# Software Engineering

## Software Reuse:

Software reuse of existing artifacts to build new software components improves speed and quality in feature and system delivery while encouraging effective use of your resources.

Software reuse is the use of existing artifacts to build new software components. Software reuse also ensures that teams aren't reinventing the wheel when it comes to solving repetitive problems. Software libraries, design patterns, and frameworks are all great ways to achieve this. By taking the time to code with reuse principles in mind as well as reusing existing artifacts, you'll save time in the long term. In short, reuse is how technologists can avoid the proverbial reinvention of the wheel.

### The importance of Reuse

There are several benefits to reusing existing artifacts like code or design patterns. With a robust reuse strategy, tech organizations are likely to find:

- **Elimination of duplicative work:** Duplicate efforts among many teams that net the same result means wasted effort, cycles, and less time spent on potentially higher value work. The potential productivity gains on higher value business constructs through the avoidance of duplicate efforts may be the most significant result for an organization.
- **Consistency of system construction and behavior:**_ Creating an ecosystem in which approved patterns, frameworks, and libraries are reused across many disparate systems provides for consistent, highly predictable system behavior. Additionally, as problems arise and are solved within these ecosystems, they can be more easily shared and implemented across organizations by leveraging those same components.
- **Shortened software development time:** Developers save time when an application they are working on requires a piece of code that already exists. Time spent reinventing wheels is always more effectively used innovating.
- **Improved speed and quality in feature and system delivery:** Consistent architecture and design patterns, coupled with a finite set of already codified components, can be used to quickly construct large portions of systems and features with a baseline range of acceptable security, performance, and reliability characteristics. This frees cycles and allows engineers to focus on higher value deliverables and solve ever more challenging problems.

## How to find and contribute to reusable assets

We have an internal community that promotes reuse and collaborative development across our company.

We embrace the open-source principles of:

- **Contribution:** Contributing back to the greater community empowers us all.
- **Collaboration:** Communities creating solutions together improves agility and accelerates innovation.
- **Consumption:** The preeminent term in *reuse* is *use*!

These principles encourage a culture of open sharing and reuse. We share our reuse assets through a dedicated Reuse Catalog, which the community regularly maintains and updates.

## Developing a Reuse mindset

Implementing a reuse mindset on your own team requires forethought. Before you build any new components, ask yourself the following questions:

- Can I find any reusable components others have already developed?
- If I need to develop a new component, what does reuse mean for this construct?
- How can others within or outside my area benefit from this component?

In order to develop and share reusable components, follow these guidelines:

- **Specify ownership and support:** Is this component supported by a fully supported product team? Is the asset provided 'As-is?' Let users know they are free to use, modify, augment, enhance, and keep the content updated.
- **Low coupling with the rest of the system:** A component should be independent of the environment where it is being used.
- **A well-defined interface:** This allows for external services to know how to interact with the component.
- **Single responsibility rule:** Reusable components should follow the single responsibility rule and fulfill a well-defined purpose rather than trying to satisfy multiple needs.
- **Ensure the solution is well documented:** Reusable components need to be sufficiently well documented so that the target audience can easily consume them with little to no assistance. Documentation should include the problem statement and the solution. You should clearly define input, outputs, and usage of your component with examples. Point to reference implementations whenever possible to facilitate their implementation and foster collaboration between teams.
- **Ensure core engineering principles are addressed:** As you deliver reusable components, ensure that core engineering principles such as security and observability are addressed so that your audience doesn't have to figure it out.

- **Respect original sources:** Provide links and references to the original source for additional learning and credit to the original sources.

## Component based software engineering:

Component-based software engineering (CBSE) is an approach to software development emerged in the 1990's that relies on the reuse of entities called 'software components'. It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific. Components are more abstract than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

**CBSE essentials**

- **Independent components** specified by their interfaces.
- **Component standards** to facilitate component integration.
- **Middleware** that provides support for component inter-operability.
- **A development process** that is geared to reuse.

Apart from the benefits of reuse, CBSE is based on sound **software engineering design principles**:

- Components are independent so do not interfere with each other;
- Component implementations are hidden;
- Communication is through well-defined interfaces;
- One components can be replaced by another if its interface is maintained;
- Component infrastructures offer a range of standard services.

**Standards** need to be established so that components can communicate with each other and inter-operate. Unfortunately, several competing component standards were established: Sun's Enterprise Java Beans, Microsoft's COM and .NET, CORBA's CCM. In practice, these multiple standards have hindered the uptake of CBSE. It is impossible for components developed using different approaches to work together.

Solution for interoperating standards: **component as a service**. An executable service is a type of independent component. It has a 'provides' interface but not a 'requires' interface. From the outset, services have been based around standards so there are no problems in communicating between services offered by different vendors. System performance may be slower with services but this approach is replacing CBSE in many systems.

### Components and component models

Components provide a service without regard to where the component is executing or its programming language. A component is an **independent executable entity** that can be made up of one or more executable objects. The component interface is published and all interactions are through the published interface.

**Component characteristics:**

**Composable**

For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.

**Deployable**

To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of that component. Rather, it is deployed by the service provider.

**Documented**

Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

**Independent**

A component should be independent--it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a "requires" interface specification.

**Standardized**

Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

The component is an independent, executable entity. It does not have to be compiled before it is used with other components. The services offered by a component are made available through an interface and all component interactions take place through that interface. The component interface is expressed in terms of parameterized operations and its internal state is never exposed.

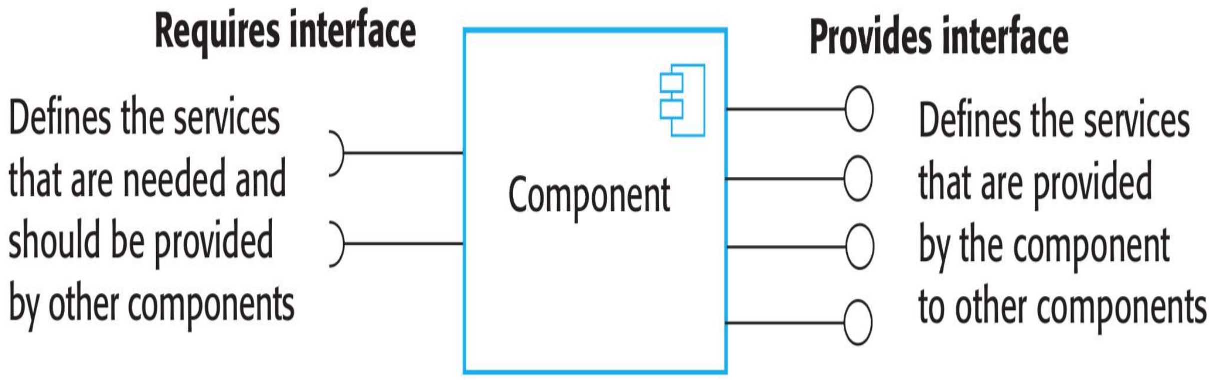
**Component interfaces:**

**"Provides" interface**

Defines the services that are provided by the component to other components. This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.

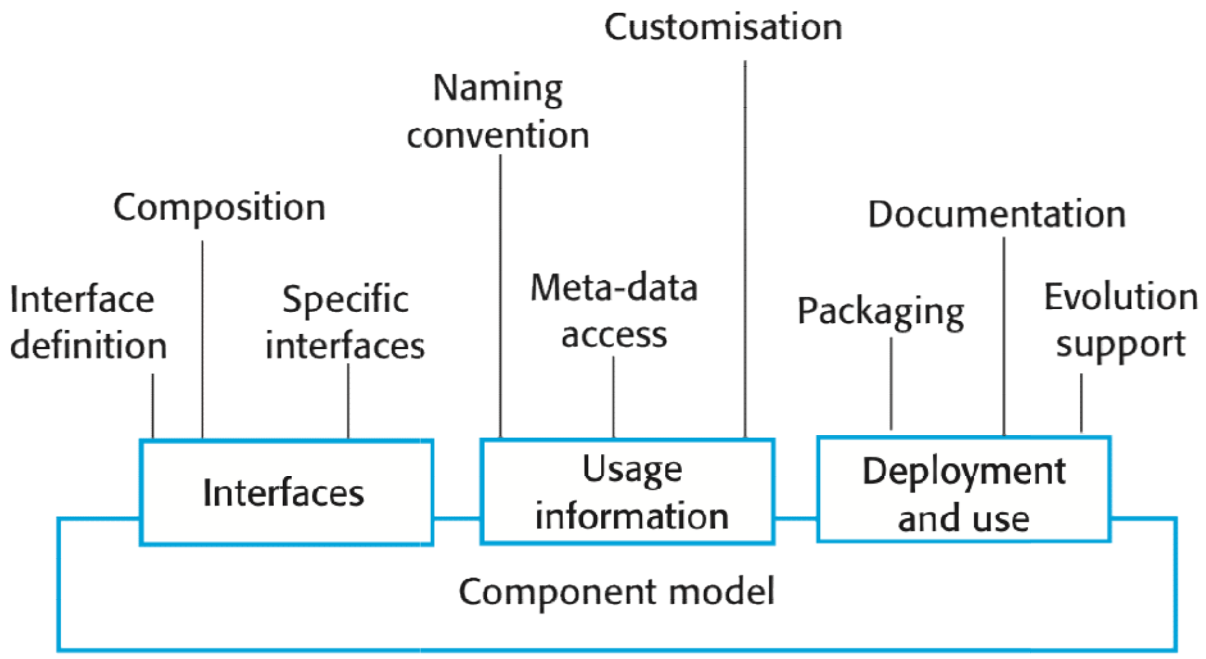
**"Requires" interface**

Defines the services that specifies what services must be made available for the component to execute as specified. This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided.


Requires interface
Defines the services that are needed and should be provided by other components
Component
Provides interface
Defines the services that are provided by the component to other components

Components are accessed using remote procedure calls (RPCs). Each component has a unique identifier (usually a URL) and can be referenced from any networked computer. Therefore it can be called in a similar way as a procedure or method running on a local computer.

