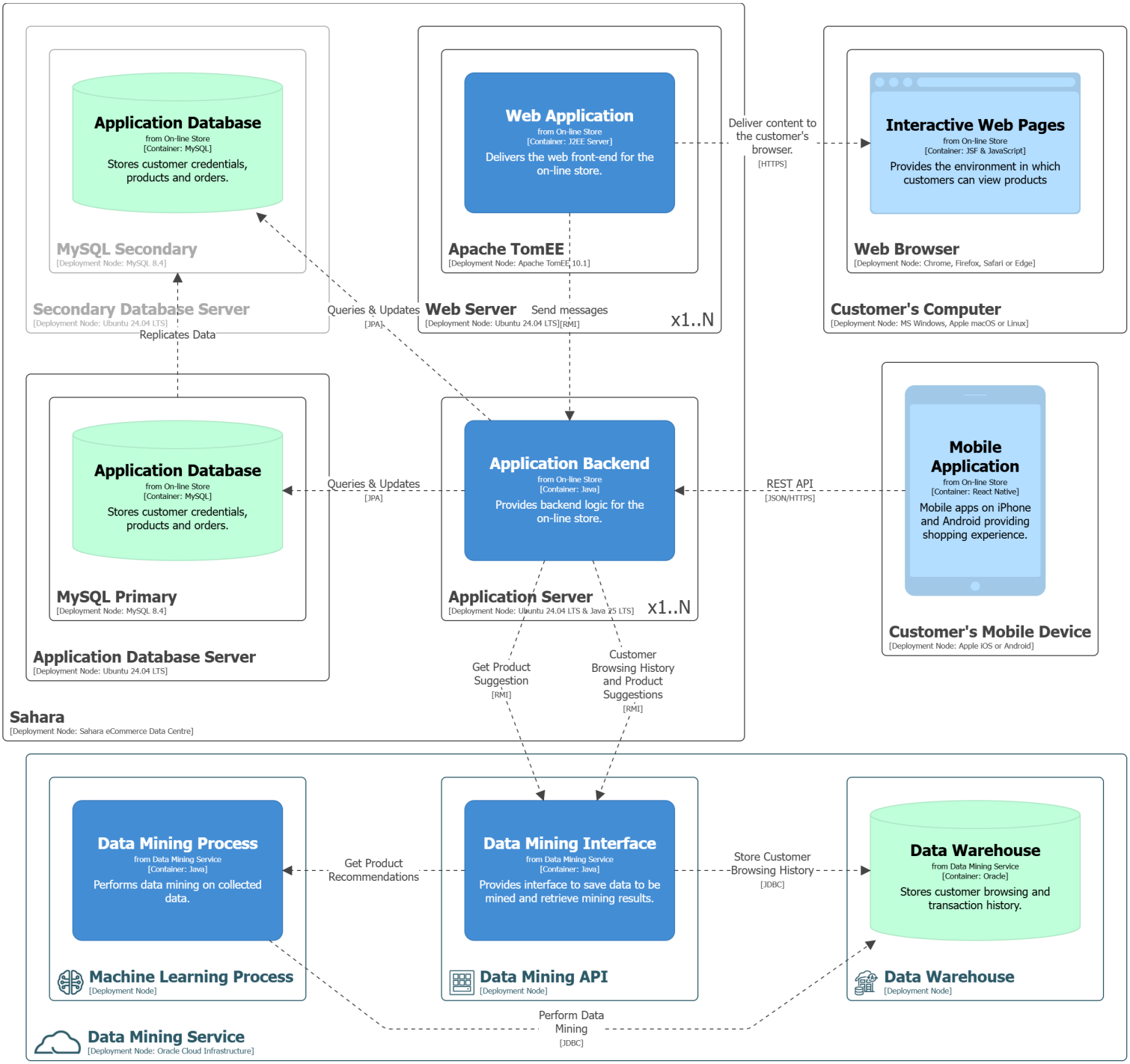


Figure 14: Deployment diagram for the Sahara eCommerce System.

