CpSc 875

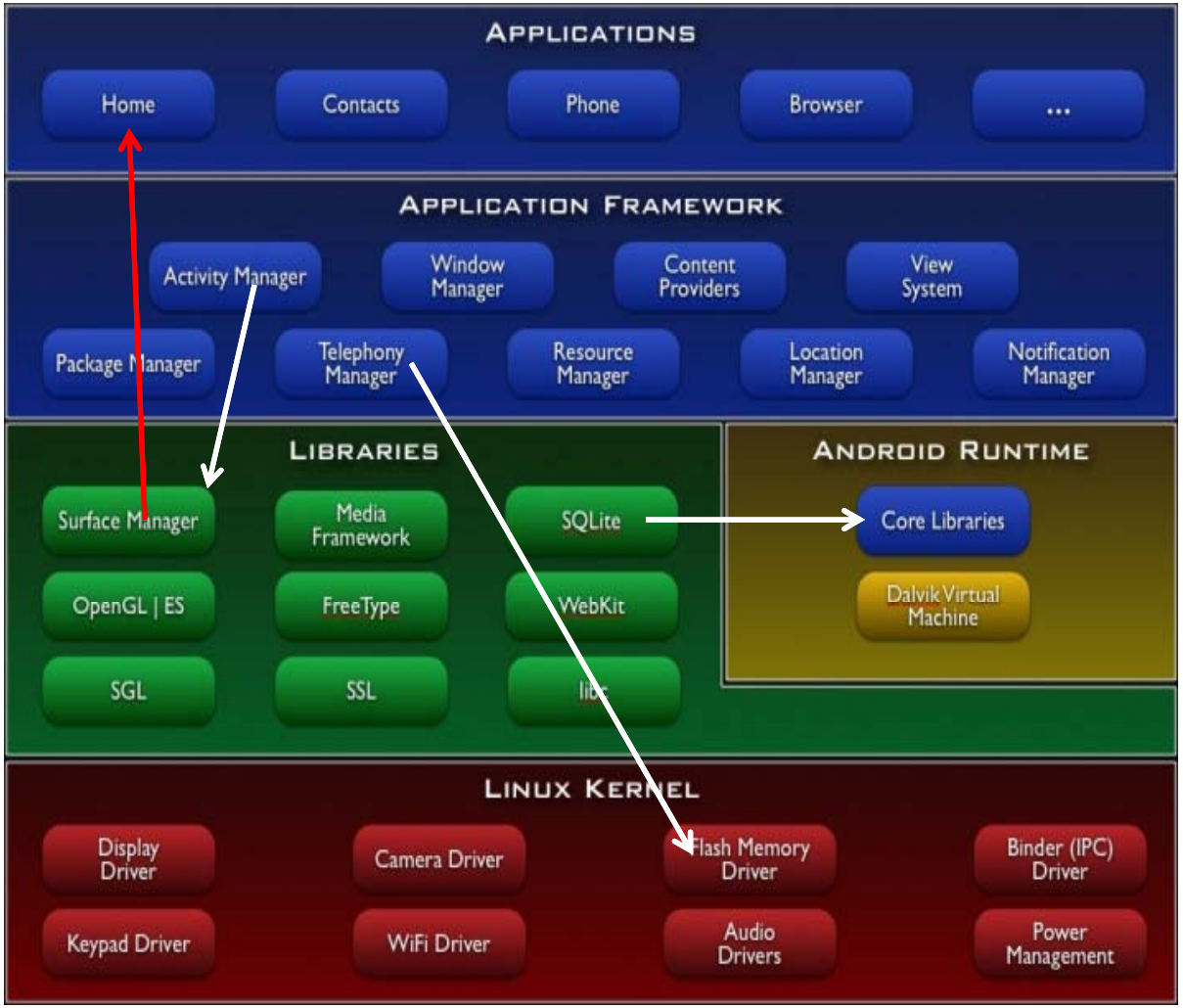
John D. McGregor

C 2 – ADD

Attribute Driven Design

Example - Android architecture

- The primary structure here is “layer.”
- Code in a layer may only “know about” code in an adjacent layer.
- Usually a “higher” layer may only address “lower” layers.
- Within a layer things change at about the same rate.
- The “higher” layers change at a faster rate than the “lower” layers.