A **component model** is a definition of standards for component implementation, documentation and deployment. Examples of component models: EJB model (Enterprise Java Beans), COM+ model (.NET model), Corba Component Model. The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.



**Elements of a component model:**

**Interfaces**

Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.

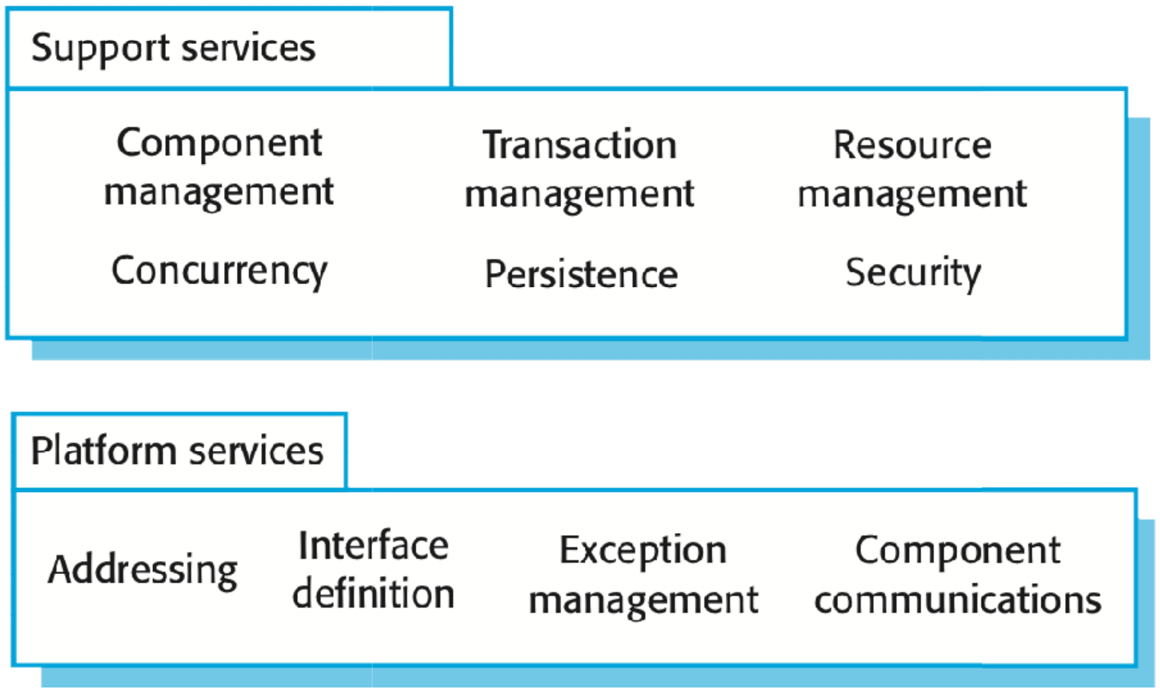
**Usage**

In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique.

**Deployment**

The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

Component models are the basis for middleware that provides support for executing components. Component model implementations provide: **platform services** that allow components written according to the model to communicate, and **support services** that are application-independent services used by different components. To use services provided by a model, components are deployed in a **container**. This is a set of interfaces used to access the service implementations.



## CBSE processes

CBSE processes are software processes that support component-based software engineering. They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components. There are two types of CBSE processes:

- **CBSE for reuse** is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.
- **CBSE with reuse** is the process of developing new applications using existing components and services.

**CBSE for reuse** focuses on component and service development. Components developed for a specific application usually have to be generalised to make them reusable. A component is most likely to be reusable if it associated with a stable domain abstraction (business object). For example, in a hospital stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

**Component reusability:**

- Should reflect stable domain abstractions;
- Should hide state representation;
- Should be as independent as possible;
- Should publish exceptions through the component interface.

There is a **trade-off between reusability and usability**. The more general the interface, the greater the reusability but it is then more complex and hence less usable.

To make an existing component reusable:

- Remove application-specific methods.
- Change names to make them general.
- Add methods to broaden coverage.
- Make exception handling consistent.
- Add a configuration interface for component adaptation.
- Integrate required components to reduce dependencies.

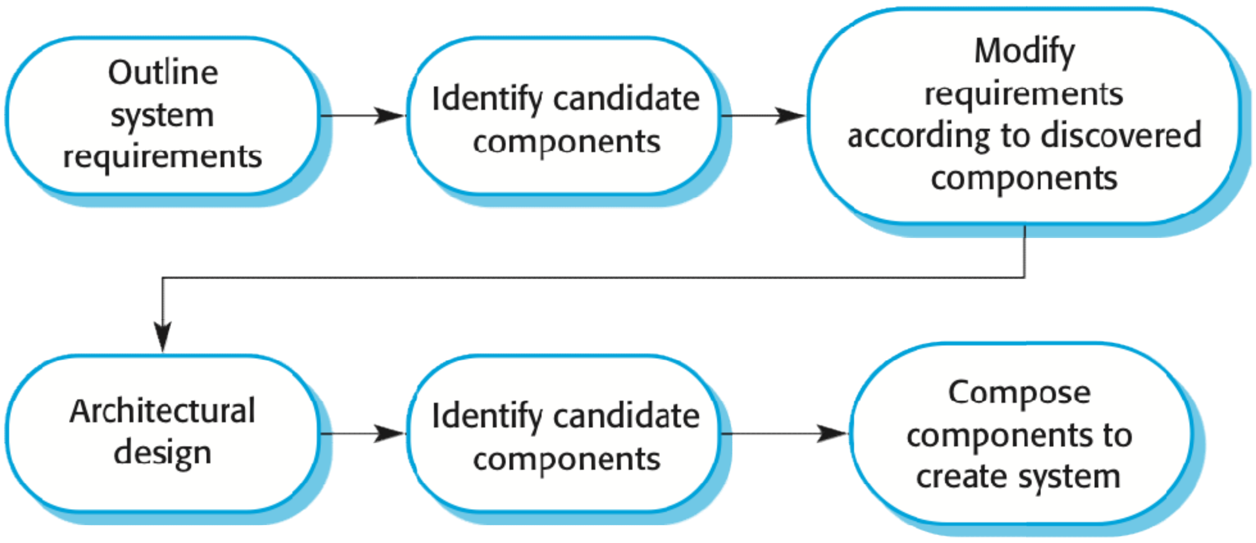
Existing **legacy systems** that fulfill a useful business function can be re-packaged as components for reuse. This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system. Although costly, this can be much less expensive than rewriting the legacy system.

The **development cost** of reusable components may be higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost. Generic components may be less space-efficient and may have longer execution times than their specific equivalents.

**Component management** involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions. A company with a reuse program may carry out some form of component certification before the component is made available for reuse. Certification means that someone apart from the developer checks the quality of the component.

**CBSE with reuse** process has to find and integrate reusable components. When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components. This involves:

- Developing outline requirements;
- Searching for components then modifying requirements according to available functionality;
- Searching again to find if there are better components that meet the revised requirements;
- Composing components to create the system.


Outline system requirements
Identify candidate components
Modify requirements according to discovered components
Architectural design
Identify candidate components
Compose components to create system

Component identification issues:

- You need to be able to **trust** the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.
- Different groups of components will satisfy different **requirements**.
- **Validation.** The component specification may not be detailed enough to allow comprehensive tests to be developed. Components may have unwanted functionality. How can you test this will not interfere with your application?

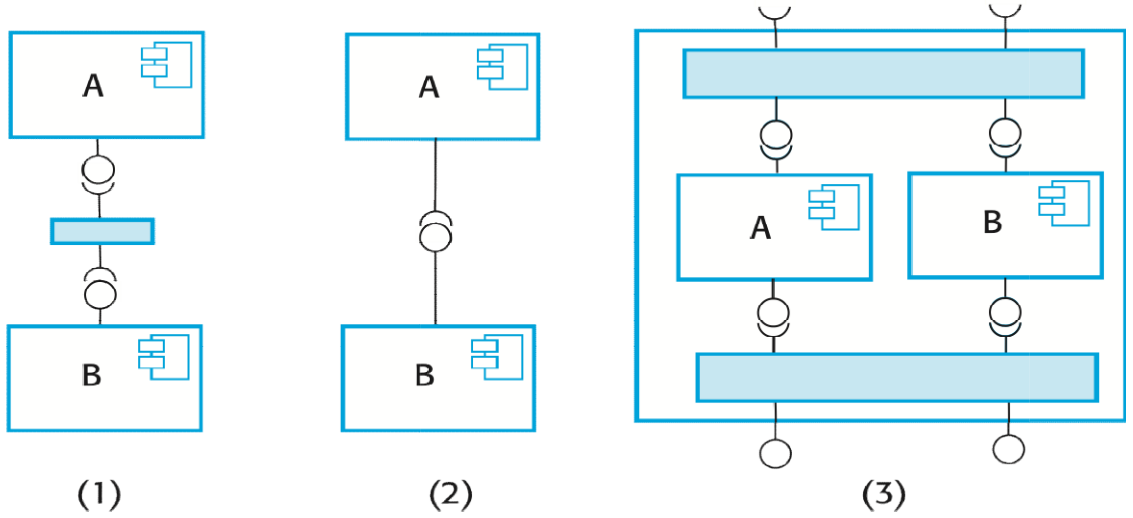
Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests. The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need.

## Component composition

Component composition is the process of **assembling components to create a system**. Composition involves integrating components with each other and with the component infrastructure. Normally you have to write 'glue code' to integrate components.

Three types of component composition:

1. **Sequential composition** where the composed components are executed in sequence. This involves composing the provides interfaces of each component.
2. **Hierarchical composition** where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
3. **Additive composition** where the interfaces of two components are put together to create a new component. Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.



When components are developed independently for reuse, their interfaces are often incompatible. Three types of incompatibility can occur:

- **Parameter incompatibility** where operations have the same name but are of different types.
- **Operation incompatibility** where the names of operations in the composed interfaces are different.
- **Operation incompleteness** where the provides interface of one component is a subset of the requires interface of another.