Figure 14 is an example C4 deployment diagram for the Sahara eCommerce system. It shows that the on-line store software system runs in Sahara's data centre. The data mining service runs on Oracle's cloud infrastructure. This approach of a system that uses cloud services for some of its implementation is called a *hybrid cloud* application. There are also the apps running on mobile devices and the code running in the customer's browser.

A software environment is embedded in the hardware environment on which it runs. The web application runs in an Apache TomEE[15] J2EE server, which is running on a Ubuntu server. The "`x1..N`" inside the web and application server deployment nodes indicate that there will be at least one of each of these servers to share the load. The application backend runs on Ubuntu servers, providing the core business

[15] https://tomee.apache.org/

logic shared by the web and mobile applications.

The application database runs in MySQL on its own Ubuntu server. The application database is replicated on another server, allowing for failover. The application backend can continue to operate if the primary application database fails.

The application backend communicates with the data mining service through an API published by the data mining interface running in a virtual machine on Oracle's cloud infrastructure. The data mining service uses Oracle's machine learning services to perform the data mining. Oracle's cloud-based data warehouse infrastructure is used to hold all the data.

Figure 15 is the key describing the icons and colours used in the deployment diagram.

Figure 15: Deployment diagram key.

3.8 Delivering Architecturally Significant Requirements

In section 3.1, four ASRs were identified for the Sahara eCommerce system. These were the abililty to continue shopping on different devices, scalability, robustness and availability.

Implementing shared logic on an application server, as shown in figure 14, enables the web and mobile applications to share common logic and state. This delivers the functionality of allowing a customer to start shopping on one device and to continue on another device. It also minimises duplication of logic, as it is shared by the frontend applications.

Using separate computing infrastructure for the web server, application server, application database, and data mining service, as shown in figure 14, provides more options to deliver scalability, robustness and availability. For scalability and performance, each computing environment can be optimised for the services it delivers. It also means that new infrastructure can be deployed to target specific bottlenecks. The system follows the *stateless architecture* pattern[16]. The web and mobile applications do not store any application state (e.g. products currently stored in the shopping cart). Every time the customer performs a transaction (e.g. viewing product details or adding a product to their shopping cart), the web or mobile

[16] https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless

application sends a message to the application server to perform the action. The application server then saves or loads data to or from the application database.

This means that web and mobile applications can send messages to a different application server for each request. This facilitates scalability, robustness and availability. A new application server can be started to cater for increasing system load, or to replace a failed server. The stateless nature of the application server logic means that no data will be lost if a server fails, or if a frontend application accesses a different application server in the middle of a customer's shopping experience.

Having multiple application servers, web servers, and multiple application databases, means that if one server fails its load can be picked up by other servers. Automating the process of starting or restarting servers improves robustness and availability. Running the data mining service on Oracle's cloud infrastructure means that we can rely on their management of services to guarantee our required level of scalability and availability.

One challenge of a stateless architecture is providing a replicated database that contains up-to-date copies of the system's state. We will look at this issue later in the course.

By designing the architecture as a set of components running on different servers, it is also easier to migrate the application to cloud-based infrastructure. Figure 14 does not constrain the system to run on physical hardware hosted by Sahara eCommerce. Any of the nodes could be provisioned by a service offered by a cloud provider.

4 Tools

We will use C4 as our standard notation in this course, supplemented with UML diagrams when their level of detail is appropriate. C4 is popular because it has a basic structure, but the rules are intentionally loose to make it easy to adopt. You should use tools to aid the creation of your diagrams and documentation.

The important thing is that you should use a modelling tool, not a drawing tool. A few drawing tools provide C4 templates. The issue with drawing tools is that they do not know what the elements of the diagram mean. If a container name is changed in a drawing tool, you will need to manually change it wherever it is referenced in other diagrams. A modelling tool will track the information that describes the model, so that a change to a model element in one place, will be replicated wherever that element appears in other diagrams.