This slide set is based on:

Attribute-Driven Design (ADD), Version 2.0

By Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, and Bill Wood

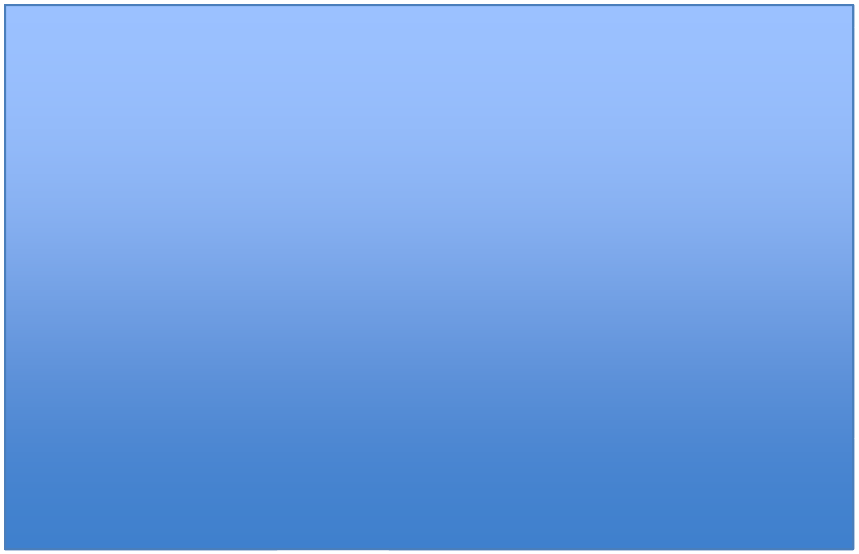
An SEI Technical Report: CMU/SEI-2006-TR-023

Functional/non-functional

- If all we cared about were functional requirements then no structure would be needed. The box below would be the product and the architecture.

Modularity

- Dividing the box into two pieces results in more modularity and a more maintainable system but it may result in a slower system depending on how the two pieces communicate. ADD gives a technique for knowing what attributes are important.



One architectural transformation

- It slices, it dices, and so much more

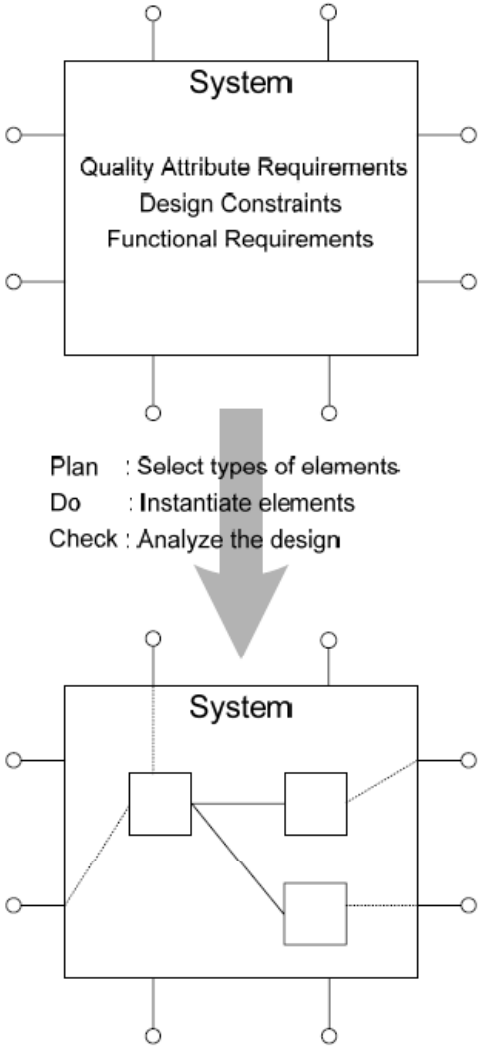
http://www.heartlandamerica.com/browse/item.asp?PIN=124145&SC=WIF20001&