Adaptor components address the problem of component incompatibility by reconciling the interfaces of the components that are composed. Different types of adaptor are required depending on the type of composition.

When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution. You need to make decisions such as:

- What composition of components is effective for delivering the functional requirements?
- What composition of components allows for future change?
- What will be the emergent properties of the composed system?

## Distributed Software Engineering

Virtually all large computer-based systems are now distributed systems. A distributed system is "a collection of independent computers that appears to the user as a single coherent system." Information processing is distributed over several computers rather than confined to a single machine. Distributed software engineering is therefore very important for enterprise computing systems.

**Benefits** of distributed systems:

- **Resource sharing:** sharing of hardware and software resources.
- **Openness:** use of equipment and software from different vendors.
- **Concurrency:** concurrent processing to enhance performance.
- **Scalability:** increased throughput by adding new resources.
- **Fault tolerance:** the ability to continue in operation after a fault has occurred.

### Distributed systems

Distributed systems are more complex than systems that run on a single processor. Complexity arises because different parts of the system are independently managed as is the network. There is no single authority in charge of the system so top-down control is impossible.

**Design issues** in distributed systems engineering:

**Transparency: to what extent should the distributed system appear to the user as a single system?**

Ideally, users should not be aware that a system is distributed and services should be independent of distribution characteristics. In practice, this is impossible because parts of the system are independently managed and because of network delays. Often better to make users aware of distribution so that they can cope with problems. To achieve transparency, resources should be abstracted and addressed logically rather than physically. Middleware maps logical to physical resources.

**Openness: should a system be designed using standard protocols that support interoperability?**

Open distributed systems are systems that are built according to generally accepted standards. Components from any supplier can be integrated into the system and can inter-operate with the other system components. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components. Web service standards for service-oriented architectures were developed to be open standards.

**Scalability: how can the system be constructed so that it is scaleable?**

The scalability of a system reflects its ability to deliver a high quality service as demands on the system increase:

- **Size:** it should be possible to add more resources to a system to cope with increasing numbers of users.
- **Distribution:** it should be possible to geographically disperse the components of a system without degrading its performance.
- **Manageability:** it should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.

There is a distinction between scaling-up and scaling-out. Scaling up is more powerful system; scaling out is more system instances.

**Security: how can usable security policies be defined and implemented?**

When a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems. If a part of the system is successfully attacked then the attacker may be able to use this as a 'back door' into other parts of the system. Difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. The types of attack that a distributed system must defend itself against are:

- **Interception,** where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
- **Interruption,** where system services are attacked and cannot be delivered as expected.
- **Denial of service** attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
- **Modification,** where data or services in the system are changed by an attacker.
- **Fabrication,** where an attacker generates information that should not exist and then uses this to gain some privileges.

**Quality of service: how should the quality of service be specified?**

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users. Quality of service is particularly critical when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand.

**Failure management: how can system failures be detected, contained and repaired?**

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. *"You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."* Distributed systems should include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, automatically recover from the failure.

**Two types of interaction** between components in a distributed system:

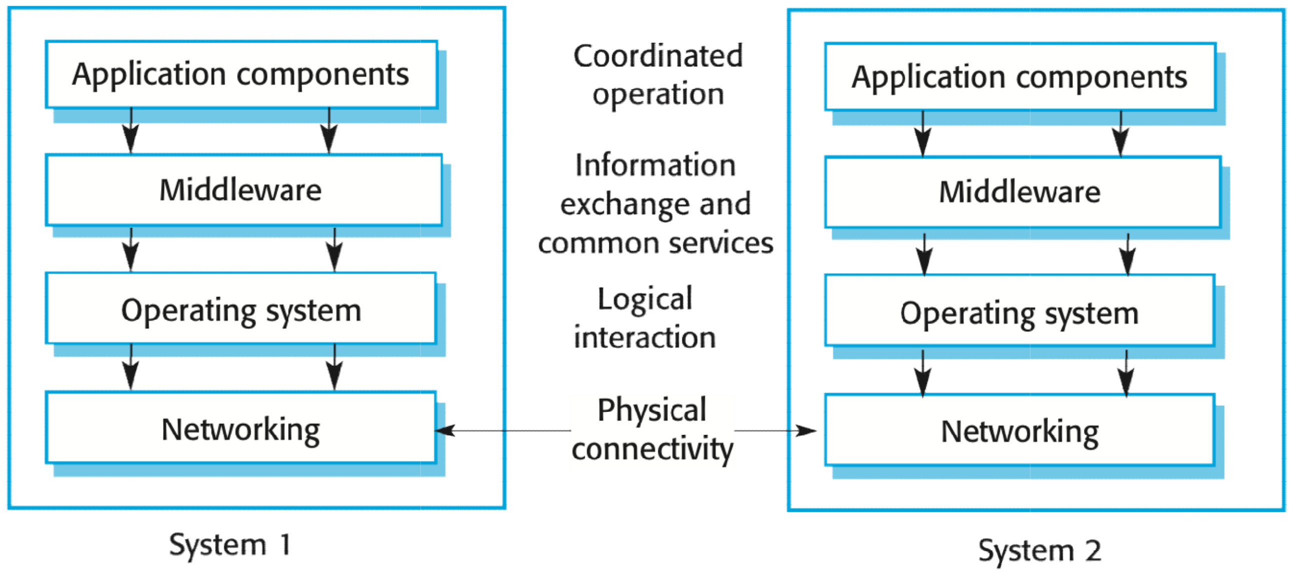
**Procedural interaction, where one computer calls on a known service offered by another computer and waits for a response.**

Procedural communication in a distributed system is implemented using remote procedure calls (RPC). In a remote procedure call, one component calls another component as if it was a local procedure or method. The middleware in the system intercepts this call and passes it to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. A problem with RPCs is that the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other.

**Message-based interaction, involves the sending computer sending information about what is required to another computer. There is no necessity to wait for a response.**

Message-based interaction normally involves one component creating a message that details the services required from another component. Through the system middleware, this is sent to the receiving component. The receiver parses the message, carries out the computations and creates a message for the sending component with the required results. In a message-based approach, it is not necessary for the sender and receiver of the message to be aware of each other. They simple communicate with the middleware.

Middleware is software that can manage diverse components of a distributed system, and ensure that they can communicate and exchange data. The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation and protocols for communication may all be different.



Interaction support,where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components. The provision of common services, where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily inter-operate and provide user services in a consistent way.

## Client-server computing

Distributed systems that are accessed over the Internet are normally organized as client-server systems. In a client-server system, the user interacts with a program running on their local computer (e.g. a web browser or mobile application). This interacts with another program running on a remote computer (e.g. a web server). The remote computer provides services, such as access to web pages, which are available to external clients.

**Layers** in a client/server system:

- **Presentation** is concerned with presenting information to the user and managing all user interaction.
- **Data handling** manages the data that is passed to and from the client. Implement checks on the data, generate web pages, etc.
- **Application processing layer** is concerned with implementing the logic of the application and so providing the required functionality to end users.
- **Database** stores data and provides transaction management services, etc.

### Architectural patterns for distributed systems

Common **architectural patterns** for organizing the architecture of a distributed system:

**Master-slave architecture**

Master-slave architectures are commonly used in real-time systems in which guaranteed interaction response times are required. There may be separate processors associated with data acquisition from the system's environment, data processing and computation and actuator management. The 'master' process is usually responsible for computation, coordination and communications and it controls the 'slave' processes. 'Slave' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

**Two-tier client-server architecture**

In a two-tier client-server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server. **Thin-client** model, where the presentation layer is implemented on the client and all other layers (data management, application processing and database) are implemented on a server. **Fat-client** model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. Distinction between thin and fat client architectures has become blurred. Javascript allows local processing in a browser so 'fat-client' functionality available without software installation. Mobile apps carry out some local processing to minimize demands on network. Auto-update of apps reduces management problems. There are now very few thin-client applications with all processing carried out on remote server.

**Multi-tier client-server architecture**

In a multi-tier client-server architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors. This avoids problems with scalability and performance if a thin-client two-tier model is chosen, or problems of system management if a fat-client model is used. Used when there is a high volume of transactions to be processed by the server.

**Distributed component architecture**

There is no distinction in a distributed component architecture between clients and servers. Each distributable entity is a component that provides services to other components and receives services from other components. Component communication is through a middleware system. Used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems. Benefits include:

- It allows the system designer to delay decisions on where and how services should be provided.
- It is a very open system architecture that allows new resources to be added as required.
- The system is flexible and scalable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Distributed component architectures suffer from two major disadvantages:

- They are more complex to design than client-server systems. Distributed component architectures are difficult for people to visualize and understand.
- Standardized middleware for distributed component systems has never been accepted by the community. Different vendors, such as Microsoft and Sun, have developed different, incompatible middleware.

As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

**Peer-to-peer architecture**

Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network. The overall system is designed to take advantage of the computational power and storage of a large number of networked computers. Most p2p systems have been personal systems but there is increasing business use of this technology. Used when clients exchange locally stored information and the role of the server is to introduce clients to each other. Examples:

- File sharing systems based on the BitTorrent protocol
- Messaging systems such as Jabber
- Payments systems, e.g. Bitcoin
- Databases, e.g. Freenet is a decentralized database
- Phone systems, e.g. Viber
- Computation systems, e.g. SETI@home

P2P architectures are used when

- A system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations.
- A system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally-stored or managed.

Security issues:

- Security concerns are the principal reason why p2p architectures are not widely used.
- The lack of central management means that malicious nodes can be set up to deliver spam and malware to other nodes in the network.
- P2P communications require careful setup to protect local information and if not done correctly, then this is exposed to other peers.

## Software as a service

Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC. The software is owned and managed by a software provider, rather than the organizations using the software. Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Example: Google Docs. Key elements of SaaS include:

- Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- The software is owned and managed by a software provider, rather than the organizations using the software.
- Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

Software as a service (SaaS) and service-oriented architectures (SOA) are related, but they are not the same. Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions e.g. editing a document. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something then returns a result.