There are a few tools that support C4. Some to consider are Structurizr[17], C4-PlantUML[18], Archi[19], IcePanel[20], or Gaphor[21].

Structurizr was developed by Simon Brown as a tool to support generating C4 diagrams from textual descriptions. Structurizr is an open source tool[22]. You can use a domain specific language[23] to describe a C4 model, or you can embed the details in Java[24] code.

C4-PlantUML extends the UML modelling tool PlantUML to support C4.

Archi is an open source visual modelling tool supporting C4[25] and ArchiMate models.

IcePanel is a cloud-based visual modelling tool supporting C4. There is a limited free license for the tool.

Gaphor is an open source visual modelling tool supporting UML and C4.

[17] https://https://www.patreon.com/posts/146923136/
[18] https://github.com/plantuml-stdlib/C4-PlantUML
[19] https://www.archimatetool.com/
[20] https://icepanel.io/
[21] https://gaphor.org/
[22] https://github.com/structurizr/
[23] https://docs.structurizr.com/dsl/language
[24] https://docs.structurizr.com/java
[25] https://www.archimatetool.com/blog/2020/04/18/c4-model-architecture-viewpoint-and-archi-4-7/

4.1 Textual vs. Visual Modelling

The tools described above include both graphical and textual modelling tools. Graphical tools, such as Archi and Gaphor, allow you to create models by drawing them. This approach is often preferred by visually oriented learners[26]. Text-based tools, such as C4-PlantUML and Structurizr, allow you to create models by providing a textual description of the model. This approach is often preferred by read/write oriented learners[27].

Despite preferences, there are situations where there are advantages of using a text-based modelling tool. Being text, the model can be stored and versioned in a version control system (e.g. git). For team projects, it is much easier for everyone to edit the model and ensure that you do not destroy other team members' work. It is also possible to build a tool pipeline that will generate diagrams and embed them into the project documentation.

Text-based modelling tools, such as Structurizr or C4-PlantUML, use a domain specific language[28] (DSL) to describe the model. These tools require that you learn the syntax and semantics of the DSL. The following sources of information will help you learn the Structurizr DSL:

- language reference manual[29],
- language examples[30],
- on-line editable examples[31], and
- off-line tool[32].

4.2 Example Diagrams

You may find the Sahara eCommerce C4 model useful as an example of a number of features of the Structurizr DSL. You are able to download the C4 model of the Sahara eCommerce example, from the course website. The C4 model[33] was created using the Structurizr DSL.

5 *Software Architecture in Practice* Views

The seminal architecture book, *Software Architecture in Practice* [5], categorises architectural views into three groups. These three groups each answer different questions about the architecture, specifically:

Module Views How implementation components of a system are structured and depended upon.

Component-and-connector Views How individual components communicate with each other.

Allocation Views How the components are allocated to personnel, file stores, hardware, etc.

5.1 Module Views

Module views are composed of modules, which are static units of functionality such as classes, functions, packages, or whole programs. The defining characteristic of a module is that it represents software responsible for some well-defined functionality. For example, a class which converts JSON to XML would be considered a module, as would a function which performs the same task.

[26] https://vark-learn.com/strategies/visual-strategies/
[27] https://vark-learn.com/strategies/readwrite-strategies/
[28] https://opensource.com/article/20/2/domain-specific-languages
[29] https://docs.structurizr.com/dsl/language
[30] https://docs.structurizr.com/dsl/cookbook/
[31] https://playground.structurizr.com/
[32] https://github.com/structurizr/
[33] https://csse6400.uqcloud.net/resources/sahara-c4.zip

The primary function of module views is to communicate the dependencies of a module. Rarely does software work completely in isolation, often it is constructed with implicit or explicit dependencies. A module which converts JSON to XML might depend upon a module which parses JSON and a module which can format XML. Module views make these dependencies explicit.

Module views focus on the developer's perspective of how the software is implemented, rather than how it manifests itself when deployed in a computing environment.