Transformation

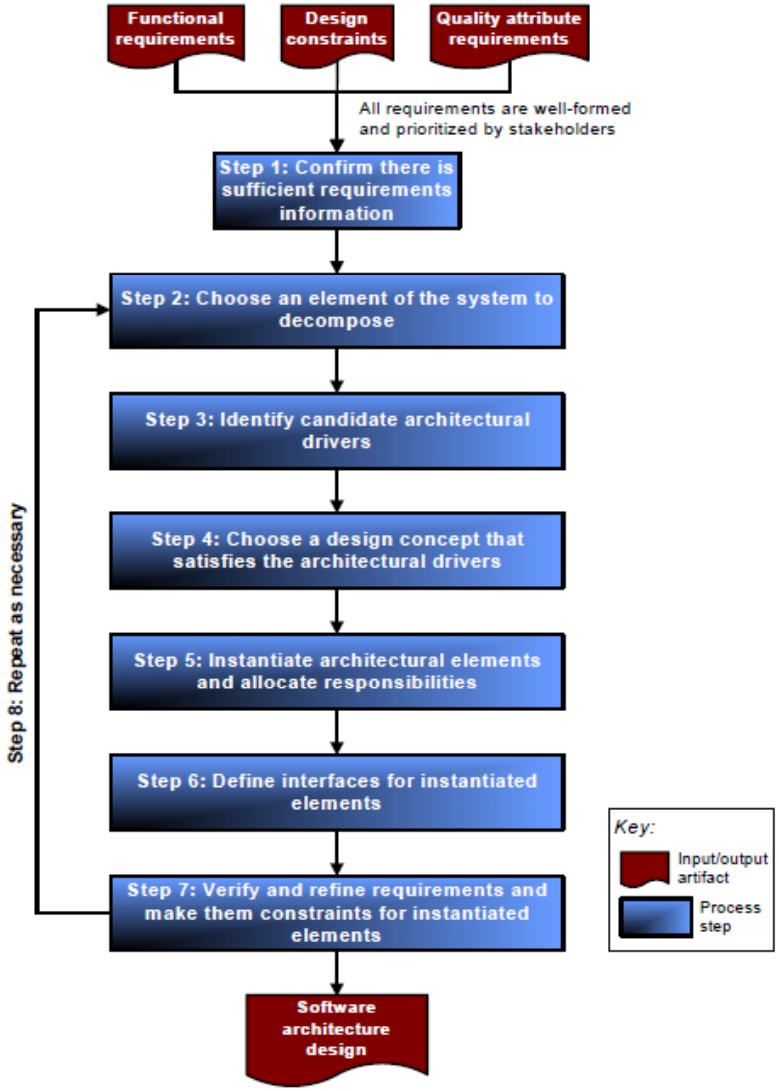
- One form is transformed into another form by applying an operator
- “decompose” is an operator
- A transformation makes specific changes to specific attributes of a form but leaves other attributes unchanged.
- Decompose does not destroy anything it simply repackages it.

ADD Overview

- Plan – what to do to the architecture
- Do – transform the architecture
- Check – evaluate to see if the results are what you expected



Steps


Functional requirements
Design constraints
Quality attribute requirements
All requirements are well-formed and prioritized by stakeholders
Step 1: Confirm there is sufficient requirements information
Step 2: Choose an element of the system to decompose
Step 3: Identify candidate architectural drivers
Step 4: Choose a design concept that satisfies the architectural drivers
Step 5: Instantiate architectural elements and allocate responsibilities
Step 6: Define interfaces for instantiated elements
Step 7: Verify and refine requirements and make them constraints for instantiated elements
Step 8: Repeat as necessary
Software architecture design
Key:
Input/output artifact
Process step

Inputs to ADD

- Inputs to ADD are functional requirements, design constraints, and quality attribute requirements that system stakeholders have prioritized according to business and mission goals.
- The architect leads the effort to prioritize quality attributes, many of which contradict each other.

Outputs from ADD

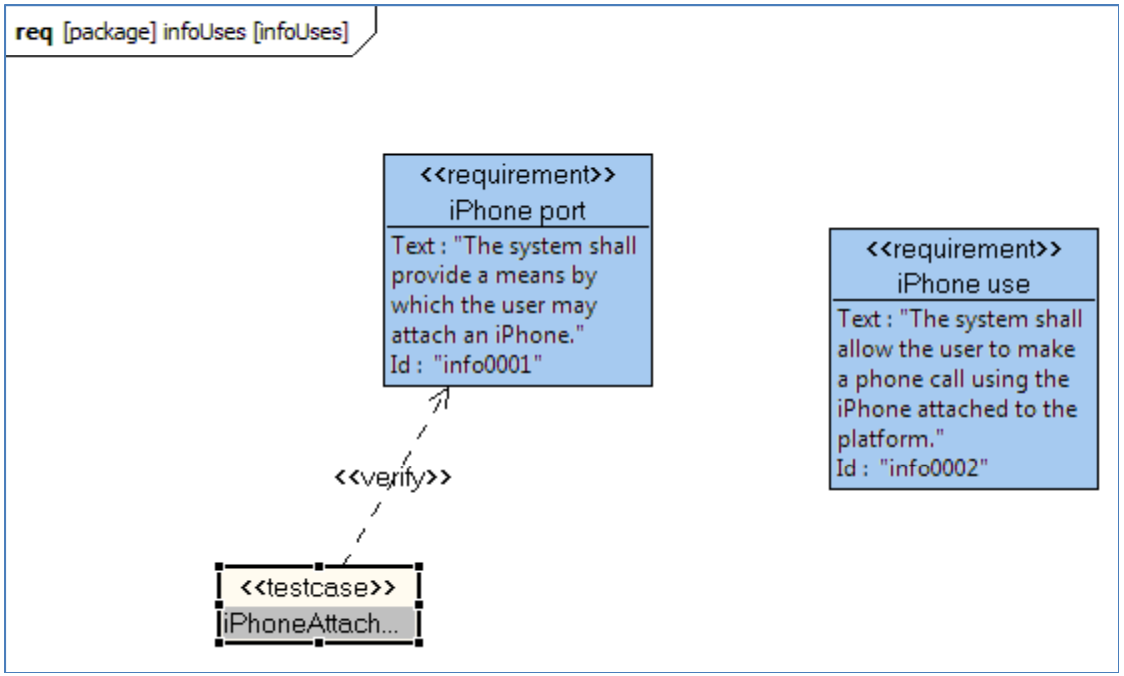
- The output of ADD is a system design in terms of the roles, responsibilities, properties, and relationships among software elements. The following terms are used throughout this document:
- software element: a computational or developmental artifact that fulfills various roles and responsibilities, has defined properties, and relates to other software elements to compose the architecture of a system [Bass 03]
- role: a set of related responsibilities [Wirfs-Brock 03]
- responsibility: the functionality, data, or information that a software element provides
- property: additional information about a software element such as name, type, quality attribute characteristic, protocol, and so on [Clements 03]
- relationship: a definition of how two software elements are associated with or interact with one another