**Implementation factors** for SaaS:

**Configurability**

How do you configure the software for the specific requirements of each organization? Service configuration includes:

- Branding, where users from each organization, are presented with an interface that reflects their own organization.
- Business rules and workflows, where each organization defines its own rules that govern the use of the service and its data.
- Database extensions, where each organization defines how the generic service data model is extended to meet its specific needs.
- Access control, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

**Multi-tenancy**

How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?

- Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- It must appear to each user that they have the sole use of the system.
- Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.

**Scalability**

How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

- Develop applications where each component is implemented as a simple stateless service that may be run on any server.
- Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request).
- Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
- Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.

## Service-oriented Software Engineering

A **web service** is an instance of a more general notion of a service: *"an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the*

*factors of production."*
The essence of a service, therefore, is that **the provision of the service is independent of the application** using the service. Service providers can develop specialized services and offer these to a range of service users from different organizations.

**Services are reusable components** that are **independent** (no requires interface) and are **loosely coupled**. A web service is:
*A loosely coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.*
Services are platform and implementation-language independent.

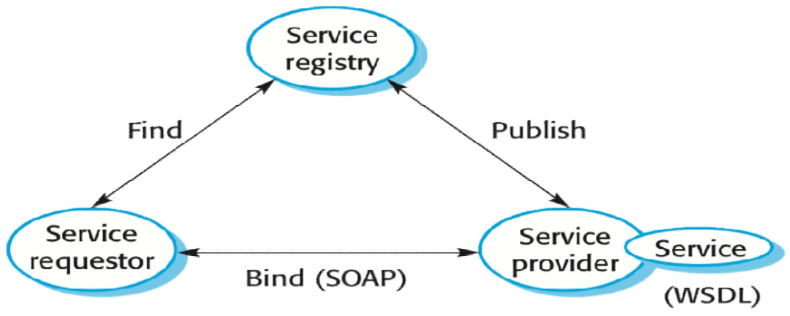
**Benefits** of a service-oriented approach:

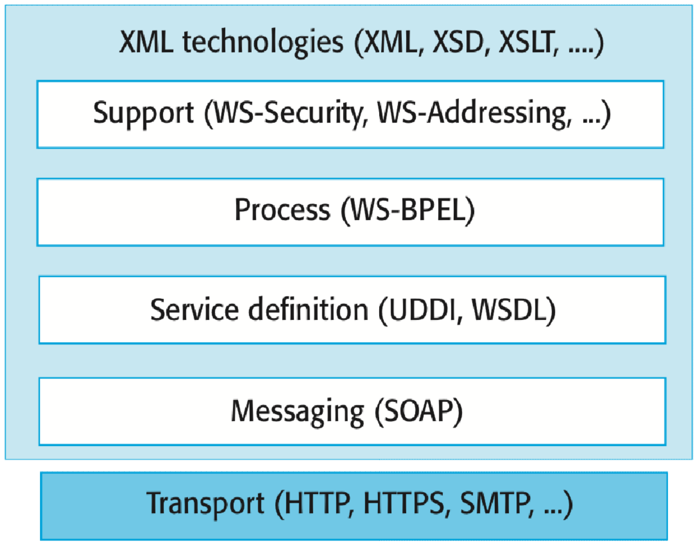
- Services can be **offered by any service provider** inside or outside of an organization so organizations can create applications by integrating services from a range of providers.
- The service provider makes information about the service public so that **any authorized user can use the service**.
- Applications can delay the binding of services until they are deployed or until execution. This means that **applications can be reactive** and adapt their operation to cope with changes to their execution environment.
- **Opportunistic construction of new services** is possible. A service provider may recognize new services that can be created by linking existing services in innovative ways.
- Service users **can pay for services according to their use** rather than their provision. Instead of buying a rarely-used component, the application developers can use an external service that will be paid for only when required.
- Applications can be made smaller, which is particularly important for mobile devices with limited processing and memory capabilities. **Computationally-intensive processing can be offloaded to external services**.

## Service-oriented architectures

Service-oriented software engineering is as **significant** as object-oriented software engineering. Building applications based on services allows companies and other organizations to cooperate and make use of each other's business functions. Service-based applications may be constructed by linking services from various providers using either a standard programming language or a specialized workflow language.

**Service-oriented architecture (SOA)** is a means of **developing distributed systems** where the **components are stand-alone services**. Services may execute on different computers from different service providers. Standard protocols have been developed to support service communication and information exchange.




XML technologies (XML, XSD, XSLT, ....)
Support (WS-Security, WS-Addressing, ...)
Process (WS-BPEL)
Service definition (UDDI, WSDL)
Messaging (SOAP)
Transport (HTTP, HTTPS, SMTP, ...)

**Benefits** of SOA:

- Services can be **provided locally or outsourced** to external providers
- Services are **language-independent**
- Investment in **legacy systems** can be preserved
- Inter-organizational computing is facilitated through **simplified information exchange**

Key **standards**:

- **SOAP (Simple Object Access Protocol)**: a message exchange standard that supports service communication
- **WSDL (Web Service Definition Language):** allows a service interface and its bindings to be defined
- **WS-BPEL (Web Services Business Process Execution Language):** a standard for workflow languages used to define service composition

Existing approaches to software engineering have to evolve to reflect the service-oriented approach to software development:

- **Service engineering** (software development for reuse): the development of dependable, reusable services.

- **Software development with services** (software development with reuse): the development of dependable software where services are the fundamental components.

A **service** canbedefinedas:
*A loosely-coupled, reusable software component that encapsulates discrete functionality which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols*
A critical distinction between a service and a component as defined in CBSE is that services are independent. Services do not have a 'requires' interface. Services rely on message-based communication with messages expressed in XML.

The **service interface** is defined in a service description expressed in WSDL (Web Service Description Language). The WSDL specification defines:

- **What** operations the service supports and the format of the messages that are sent and received by the service. The 'what' part of a WSDL document, called an interface, specifies what operations the service supports, and defines the format of the messages that are sent and received by the service.
- **How** the service is accessed - that is, the binding maps the abstract interface onto a concrete set of protocols. The 'how' part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a Web service.
- **Where** the service is located. This is usually expressed as a URI (Universal Resource Identifier). The 'where' part of a WSDL document describes the location of a specific Web service implementation (its endpoint).

## RESTful services

Current web services standards have been criticized as 'heavyweight' standards that are over-general and inefficient. **REST (REpresentational State Transfer)** is an architectural style based on **transferring representations of resources from a server to a client**. This style **underlies the web as a whole** and is simpler than SOAP/WSDL for implementing web services. RESTful services involve a lower overhead than so-called 'big web services' and are used by many organizations implementing service-based systems.

The **fundamental element in a RESTful architecture** is a **resource**. Essentially, a resource is simply **a data element** such as a catalog, a medical record, or a document. In general, resources may have multiple representations i.e. they can exist in different formats.

**Resource operations:**

- **Create** - bring the resource into existence.
- **Read** - return a representation of the resource.
- **Update** - change the value of the resource.
- **Delete** - make the resource inaccessible.

**The Web is an example of a system that has a RESTful architecture.** Web pages are resources, and the unique identifier of a web page is its URL.

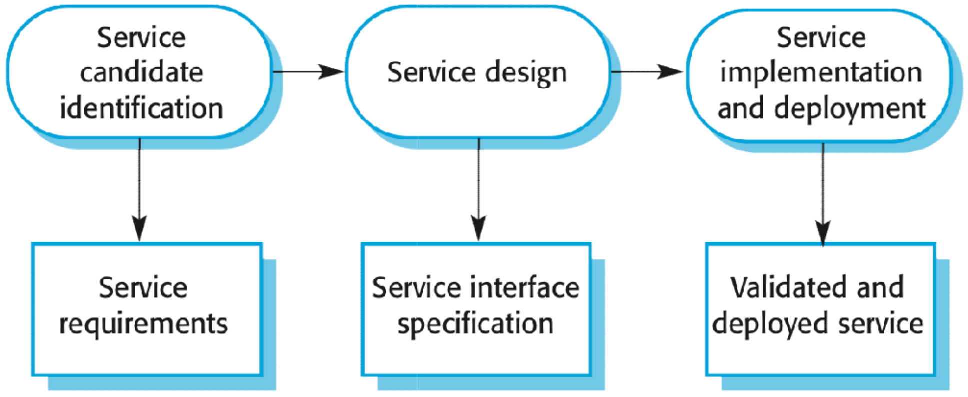
- POST is used to create a resource. It has associated data that defines the resource.
- GET is used to read the value of a resource and return that to the requestor in the specified representation, such as XHTML, that can be rendered in a web browser.
- PUT is used to update the value of a resource.
- DELETE is used to delete the resource.

**Disadvantages** of RESTful approach:

- When a service has a complex interface and is not a simple resource, it can be **difficult to design** a set of RESTful services to represent this.
- There are **no standards** for RESTful interface description so service users must rely on informal documentation to understand the interface.
- When you use RESTful services, you have to implement your **own infrastructure** for monitoring and managing the **quality of service** and the **service reliability**.

## Service engineering

Service engineering is the process of **developing services for reuse** in service-oriented applications. The service has to be designed as a reusable abstraction that can be used in different systems. Generally useful functionality associated with that abstraction must be designed and the service must be robust and reliable. The service must be documented so that it can be discovered and understood by potential users.