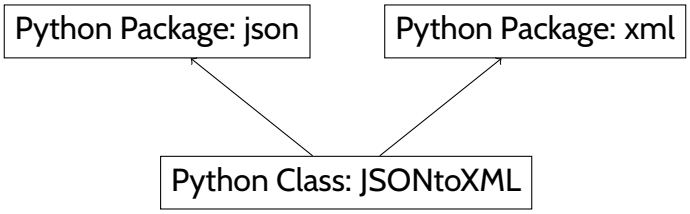
```
import json
import xml

class JSONtoXML:
    def load(self, json_file):
        with open(json_file) as f:
            data = json.load(f)
        self.data = self.convert(data)

    def export(self, xml_file):
        xml.write(xml_file, data)

    def convert(self, data: JSON) -> XML:
        ...
```

(a) Pseudo-code to convert JSON to XML



(b) An example of a module view which illustrates the dependencies of the `JSONtoXML` class

Figure 16: A simple module view of a JSON to XML program.

5.2 Component-and-Connector Views

Component-and-connector views focus on the structures that deliver the runtime, or dynamic behaviour of a system. Components are units which perform some computation or operation at runtime. These components could overlap with the modules of a module view but are often at a higher level of abstraction. The focus of component-and-connector views is how these components communicate at runtime. Runtime communication is the connector of components. For example, a service which registers users to a website might have new registrations communicated via a REST request. The service may then communicate the new user information to a database via SQL queries.

When we look at software architecture, component-and-connector views are the most commonly used views. They are common because they contain runtime information which is not easily automatically extracted. Module views can be generated after the fact, i.e. it is easy enough for a project to generate a UML class diagram. (Simple tools will create an unreadably complex class diagram. Tagging important information in the source code, or manually removing small details is required to end up with readable diagrams.) Component-and-connector views are often maintained manually by architects and developers.

5.3 Allocation Views

According to Bass et al, allocation views map the software's structures to the system's non-software structures [5]. They include concepts such as who is developing which software elements, where are source files stored for different activities such as development and testing, and where are software elements executed. The first two points are important for project management and build management. The last point of how the software is executed on different processing nodes is important for architectural design. This is sometimes called the *deployment structure* or the software system's *physical architecture*.

Understanding the physical architecture (simplistically the hardware[34] on which the software is executed) is important when designing the software's *logical architecture*. Component-and-connector views describe the software's logical architecture. This design of the logical architecture must contain components that can be allocated appropriately to processing nodes, and these nodes must have communication links that enable the components to interact.

6 4+1 Views

Philippe Kruchten was one of the earliest to advocate the idea of using views to design and document software architectures. In "4+1 View Model of Software Architecture" [6] he describes five different views. These are logical, process, development, physical, and scenario views, which are summarised below.

Logical How functionality is implemented, using class and state diagrams.

Process Runtime behaviour, including concurrency, distribution, performance and scalability. Sequence, communication and activity diagrams are used to describe this view.

Development The software structure from a developer's perspective, using package and component diagrams. This is also known as the implementation view.

Physical The hardware environment and deployment of software components. This is also known as the deployment view.

Scenario The key usage scenarios that demonstrate how the architecture delivers the functional requirements. This is the '+1' view as it is used to validate the software architecture. This is also known as the use case view, as high-level use case diagrams are used to outline the key use cases and actors. A high-level use case diagram provides contextual information, similar to the intent of the C4 context diagram. A use case diagram allows more information to be conveyed and if it is supported with activity diagrams, it provides a significant amount of information that can be used to validate the architecture.

The 4+1 View Model came from Philippe's experience leading development of the Canadian air traffic control management system. The system provides an integrated air traffic control system for the entire Canadian airspace. This airspace is about double the size of the Australian airspace and borders the two busiest airspaces in the world. The project's architecture was designed by a team of three people led by Philippe. Development was done by a team of about 2500 developers from two large consulting companies. The project was delivered on-time and on-budget, with three incremental releases in less than three years[35]. This project also developed many of the ideas that led to the Rational Unified Process [8].