Step 1: Confirm There Is Sufficient Requirements Information

- make sure that the system's stakeholders have prioritized the requirements according to business and mission goals.
  - Later we will see a way to do this during architecture definition
- confirm that there is sufficient information about the quality attribute requirements to proceed

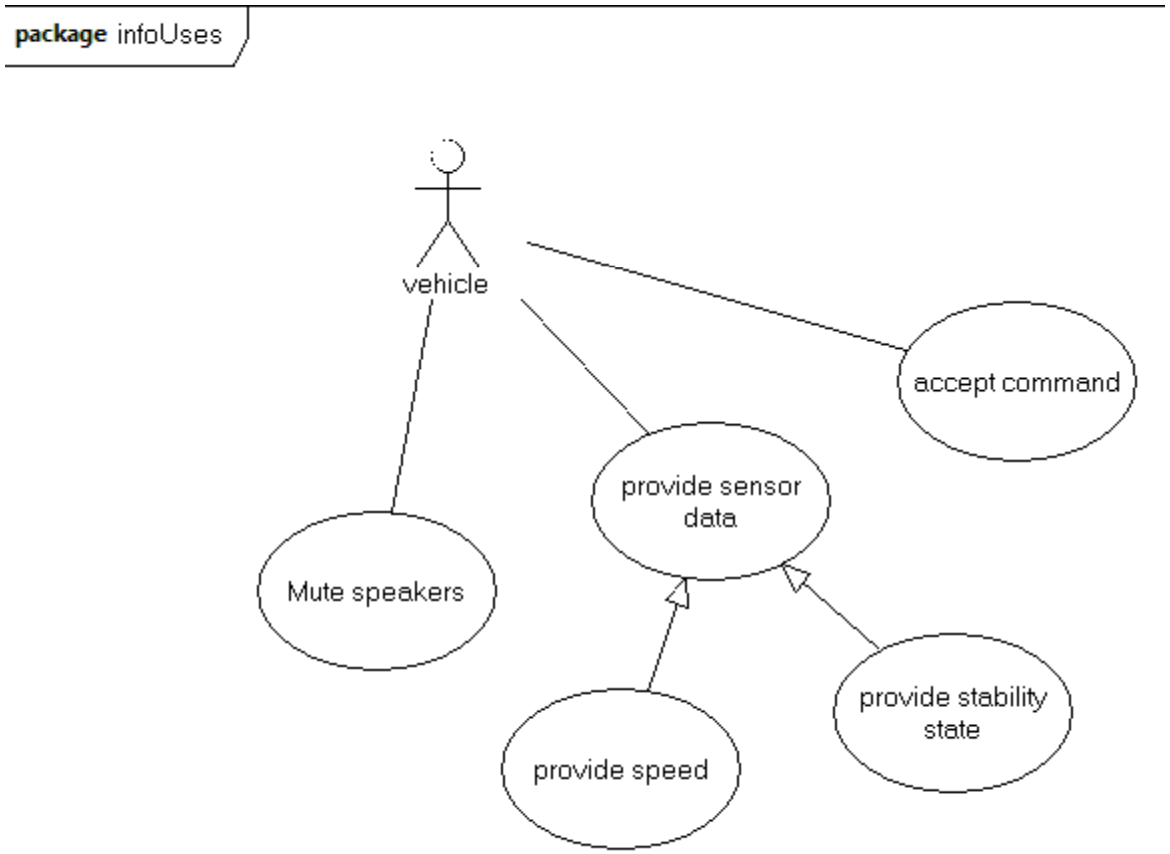
Requirements

- The model describes what the system must do from an abstract perspective.