Service candidate identification
Service design
Service implementation and deployment
Service requirements
Service interface specification
Validated and deployed service

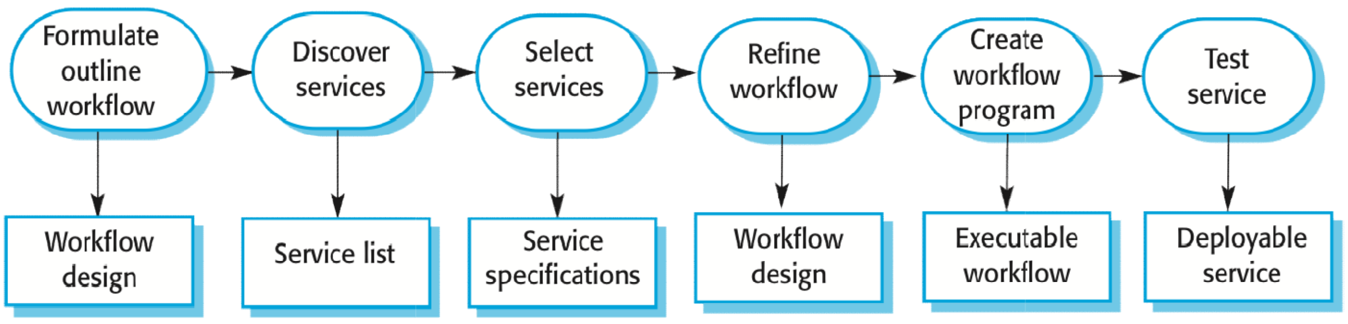
Stages of service engineering include:

- **Service candidate identification**, where you identify possible services that might be implemented and define the service requirements.It involves understanding an organization's business processes to decide which reusable services could support these processes. Three **fundamental types** of service:
  - **Utility services** that implement general functionality used by different business processes.
  - **Business services** that are associated with a specific business function e.g., in a university, student registration.
  - **Coordination services** that support composite processes such as ordering.

- **Service design**, where you design the logical service interface and its implementation interfaces (SOAP and/or RESTful). Involves thinking about the operations associated with the service and the messages exchanged. The number of messages exchanged to complete a service request should normally be minimized. Service state information may have to be included in messages. Interface design stages:
    - **Logical interface design.** Starts with the service requirements and defines the operation names and parameters associated with the service. Exceptions should also be defined.
    - **Message design (SOAP).** For SOAP-based services, design the structure and organization of the input and output messages. Notations such as the UML are a more abstract representation than XML. The logical specification is converted to a WSDL description.
    - **Interface design (REST).** Design how the required operations map onto REST operations and what resources are required.
- **Service implementation and deployment**, where you implement and test the service and make it available for use. Programming services using a standard programming language or a workflow language. Services then have to be tested by creating input messages and checking that the output messages produced are as expected. Deployment involves publicizing the service and installing it on a web server. Current servers provide support for service installation.

## Service composition

Existing services are composed and configured to create new composite services and applications. The basis for service composition is often a workflow. Workflows are logical sequences of activities that, together, model a coherent business process. For example, provide a travel reservation services which allows flights, car hire and hotel bookings to be coordinated.



**Service construction by composition:**

**Formulate outline workflow**

In this initial stage of service design, you use the requirements for the composite service as a basis for creating an 'ideal' service design.

**Discover services**

During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services and the details of the service provision.

**Select possible services**

Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

**Refine workflow**

This involves adding detail to the abstract description and perhaps adding or removing workflow activities.

**Create workflow program**

During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or a workflow language, such as WS-BPEL.

**Test completed service or application**

The process of testing the completed, composite service is more complex than component testing in situations where external services are used.

## Real-time Software Engineering

Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants. Their software must react to events generated by the hardware and, often, issue control signals in response to these events. **The software in these systems is embedded in system hardware**, often in read-only memory, and usually responds, in real time, to events from the system's environment.

**Responsiveness in real-time** is the critical difference between embedded systems and other software systems, such as information systems, web-based systems or personal software systems. For non-real-time systems, correctness can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system. In a real-time system, the **correctness depends both on the response to an input and the time taken to generate that response**. If the system takes too long to respond, then the required response may be ineffective.

A **real-time system** is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. A **soft real-time system** is a system whose operation is degraded if results are not produced according to the specified timing requirements. A **hard real-time system** is a system whose operation is incorrect if results are not produced according to the timing specification.

**Characteristics of embedded systems:**

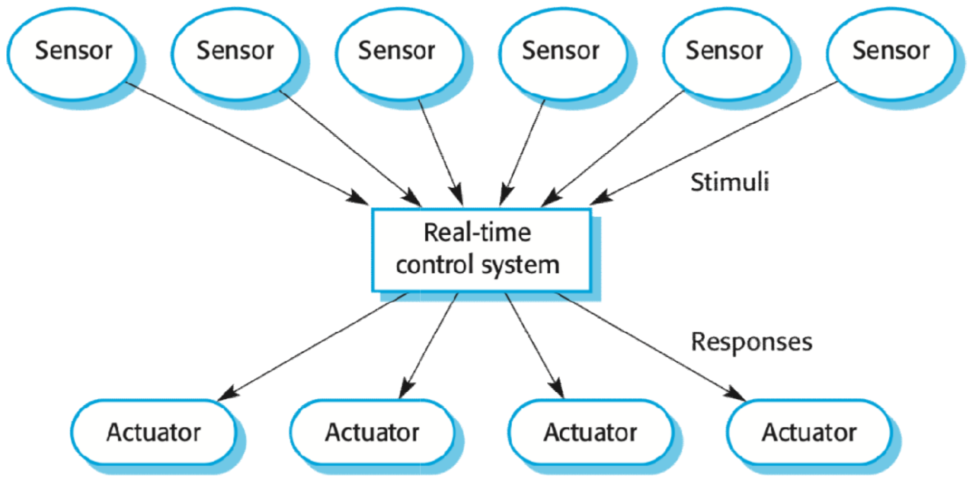
- Embedded systems generally **run continuously** and do not terminate.
- **Interactions** with the system's environment are **unpredictable**.
- There may be **physical limitations** that affect the design of a system.
- **Direct hardware interaction** may be necessary.
- Issues of **safety and reliability** may dominate the system design.

## Embedded system design

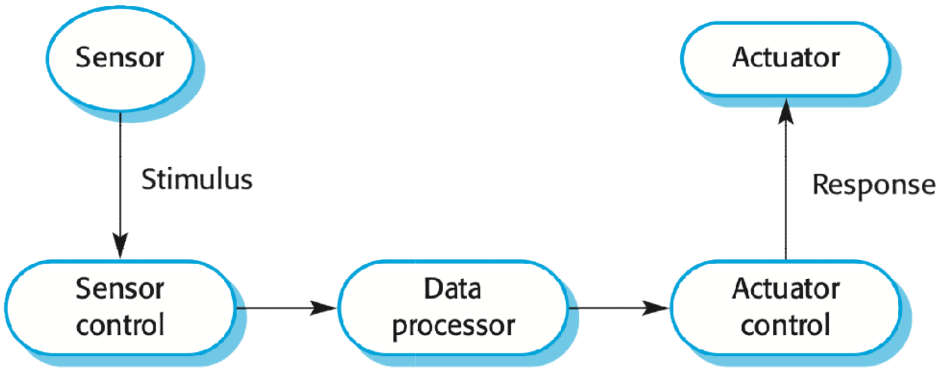
The design process for embedded systems is a systems engineering process that has to consider, in detail, the design and performance of the system hardware. Part of the design process may involve deciding which system capabilities are to be implemented in software and which in hardware. Low-level decisions on hardware, support software and system timing must be considered early in the process. These may mean that additional software functionality, such as battery and power management, has to be included in the system.

Real-time systems are often considered to be **reactive systems**. Given a stimulus, the system must produce a reaction or response within a specified time. Stimuli come from sensors in the systems environment and from actuators controlled by the system.

- **Periodic stimuli** occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
- **Aperiodic stimuli** occur irregularly and unpredictably and are may be signalled using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.



Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for **fast switching between stimulus handlers**. Timing demands of different stimuli are different so a simple sequential loop is not usually adequate. Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.



- **Sensor control processes** collect information from sensors. May buffer information collected in response to a sensor stimulus.
- **Data processor** carries out processing of collected information and computes the system response.
- **Actuator control processes** generate control signals for the actuators.

Processes in a real-time system have to be coordinated and share information. **Process coordination mechanisms ensure mutual exclusion to shared resources.** When one process is modifying a shared resource, other processes should not be able to change that resource. When designing the information exchange between processes, you have to take into account the fact that these processes may be running at different speeds.

Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available. Producer and consumer processes must be mutually excluded from accessing the same element. The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.

**The effect of a stimulus in a real-time system may trigger a transition from one state to another.** State models are therefore often used to describe embedded real-time systems. UML state diagrams may be used to show the states and state transitions in a real-time system.

Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages. Systems-level languages, such as C, which allow efficient code to be generated are widely used in preference to languages such as Java. There is a performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The loss of performance may make it impossible to meet real-time deadlines.

## Architectural patterns for real-time software

Characteristic system architectures for embedded systems:

- **Observe and React** pattern is used when a set of sensors are routinely monitored and displayed.

- **Environmental Control** pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment.
- **Process Pipeline** pattern is used when data has to be transformed from one representation to another before it can be processed.

**Observe and React pattern description**

The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, in certain cases, to take actions in response to the exceptional value.

**Stimuli**

Values from sensors attached to the system.

