Designing Software Architecture to Achieve Business Goals

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Len Bass

Software Engineering Institute | Carnegie Mellon

Sponsored by the
U.S. Department of Defense

Business goals

Truism: Software systems are constructed to satisfy business goals.

Question 1: Why does the software architect need to know business goals?

Question 2: How does the software architect determine the business goals for a system?

Question 3: Where in your curriculum is this material taught?

Why does an architect need to know business goals?

Software design is driven by quality attribute requirements. If software design is only driven by function, then a monolithic system would suffice. But we routinely see

- Redundancy to improve availability
- Layers to improve portability
- Caching to improve performance
- …

Quality attribute requirements reflect business goals. Otherwise why does the requirement exist?

For example, response time requirements might come from

- Differentiating the product from its competition
- Response time makes the soldier a more effective warfighter
- Accurate response time makes the engine run efficiently which leads to customer satisfaction and more sales
- …

The architect needs to know business goals because they lead to quality attribute requirements that, in turn, lead to design choices.

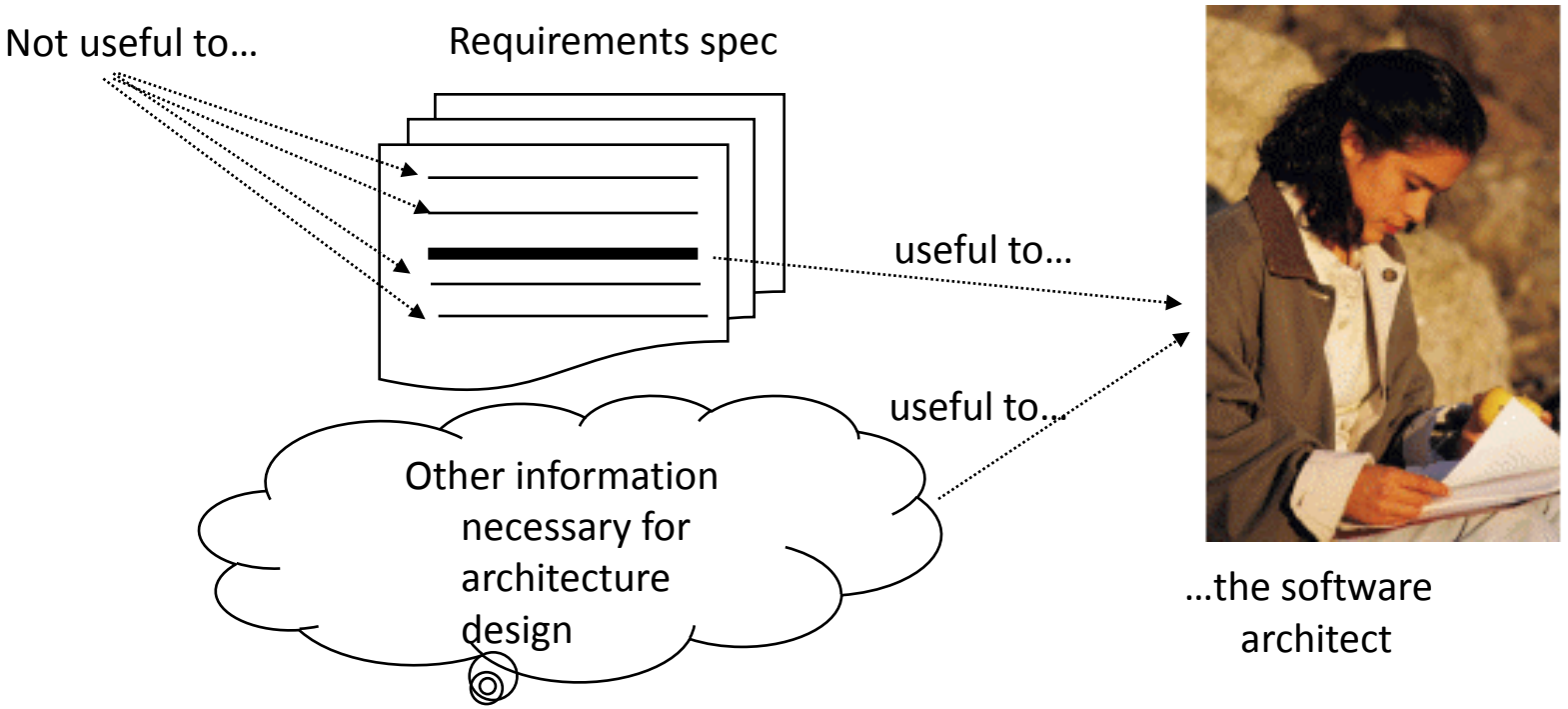
What does this knowledge let the architect do?