Use cases

- Use cases restate requirements from the user's perspective.
- A use case has a many-to-many relationship with requirements



Scenarios

- Each use case has several related scenarios
  - Normal (sunny day) scenarios
  - Error (rainy day) scenarios
  - Exceptional scenarios
- A scenario is a dialog

The actor:	The system responds by:

Example

The actor:	The system responds by:
Inserts a DVD into the slot	Beginning to show the video on the currently selected screens
Adjusts the volume on the console screen	Adjusting the output of the speaker system

C^3

- The requirements model should be:
  - Complete
    - Nothing could be added
  - Correct
    - Corresponds to expert judgment
  - Consistent
    - No contradictions
- But it will not be those things immediately

Who determines priorities

- Business goals – set by a dictator or by a consensus building process set a high-level direction
- Stakeholders
  - Users
  - Customers
  - Suppliers
  - Developers
  - Testers
  - Etc.

What do we use to determine priorities

- The Scenarios from the use cases can be used
- In most cases there are so many that only the sunny day scenarios are used
- The discussion during this exercise often results in rewordings, refactorings, adding and deleting use cases

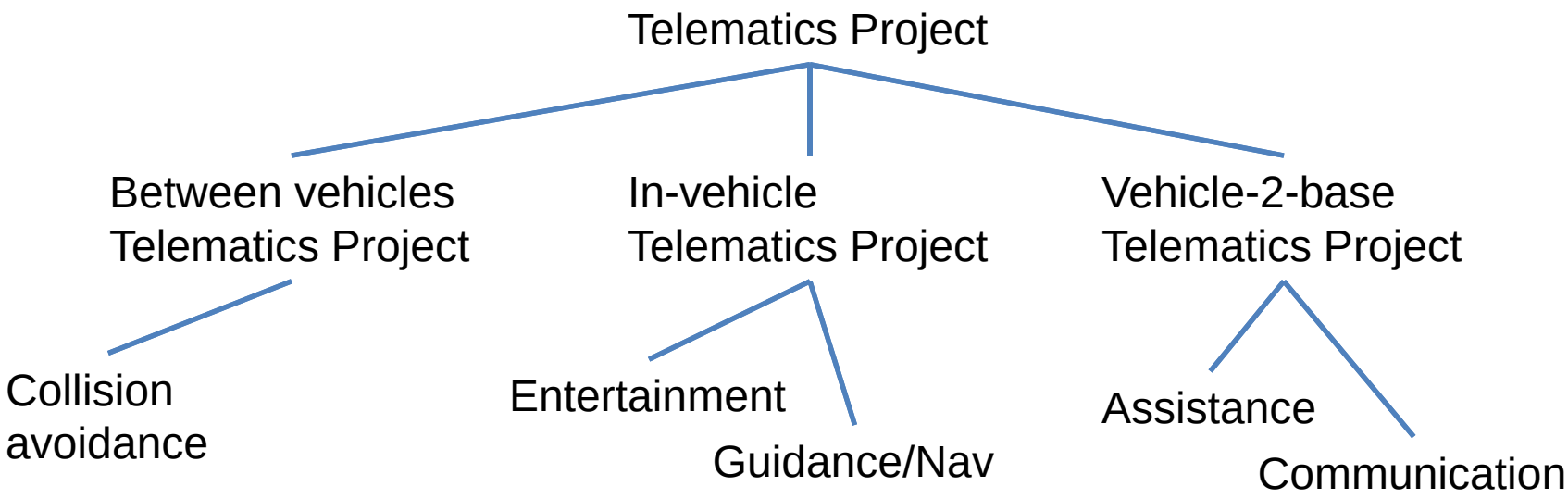
How do we determine priorities

- Decide on the relative importance of each stakeholder
- Hold a stakeholder meeting
- Assign a number of votes to each stakeholder that reflects their importance
- Allow each stakeholder to assign their votes to the use cases
- Tally to find ordering

Driving requirements

- The highest vote getters are the driving requirements.
- Often these are the only ones we will have time to consider as we make design decisions.