**Responses**

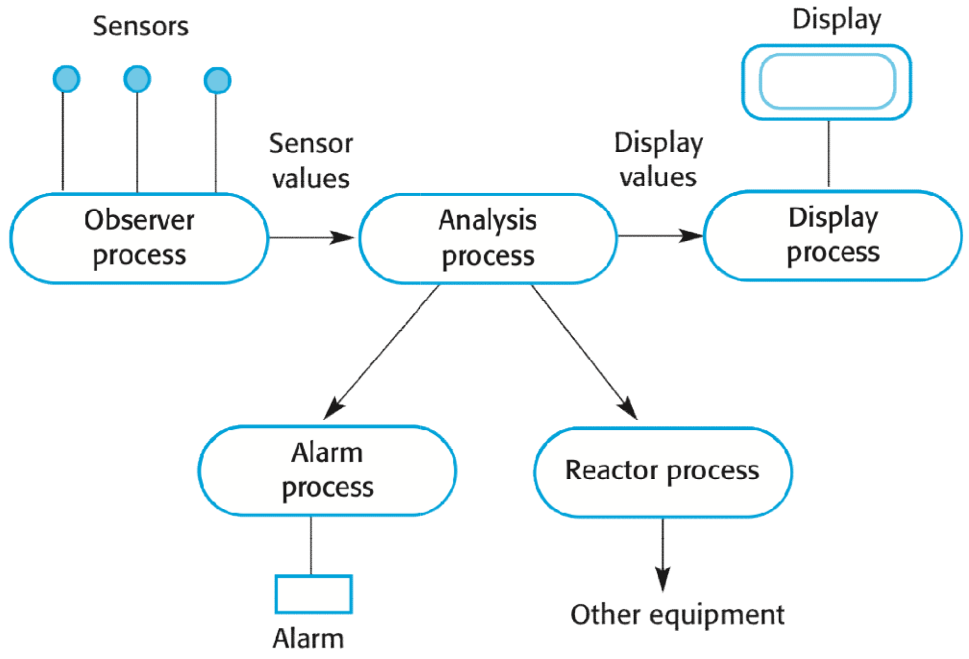
Outputs to display, alarm triggers, signals to reacting systems.

**Processes**

Observer, Analysis, Display, Alarm, Reactor.

**Used in**

Monitoring systems, alarm systems.



**Environmental Control pattern description**

The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators that then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.

**Stimuli**

Values from sensors attached to the system and the state of the system actuators.

**Responses**

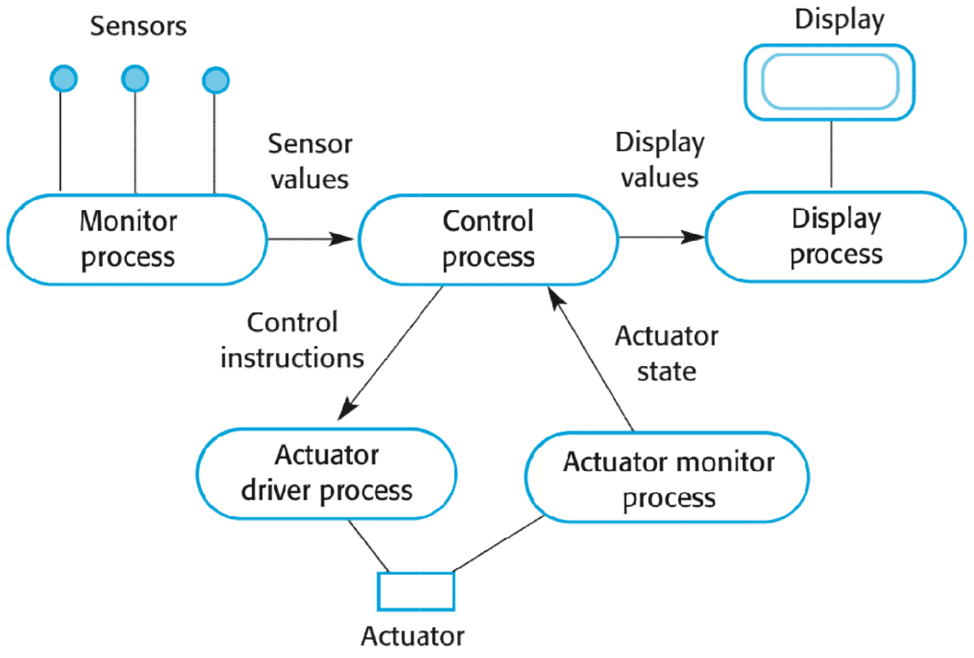
Control signals to actuators, display information.

**Processes**

Monitor, Control, Display, Actuator Driver, Actuator monitor.

**Used in**

Control systems.



**Process Pipeline pattern description**

A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage or the pipeline may terminate in an actuator.

**Stimuli**

Input values from the environment or some other process

**Responses**

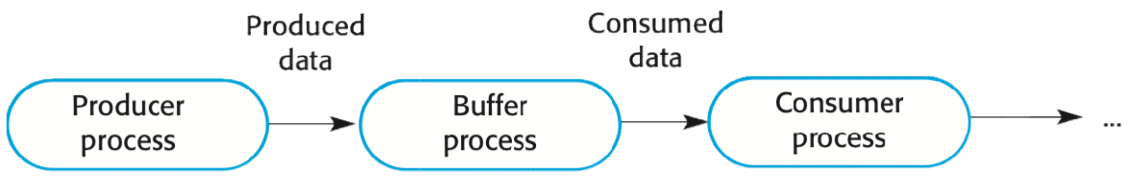
Output values to the environment or a shared buffer

**Processes**

Producer, Buffer, Consumer

**Used in**

Data acquisition systems, multimedia systems



## Timing analysis

The correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced. In a timing analysis, you calculate how often each process in the system must be executed to ensure that all inputs are processed and all system responses produced in a timely way. The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.

**Factors in timing analysis:**

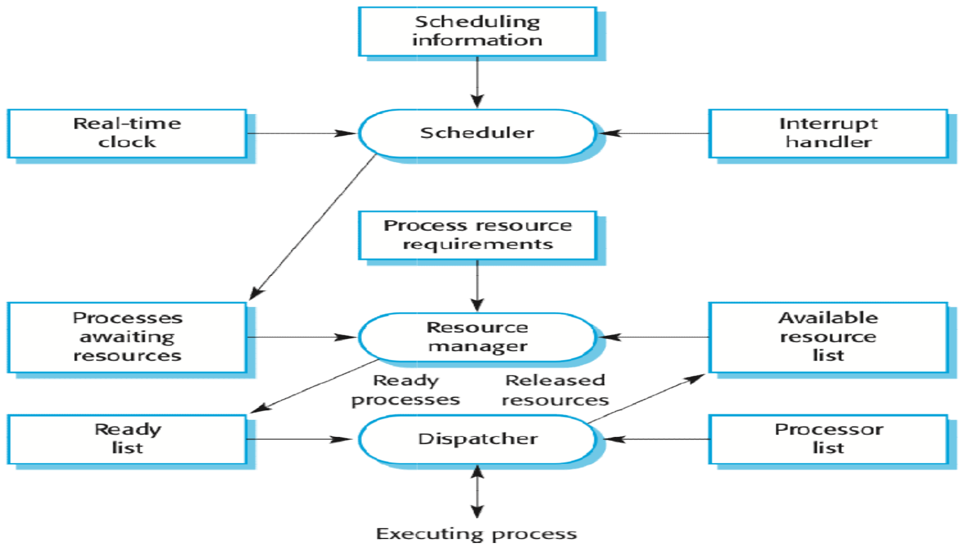
- **Deadlines**: the times by which stimuli must be processed and some response produced by the system.
- **Frequency**: the number of times per second that a process must execute so that you are confident that it can always meet its deadlines.
- **Execution time**: the time required to process a stimulus and produce a response.

## Real-time operating systems

Real-time operating systems are specialized operating systems which manage the processes in the RTS. Responsible for process management and
resource (processor and memory) allocation. May be based on a standard kernel which
is used unchanged or modified for a particular
application. Do not normally include facilities such as file management.

**Real-time operating system components:**

- **Real-time clock** provides information for process scheduling.
- **Interrupt handler** manages aperiodic requests for service.
- **Scheduler** chooses the next process to be run.
- **Resource manager** allocates memory and processor resources.
- **Dispatcher** starts process execution.



The scheduler chooses the next process to be executed by the processor. This depends on a scheduling strategy which may take the process priority into account. The resource manager allocates memory and a processor for the process to be executed. The dispatcher takes the process from ready list, loads it onto a processor and starts execution.

**Scheduling strategies:**

- **Non pre-emptive scheduling:** once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).
- **Pre-emptive scheduling:** the execution of an executing processes may be stopped if a higher priority process requires service.
- Scheduling algorithms include round-robin, rate monotonic, and shortest deadline first.

## What is Systems Engineering?

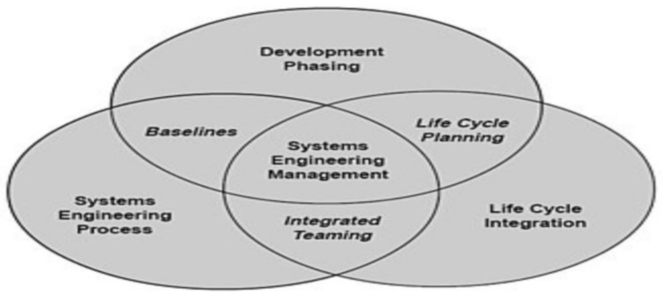
Systems Engineering is an engineering field that takes an interdisciplinary approach to product development. Systems engineers analyze the collection of pieces to make sure when working together, they achieve the intended objectives or purpose of the product. For example, in automotive development, a propulsion system or braking system will involve mechanical engineers, electrical engineers, and a host of other specialized engineering disciplines. A systems engineer will focus on making each of the individual systems work together into an integrated whole that performs as expected across the lifecycle of the product.

In this chapter, we discuss:

- The fundamentals of systems engineering
- The role of a systems engineer
- Systems engineering process
- The "V" Model of systems engineering

### What are the fundamentals of systems engineering?

In product development, systems engineering is the interdisciplinary field that focuses on designing, integrating, and managing the systems that work together to form a more complex system. Systems engineering is based around systems thinking principles, and the goal of a systems engineer is to help a product team produce an engineered system that performs a useful function as defined by the requirements written at the beginning of the project. The final product should be one where the individual systems work together in a cohesive whole that meet the requirements of the product.



## What is a system?

A system is a collection of different elements that produce results that individual elements cannot produce. Elements or parts can be wide-ranging and include people, hardware, software, facilities, policies, and documents. These elements interact with each other according to a set of rules that produce a unified whole with a purpose expressed by its functioning. An example of a system is the human auditory system; the system includes individual parts in the form of bones and tissue that interact in a way to produce sound waves, which are transferred to nerves that lead to the brain, which interprets the sounds and formulates a response. If any single part in the auditory system fails or experiences disruption, the entire system can fail to perform its function.