Knowledge of business goals enables an architect

- To make informed tradeoffs. Should performance be sacrificed to improve modifiability? It depends on the business goals for the system.
- To intelligently adjust requirements – e.g. a requirement may state that response time should be .1 sec but the goal may be to match a competitor's response time. If the competitor speeds up their response time during development, the architect can know the necessity for changing the requirement.
- To push back on unreasonable requirements. Tight performance requirements may make a system very expensive to build and may not be justified. Knowing the business justification for a requirement enables the architect to push back on unreasonable requirements.

How does the software architect determine the business goals for a system?

Despite what classical software engineering teaches, *architects need little of what is in a requirements specification, and requirements specifications contain little of the information needed by an architect.*



Architectural requirements are missing from typical requirements specifications

Architectural requirements are the requirements that drive the design of the architecture.

- Quality attribute requirements
- Business requirements for the developing organization

Quality attribute requirements are, typically, not well specified.

- The system shall be modular
- The system shall be secure

Business requirements for the developing organization are not specified at all.

- The developing organization wishes to sell the system internationally
- The developing organization wishes to protect IP from sub-contractors
- The developing organization wishes to reuse a particular framework.

Business goals for organizations involved in constructing a system

There are multiple organizations involved in most system developments

- Acquiring organization.
  - System may be for internal use and not seen outside the acquiring organization – e.g. CMU travel system
  - System may be for external use by and seen outside acquiring organization – e.g. tele-banking system
  - System may be for sale – e.g. Office
- Developing organization(s)
  - May be within business unit of acquiring organization
  - May be within acquiring organization but within different business unit from acquiring organization
  - May be sub-contractors to the developing organization, e.g. outsourced portions of the system

Business goals specific to a system

Each organization has its own business goals for the system under development

Ideally, the system will satisfy the union of all of the business goals

It is the responsibility of the architect to design the system to satisfy these goals.

What follows is an approach to gathering the business goals for a system.

How do we characterize business goals for a system?

We characterize the business goals in two fashions

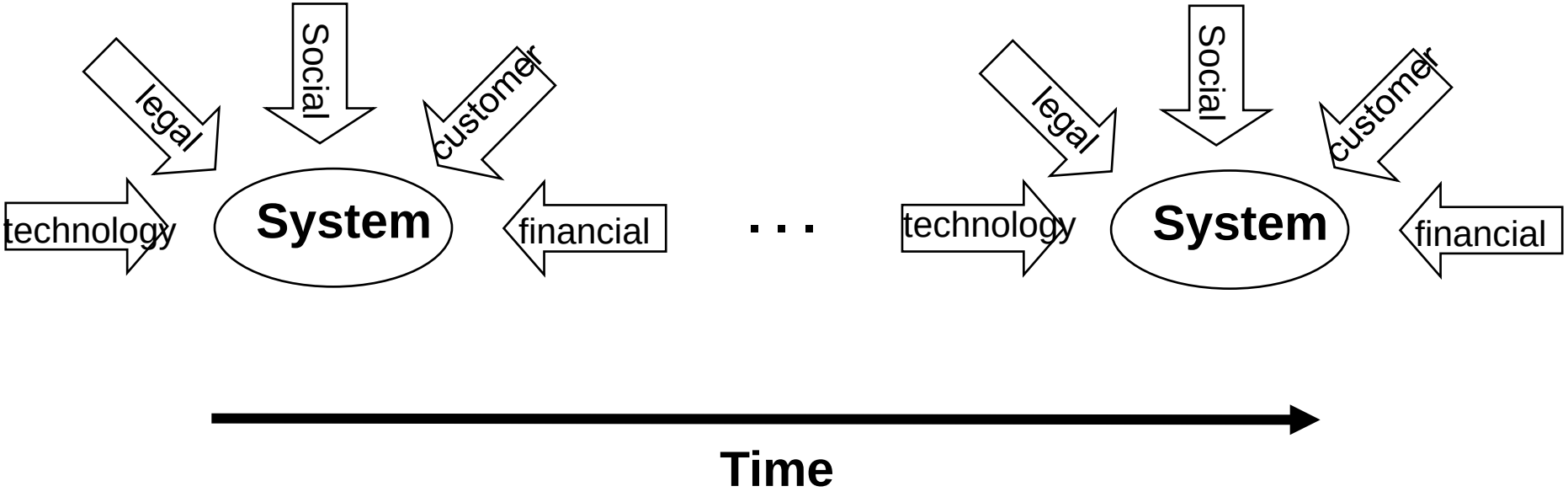
1. Goals and how they change
    - Canonical business goal categories
    - Forces acting on a system over time