Possible projects

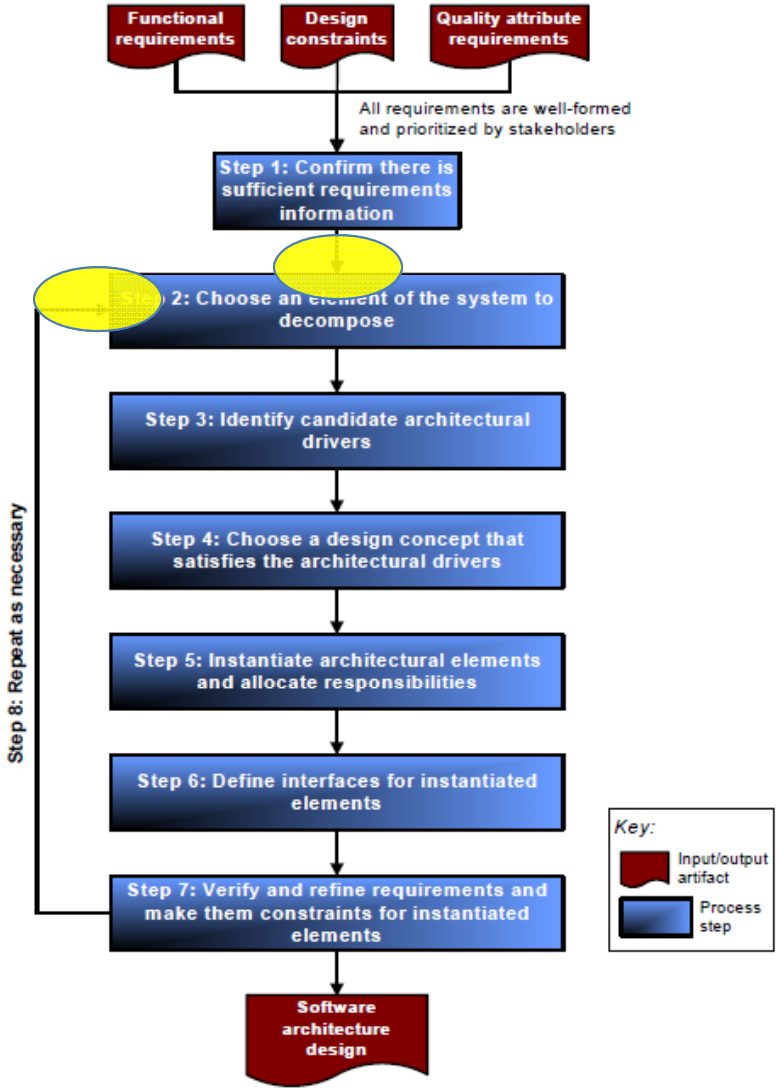


Next steps

- For now, a deeper look at the domain for the example.
- Read the sources found on the domain slide from the first class.
- Find at least two more resources.
- Create at least 15 use cases – use Topcased.
- In a group of four identify two people per subteam. At some point switch models between sub-teams, identify redundancies and gaps, create a single model.
- Take a screen shot of the model and submit a picture.
- Due Monday January 21th by 11:59pm via email to johnmc@cs.clemson.edu. Subject line: <last name>Assign1

7 day rhythm for this course

2 paths


Functional requirements
Design constraints
Quality attribute requirements
All requirements are well-formed and prioritized by stakeholders
Step 1: Confirm there is sufficient requirements information
Step 2: Choose an element of the system to decompose
Step 3: Identify candidate architectural drivers
Step 4: Choose a design concept that satisfies the architectural drivers
Step 5: Instantiate architectural elements and allocate responsibilities
Step 6: Define interfaces for instantiated elements
Step 7: Verify and refine requirements and make them constraints for instantiated elements
Step 8: Repeat as necessary
Software architecture design
Key:
Input/output artifact
Process step

Step 2: Choose an Element of the System to Decompose

1. current knowledge of the architecture
   - if it is the only element you can choose (e.g., the entire system or the last element left)
   - the number of dependencies it has with other elements of the system (e.g., many or few dependencies)
2. risk and difficulty
   - how difficult it will be to achieve the element's associated requirements
   - how familiar you are with how to achieve the element's associated requirements
   - the risk involved with achieving the element's associated requirements
3. business criteria
   - the role the element plays in incremental development of the system
   - the role it plays in incremental releases of functionality (i.e., subsetability)
   - whether it will be built, purchased, licensed, or used as open source
   - the impact it has on time to market
   - whether it will be implemented using legacy components
   - the availability of personnel to address a component
4. organizational criteria
   - the impact it has on resource utilization (e.g., human and computing re-sources)
   - the skill level involved with its development
   - the impact it has on improving development skills in the organization
   - someone of authority selected it

Step 3: Identify Candidate Architectural Drivers

- We have an element to decompose
- We evaluate the requirements that apply to this element
- They are ranked (high,medium,low) on two criteria
  - First, how important is the requirement to the success of the system
  - Second, how much of an impact will this requirement have on the structure of the architecture
- The requirements that are ranked (H,H) are referred to as “architecture drivers”

Step 4: Choose a Design Concept That Satisfies the Architectural Drivers

1. Identify the design concerns that are associated with the candidate architectural drivers. For example, for a quality attribute requirement regarding avail-ability, the major design concerns might be fault prevention, fault detection, and fault recovery [Bass 03].
2. For each design concern, create a list of alternative patterns that address the concern.6 Patterns on the list are derived from

   − your knowledge, skills, and experience about which patterns might be appropriate

   − known architectural tactics for achieving quality attributes [Bass 03] If a candidate architectural driver concerns more than one quality attribute, multiple tactics may apply.