## What is the role of a systems engineer?

A systems engineer is tasked with looking at the entire integrated system and evaluating it against its desired outcomes. In that role, the systems engineer must know a little bit about everything and have an ability to see the "big picture." While specialists can focus on their specific disciplines, the systems engineer must evaluate the complex system as a whole against the initial requirements and desired outcomes.

Systems engineers have multi-faceted roles to play but primarily assist with:

- Design compatibility
- Definition of requirements
- Management of projects
- Cost analysis
- Scheduling
- Possible maintenance needs
- Ease of operations
- Future systems upgrades
- Communication among engineers, managers, suppliers, and customers in regards to the system's operations

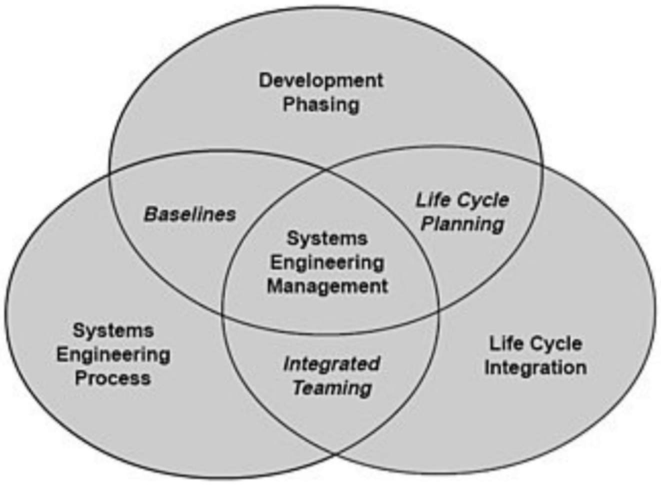
Systems Engineering is an engineering field that takes an interdisciplinary approach to product development. Systems engineers analyze the collection of pieces to make sure when working together, they achieve the intended objectives or purpose of the product. For example, in automotive development, a propulsion system or braking system will involve mechanical engineers, electrical engineers, and a host of other specialized engineering disciplines. A systems engineer will focus on making each of the individual systems work together into an integrated whole that performs as expected across the lifecycle of the product.

In this chapter, we discuss:

- The fundamentals of systems engineering
- The role of a systems engineer
- Systems engineering process
- The “V” Model of systems engineering

## What are the fundamentals of systems engineering?

In product development, systems engineering is the interdisciplinary field that focuses on designing, integrating, and managing the systems that work together to form a more complex system. Systems engineering is based around systems thinking principles, and the goal of a systems engineer is to help a product team produce an engineered system that performs a useful function as defined by the requirements written at the beginning of the project. The final product should be one where the individual systems work together in a cohesive whole that meet the requirements of the product.



## What is a system?

A system is a collection of different elements that produce results that individual elements cannot produce. Elements or parts can be wide-ranging and include people, hardware, software, facilities, policies, and documents. These elements interact with each other according to a set of rules that produce a unified whole with a purpose expressed by its functioning. An example of a system is the human auditory system; the system includes individual parts in the form of bones and tissue that interact in a way to produce sound waves, which are transferred to nerves that lead to the brain, which interprets the sounds and formulates a response. If any single part in the auditory system fails or experiences disruption, the entire system can fail to perform its function.

RELATED ARTICLE: The Complete Guide to the Systems Engineering Body of Knowledge (SEBoK)

## What is systems thinking?

Systems thinking is a way of thinking that looks at the overall function of a complex system rather than breaking it down into smaller parts. For example, systems thinking would consider an automobile a complex system that consists of smaller, specialized elements. While an electrical engineer might only be concerned with the electrical system of the automobile, someone looking at the entire complex system would consider how the electrical system would impact other systems in the automobile — and how those other systems might impact the electrical system. If one piece of the electrical system fails, for instance, how would that failure cascade to other systems to impact the operability of the automobile? Systems thinking will take a "big picture" approach to the overall product.

## What is the role of a systems engineer?

A systems engineer is tasked with looking at the entire integrated system and evaluating it against its desired outcomes. In that role, the systems engineer must know a little bit about everything and have an ability to see the "big picture." While specialists can focus on their specific disciplines, the systems engineer must evaluate the complex system as a whole against the initial requirements and desired outcomes.

Systems engineers have multi-faceted roles to play but primarily assist with:

- Design compatibility
- Definition of requirements
- Management of projects
- Cost analysis
- Scheduling
- Possible maintenance needs
- Ease of operations
- Future systems upgrades
- Communication among engineers, managers, suppliers, and customers in regards to the system's operations

### How can systems engineers help improve traceability?

For many systems engineers, balancing the needs of the individual systems and their engineers against the system as a whole results in addressing problems after the fact, holding unwanted meetings, and trying to persuade others to change behavior. Many organizations may not adequately focus on requirements and traceability, resulting in a lack of data that would allow a systems engineer to better evaluate the product.

To avoid constantly chasing problems and start streamlining processes, systems engineers can use three best practices:

- **Baseline the current traceability performance:** Traceability spans the product development process, and product team members understand the value of data management, especially as concerns meeting industry requirements. By establishing a baseline of traceability performance, the entire team will be able to see existing risks and potential savings and improvements. In addition, a baseline can give a foundation for a plan of action to move toward Live Traceability.
- **Build the business case for Live Traceability:** With a baseline in hand, systems engineers can offer a case for moving to Live Traceability based on data. The data can establish the ROI, productivity improvements, and risk reduction of moving from static traceability to Live Traceability.
- **Create quick wins:** Once the advantages of Live Traceability are established, the systems engineer can set up continuous syncing between requirements and task management programs, thus automating traceability from requirements to user stories. This simple shift can help demonstrate the value of shifting from after-the-fact traceability to Live Traceability.

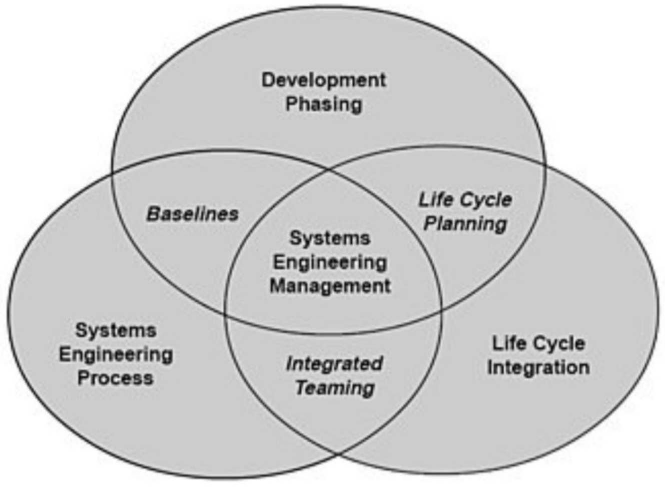
Systems Engineering is an engineering field that takes an interdisciplinary approach to product development. Systems engineers analyze the collection of pieces to make sure when working together, they achieve the intended objectives or purpose of the product. For example, in automotive development, a propulsion system or braking system will involve mechanical engineers, electrical engineers, and a host of other specialized engineering disciplines. A systems engineer will focus on making each of the individual systems work together into an integrated whole that performs as expected across the lifecycle of the product.

In this chapter, we discuss:

- The fundamentals of systems engineering
- The role of a systems engineer
- Systems engineering process
- The “V” Model of systems engineering

## What are the fundamentals of systems engineering?

In product development, systems engineering is the interdisciplinary field that focuses on designing, integrating, and managing the systems that work together to form a more complex system. Systems engineering is based around systems thinking principles, and the goal of a systems engineer is to help a product team produce an engineered system that performs a useful function as defined by the requirements written at the beginning of the project. The final product should be one where the individual systems work together in a cohesive whole that meet the requirements of the product.



## What is a system?

A system is a collection of different elements that produce results that individual elements cannot produce. Elements or parts can be wide-ranging and include people, hardware, software, facilities, policies, and documents. These elements interact with each other according to a set of rules that produce a unified whole with a purpose expressed by its functioning. An example of a system is the human auditory system; the system includes individual parts in the form of bones and tissue that interact in a way to produce sound waves, which are transferred to nerves that lead to the brain, which interprets the sounds and formulates a response. If any single part in the auditory system fails or experiences disruption, the entire system can fail to perform its function.

## What is systems thinking?

Systems thinking is a way of thinking that looks at the overall function of a complex system rather than breaking it down into smaller parts. For example, systems thinking would consider an automobile a complex system that consists of smaller, specialized elements. While an electrical engineer might only be concerned with the electrical system of the automobile, someone looking at the entire complex system would consider how the electrical system would impact other systems in the automobile — and how those other systems might impact the electrical system. If one piece of the electrical system fails, for instance, how would that failure cascade to other systems to impact the operability of the automobile? Systems thinking will take a "big picture" approach to the overall product.

## What is the role of a systems engineer?

A systems engineer is tasked with looking at the entire integrated system and evaluating it against its desired outcomes. In that role, the systems engineer must know a little bit about everything and have an ability to see the "big picture." While specialists can focus on their specific disciplines, the systems engineer must evaluate the complex system as a whole against the initial requirements and desired outcomes.

Systems engineers have multi-faceted roles to play but primarily assist with:

- Design compatibility
- Definition of requirements
- Management of projects
- Cost analysis
- Scheduling
- Possible maintenance needs
- Ease of operations
- Future systems upgrades
- Communication among engineers, managers, suppliers, and customers in regards to the system's operations

## How can systems engineers help improve traceability?

For many systems engineers, balancing the needs of the individual systems and their engineers against the system as a whole results in addressing problems after the fact, holding unwanted meetings, and trying to persuade others to change behavior. Many organizations may not adequately focus on requirements and traceability, resulting in a lack of data that would allow a systems engineer to better evaluate the product.