[34] Whether it is virtualised or physical hardware

[35] Contrast this to the United States Federal Aviation Administration's Advanced Automation System project from a similar era. The original estimate was $2.5 billion and fourteen years to implement. The project was effectively cancelled after twelve years. By then the estimate had almost tripled and the project was at least a decade behind schedule [7].

7 Conclusion

Architectural views help developers understand different dimensions and details of a complex software architecture. They are useful both during design and as documentation. During design, views help you to focus on a particular aspect of the software architecture and ensure that it will allow the system to deliver all of its requirements. As documentation, views help developers to understand how different aspects of the architecture are intended to behave.

We have intentionally looked at a few different approaches to helping you describe a software architecture. You should be conversant with multiple different approaches. The hallmark of a professional is to know when to select a particular approach.

If you compare the "4+1 View Model" [6] with the views described in *Software Architecture in Practice* (SAP) [5], you will see that there are obvious similarities but also some differences. The logical, development and process views from the 4+1 view model map closely to the module and component-and-connector (C&C) views from SAP. The physical view corresponds to the allocation view. The 4+1 view model also includes the scenario view, which does not correspond directly to the SAP views.

The scenario view is used to demonstrate how the architecture described in the other views delivers the core functional requirements. It is used while designing a software architecture to validate that it is suitable for the system.

Kruchten intentionally separated the process view from the logical and development views, rather than bundling them together like the C&C view in SAP. This is because for some systems the dynamic details, which are described by the process view, can be complex and important. Dealing with issues such as complex concurrency, real-time interactions or latency, can often be more easily considered by having a separate view for them.

Kruchten's experience with Canada's integrated, nation-wide, air traffic control system was such a case. Data from radar systems and aircraft transponders have to be processed and reported to air traffic controllers in near real-time. With thousands of input sources and hundreds of controller stations, understanding concurrency issues is critical. Tracking aircraft from one control space to another means that communication latency is important. Each control space has its own hardware and is separated from neighbouring spaces by hundreds or thousands of kilometres.

The C4 model does not explicitly include the concept of views. Like SAP, it emphasises the structure of the software architecture, adding a hierarchical lens in its structural view. These diagrams map to the 4+1 logical and development views and the SAP module and C&C views. Its behavioural view maps to aspects of the 4+1 process view. Its infrastructure view maps to the 4+1 physical view and the SAP allocation view.

As a software architect you need to choose which views provide meaningful information about your software system. The graphical notation used to describe a view is only one part of the view (though an important part). Ensure you provide enough supporting information so others will know how to work with your architecture and why you made the choices that you did.

References

[1] R. Lister, C. Fidge, and D. Teague, "Further evidence of a relationship between explaining, tracing and writing skills in introductory programming," in *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, (New York, NY, USA), p. 161–165, Association for Computing Machinery, 2009.

[2] R. Lister, "Concrete and other neo-piagetian forms of reasoning in the novice programmer," in *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, (AUS), p. 9–18, Australian Computer Society, Inc., 2011.

[3] S. Brown, *The C4 Model: Visualising Software Architecture*. O'Reilly Media, Inc., July 2026. https://www.oreilly.com/library/view/the-c4-model/9798341660113/.

[4] S. Brown, *The C4 Model for Visualising Software Architecture*. Leanpub, November 2025. https://leanpub.com/visualising-software-architecture.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 4th ed., August 2021.

[6] P. Kruchten, "Architectural blueprints — the '4+1' view model of software architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995. https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

[7] T. Hilburn, A. Squires, H. Davidz, and R. Turner, "Federal aviation administration (faa) advanced automation system (aas)." https://www.sebokwiki.org/wiki/Federal_Aviation_Administration_(FAA)_Advanced_Automation_System_(AAS), October 2021. Example from the *Guide to the Systems Engineering Body of Knowledge* https://www.sebokwiki.org/w/index.php?title=Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK).

[8] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2004.