   − other sources such as books, papers, conference materials, search engines, commercial products, and so forth

   For each pattern on your list, you should

   - a. identify each pattern's discriminating parameters to help you choose among the patterns and tactics in the list. For example, in any restart pattern (e.g., warm restart, cold restart), the amount of time it takes for a restart is a discriminating parameter. For patterns used to achieve modifiability (e.g., layering), a discriminating parameter is the number of dependencies that exist between elements in the pattern.

   - b. estimate the values of the discriminating parameters

3. Select patterns from the list that you feel are most appropriate for satisfying the candidate architectural drivers. Record the rationale for your selections.7 To decide which patterns are appropriate

a. Create a matrix (as illustrated in Table 1) with patterns across the top and the candidate architectural drivers listed on the left-hand side. Use the matrix to analyze the advantages/disadvantages of applying each pattern to each candidate architectural driver. Consider the following:

− What tradeoffs are expected when using each pattern?

− How well do the patterns combine with each other?

− Are any patterns mutually exclusive (i.e., you can use either pattern A or pattern B but not both)?

b. Choose patterns that together come closest to satisfying the architectural drivers.

4. Consider the patterns identified so far and decide how they relate to each other. The combination of the selected patterns results in a new pattern.

   a. Decide how the types of elements from the various patterns are related.8

   b. Decide which types of elements from the various patterns are not related.

   c. Look for overlapping functionality and use it as an indicator for how to combine patterns.

   d. Identify new element types that emerge as a result of combining patterns.

   e. Review the list of design decisions at the end of this section and confirm that you have made all the relevant decisions.

5. Describe the patterns you've selected by starting to capture different architectural views, such as Module, Component-and-Connector, and Allocation views. You don't need to create fully documented architectural views at this point. Document any information that you are confident in or that you need to have to reason about the architecture (including what you know about the properties of the various element types). Ideally, you should use view tem-plates to capture this information [Clements 03].

6. Evaluate and resolve inconsistencies in the design concept:

a. Evaluate the design against the architectural drivers. If necessary, use models, experiments, simulations, formal analysis, and architecture evaluation methods.

b. Determine if there are any architectural drivers that were not considered.

c. Evaluate alternative patterns or apply additional tactics, if the design does not satisfy the architectural drivers.

d. Evaluate the design of the current element against the design of other elements in the architecture and resolve any inconsistencies. For example, while designing the current element, you may discover certain properties that must be propagated to other elements in the architecture.

Design decisions

- You have decided on an overall design concept that includes the major types of elements that will appear in the architecture and the types of relationships among them.
- You have identified some of the functionality associated with the different types of elements (e.g., elements that ping in a Ping-Echo pattern will have ping functionality).
- You know how and when particular types of software elements map to one another (i.e., either statically or dynamically).
- You have thought out communication among the various types of elements (both internal software elements and external entities) but perhaps deferred decisions about it.
- You have reasoned about software elements and system resources but perhaps deferred decisions about them.

Design decisions - 2

- You have thought out dependencies between the various types of internal software elements but perhaps deferred decisions about them.

- You have also considered and perhaps deferred decisions about the following:
  - the abstraction mechanisms used
  - what system elements know about time
  - what process/thread model(s) will be employed
  - how quality attribute requirements will be addressed

Step 5: Instantiate Architectural Elements and Allocate Responsibilities

1. Instantiate one instance of every type of element you chose in Step 4. These instances are referred to as "children" or "child elements" of the element you are currently decomposing (i.e., the parent element).
2. Assign responsibilities to child elements according to their type. For example, ping-type elements are assigned responsibilities including ping functionality, ping frequency, data content of ping signals, and the elements to which they send ping signals.
3. Allocate responsibilities associated with the parent element among its children according to the rationale and element properties recorded in Step 4. For ex-ample, if a parent element in a banking system is responsible for managing cash distribution, cash collection, and transaction records, then allocate those responsibilities among its children. Note that all responsibilities assigned to the parent are considered at this time regardless of whether they are architecturally significant. At this point, it may be useful to consider use cases that systems typically address—regardless of whether they were given explicitly as requirements. This exercise might reveal new responsibilities (e.g., resource management). In addition, you might discover new element types and wish to create new in-stances of them. These use cases include
   - one user doing two tasks simultaneously
   - two users doing similar tasks simultaneously
   - startup
   - shutdown
   - disconnected operation
   - failure of various elements (e.g., the network, processor, process)

4. Create additional instances of element types in these two circumstances:

    a. A difference exists in the quality attribute properties of the responsibilities assigned to an element. For example, if a child element is responsible for collecting sensor data in real time and transmitting a summary of that data at a later time, performance requirements associated with data collection may prompt us to instantiate a new element to handle data collection while the original element handles transmitting a summary.

    b. You want to achieve other quality attribute requirements; for example, you reassign the functionality of one element to two elements to promote modifiability.