To avoid constantly chasing problems and start streamlining processes, systems engineers can use three best practices:

- **Baseline the current traceability performance:** Traceability spans the product development process, and product team members understand the value of data management, especially as concerns meeting industry requirements. By establishing a baseline of traceability performance, the entire team will be able to see existing risks and potential savings and improvements. In addition, a baseline can give a foundation for a plan of action to move toward Live Traceability.
- **Build the business case for Live Traceability:** With a baseline in hand, systems engineers can offer a case for moving to Live Traceability based on data. The data can establish the ROI, productivity improvements, and risk reduction of moving from static traceability to Live Traceability.
- **Create quick wins:** Once the advantages of Live Traceability are established, the systems engineer can set up continuous syncing between requirements and task management programs, thus automating traceability from requirements to user stories. This simple shift

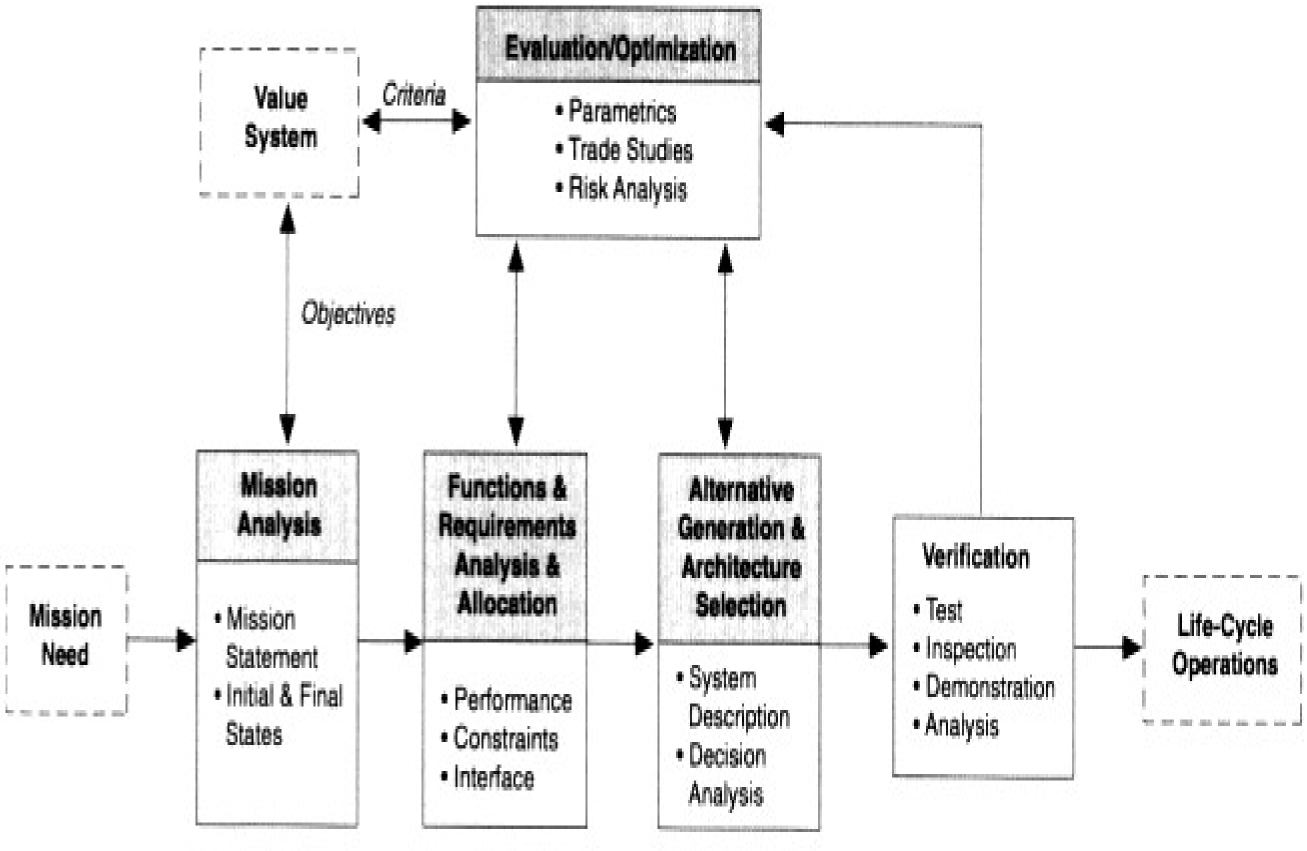
can help demonstrate the value of shifting from after-the-fact traceability to Live Traceability.

## What is the Systems Engineering Process?

The systems engineering process can take a top-down approach, bottoms up, or middle out depending on the system being developed. The process encompasses all creative, manual, and technical activities necessary to define the ultimate outcomes and see that the development process results in a product that meets objectives.

The process typically has four basic steps:

- **Task definition/analysis/conceptual:** In this step, the systems engineer works with stakeholders to understand their needs and constraints. This stage could be considered a creative or idea stage where brainstorming takes place and market analysis and end user desires are included.
- **Design/requirements:** In this phase, individual engineers and team members analyze the needs in step 1 and translate them into requirements that describe how the system needs to work. The systems engineer evaluates the systems as a whole and offers feedback to improve integration and overall design.
- **Create traceability:** Although we're listing traceability here as the third step, traceability is actually created throughout the lifecycle of development and is not a discrete activity taking place during one phase. Throughout the lifecycle of development, the team works together to design individual systems that will integrate into one cohesive whole. The systems engineer helps manage traceability and integration of the individual systems.
- **Implementation/market launch:** When everyone has executed their roles properly, the final product is manufactured or launched with the assurance that it will operate as expected in a complex system throughout its anticipated life cycle.

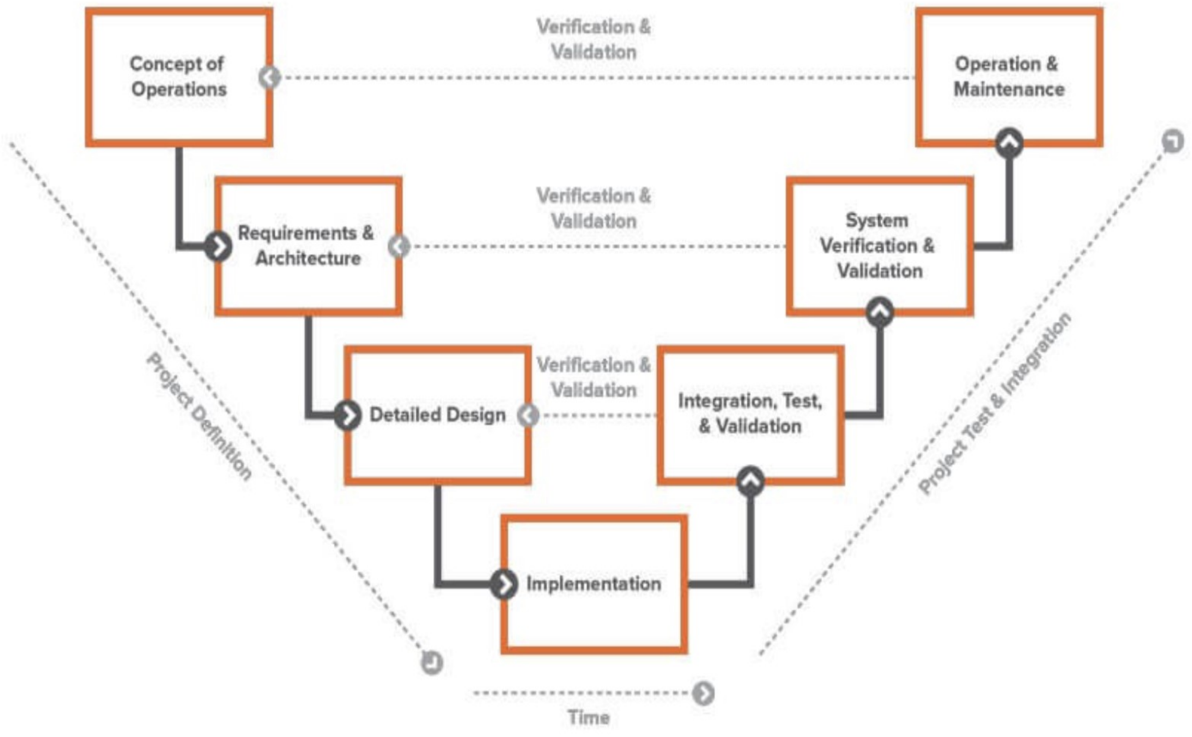


## The "V" Diagram of Systems Engineering

Developed in the 1980s, the "V" Diagram of Systems Engineering is a way of specifying the specific series of steps that make up a systems engineering approach. While it was originally employed in a pre-Agile environment, it still has relevance to product development today and can enable faster, less risky product development.

The "V" diagram allows system engineers multiple viewpoints and opportunities to evaluate systems as they integrate with each other. This approach starts with the desired outcomes and objectives and then deconstructs them into individual systems and system components for the purpose of design. Once the requirements and design details are established, individual systems can be tested and evaluated, then integrated into the overall piece for testing and verification. As the systems are integrated and become closer to the final complex system, teams have multiple opportunities to validate and verify concepts, requirements, and design.

For the systems engineer, the "V" Model can give a clear roadmap that allows the breakdown of the complex system into smaller parts and then the reintegration and reassembly of the pieces into a cohesive whole. With systems broken down to individual components, traceability, requirements management, and testing and validation become more manageable. In addition, as the pieces are reintegrated into the whole system, the "V" Model allows for an iterative process that gives a clearer view into potential risks and helps troubleshoot problems.


Concept of Operations
Verification & Validation
Operation & Maintenance
Requirements & Architecture
Verification & Validation
System Verification & Validation
Detailed Design
Verification & Validation
Integration, Test, & Validation
Implementation
Project Definition
Project Test & Integration
Time

Systems engineering is a discipline that's vital to the success of a complex system. By including systems engineers in all stages of product development and requirements management, teams can reduce risks, improve time to market, and produce better products that more adequately meet end user requirements.