    At this point, you should review the list of design decisions listed at the end of this section and confirm that you made all the relevant decisions.

5. Analyze and document the design decisions you have made during step 5 using various views such as these three:

    a. Module views are useful for reasoning about and documenting the non-runtime properties of a system (e.g., modifiability).

    b. Component-and-Connector views are useful for reasoning about and documenting the runtime behaviors and properties of a system (e.g., how elements will interact with each other at runtime to meet various requirements and what performance characteristics those elements should exhibit).

    c. Allocation views are useful for reasoning about the relationships between software and non-software (e.g., how software elements will be allocated to hardware elements).

Design decisions

- how many of each type of element will be instantiated and what individual properties and structural relations they will possess
- what computational elements will be used to support the various categories of system use
- what elements will support the major modes of operation
- how quality attribute requirements have been satisfied within the infrastructure and applications
- how functionality is divided and assigned to software elements including how functionality is allocated across the infrastructure and applications

- how software elements map to each other

− how system elements in different architectural structures map to each other (e.g., how modules map to runtime elements and how runtime elements map to processors)

− whether the mapping of one system element to another is static or dynamic (i.e., when the mapping is determined—at build time, deployment, load time, or runtime)

• communication among the various elements, both internal software elements and external entities

− which software elements need to communicate with each other

− what mechanisms and protocols will be used for communication between software elements and external entities

− the required properties of mechanisms that will be used for communication between software elements and external entities (e.g., synchronous, asynchronous, hybrid coupling)

− the quality attribute requirements associated with the communication mechanisms

− the data models on which communication depends

− what computational elements support the various categories of system use

− how legacy and COTS components will be integrated into the design

- internal software elements and system resources
  - what resources are required by software elements
  - what resources need to be managed
  - the resource limits
  - how resources will be managed
  - what scheduling strategies will be employed
  - what elements are stateful/stateless
  - the major modes of operation
- dependencies between the internal software elements
  - what execution dependencies exist among elements
  - how and where execution dependencies among elements are resolved
  - the activation and deactivation dependencies among software elements
- which abstraction mechanisms are used
- how much system elements know about time
- what process/thread model(s) will be employed
- how quality attribute requirements will be addressed

Step 6: Define Interfaces for Instantiated Elements

- Define the services and properties required and provided by the software elements in our design.
- These services and properties are referred to as the element's interface.
- Interfaces describe the PROVIDES and REQUIRES assumptions that software elements make about one another. An interface might include any of the following:
  - syntax of operations (e.g., signature)
  - semantics of operations (e.g., description, pre- and post-conditions, restrictions)
  - information exchanged (e.g., events signaled, global data)
  - quality attribute requirements of individual elements or operations
  - error handling

- Exercise the functional requirements that involve the elements you instantiated in Step 5.
- Observe any information that is produced by one element and consumed by another. Consider the interfaces from the perspective of different views. For example, a Module view will allow you to reason about information flow; a Concurrency view will allow you to reason about performance and availability; and a Deployment view will allow you to reason about security and availability.
- Record your findings in the interface documentation for each element.

Design decisions

- the external interfaces to the system
- the interfaces between high-level system partitions
- the interfaces between applications within high-level system partitions
- the interfaces to the infrastructure

Step 7: Verify and Refine Requirements and Make Them Constraints for Instantiated Elements

- 1. Verify that all functional requirements, quality attribute requirements, and design constraints assigned to the parent element have been allocated to one or more child elements in the decomposition.
- 2. Translate any responsibilities that were assigned to child elements into functional requirements for the individual elements.
- 3. Refine quality attribute requirements for individual child elements as necessary.

Step 8: Repeat Steps 2 through 7 for the Next Element of the System You Wish to Decompose

- Once you have completed Steps 1–7, you have a decomposition of the parent element into child elements. Each child element is a collection of responsibilities, each having an interface description, functional requirements, quality attribute requirements, and design constraints. You can now return to the decomposition process in Step 2 where you select the next element to decompose.

Next steps

- We will explore these steps in greater detail through out the semester

Telematics System quality attributes

- Scalability/usability
- Availability
- Performance
  - Latency
  - Throughput
- Interoperability
- Extensibility
- …