2. Source of the goals
    - Pedigree

Categories of Business Goals – taken from a survey of the business goal literature

1. Growth and continuity of the organization
2. Meeting financial objectives
3. Meeting personal objectives
4. Meeting responsibility towards employees
5. Meeting responsibility towards society
6. Meeting responsibility to country
7. Meeting responsibility towards shareholders
8. Managing market position
9. Improving business processes
10. Managing quality and reputation of products

Forces acting on the system over time



Comments on the business goal categories and forces

They are categories and forces not goals – as such they are starting points for a conversation.

Our experience using these categories and forces is that they generate far reaching conversations, e.g. the product manager viewed a system as the first element of a product line but the architect was unaware of that perspective.

Business goal scenario (pedigree)

We have a syntax to describe a business goal scenario

“Who?” defines the stakeholder the goal is serving or affecting.

“What?” describes the goal and its benefits and negative influences that affect each stakeholder.

“When?” captures the timing of goals’ effects on stakeholders.

“Where?” identifies the location for delivering benefits and other impacts.

“Why?” gives the rationale for providing the stakeholder benefits you deliver.

“How?” explains your method of providing your products and services expressed in the goal and being compensated for them.

“Value” expresses the worth of the goal to the organization.

Eliciting quality attribute requirements

For each business goal, determine how various quality attributes contribute to its achievement.

E.g. what contribution does performance make to expanding market share?

This leads to quality attribute requirements and ties these requirements to business goals.

In essence, fill out this matrix

Generic business goal	System business goal Considering how forces change over time		Security contribution ….		Other QA contribution	
Organizational goals	Goal	Pedigree	Goal	Pedigree	Goal	Pedigree
Financial goals	Goal	Pedigree	Goal	Pedigree	Goal	Pedigree
⋮	Goal	Pedigree	Goal	Pedigree	Goal	Pedigree

Using the business goals

Output is high level requirements, not detailed requirements. Determining the business goals and forces that act on them is not a substitute for normal requirements analysis.

Emphasis on business goals allows architect to make tradeoff decisions.

- Trading off one quality attribute against another
- Trading off cost for a goal

There is some repetition on the goals in the different categories.

Emphasis is on empowering architect, not on providing specific requirements for the architect to achieve.

How are business goals covered in your curriculum?

Systems as a means of satisfying business goals?

Various organizational stakeholders and their different perspectives?

Eliciting business goals?

Relating business goals to quality attribute requirements?

Moving to design

Architecturally significant requirements

Design as generate and test

Architecturally Significant Requirements - 1

Architecturally significant requirements (ASRs) are the requirements that impact the structure of the design and should be the primary focus when doing architectural analysis.

The ASR concept derives from our experience with ATAM (Architecture Tradeoff Analysis Method). ATAM uses architecture description from “30,000 ft” level. This perspective enables an understanding of what drove the architect to create the design being evaluated.

Architecturally Significant Requirements – 2

When creating an architecture, the goal is to determine what those “driving” requirements are.

RUP refers to Architecturally Significant Use Cases (same concept)

Recall that quality attribute requirements are the ones that drive the design => Architecturally significant requirements are quality attribute requirements.

Utility Tree - 1

Quality attribute utility trees provide a mechanism for translating the business drivers of a system into concrete quality attribute scenarios.

A utility tree lists

- The quality attributes for the particular system being designed as one level of the tree.
- The quality attribute “concerns” as the next level.
- Quality attribute scenarios are the leaves of the tree

The utility tree at the leaves serves to make concrete the quality attribute requirements, forcing architect and customer representatives to define relevant quality attributes precisely.

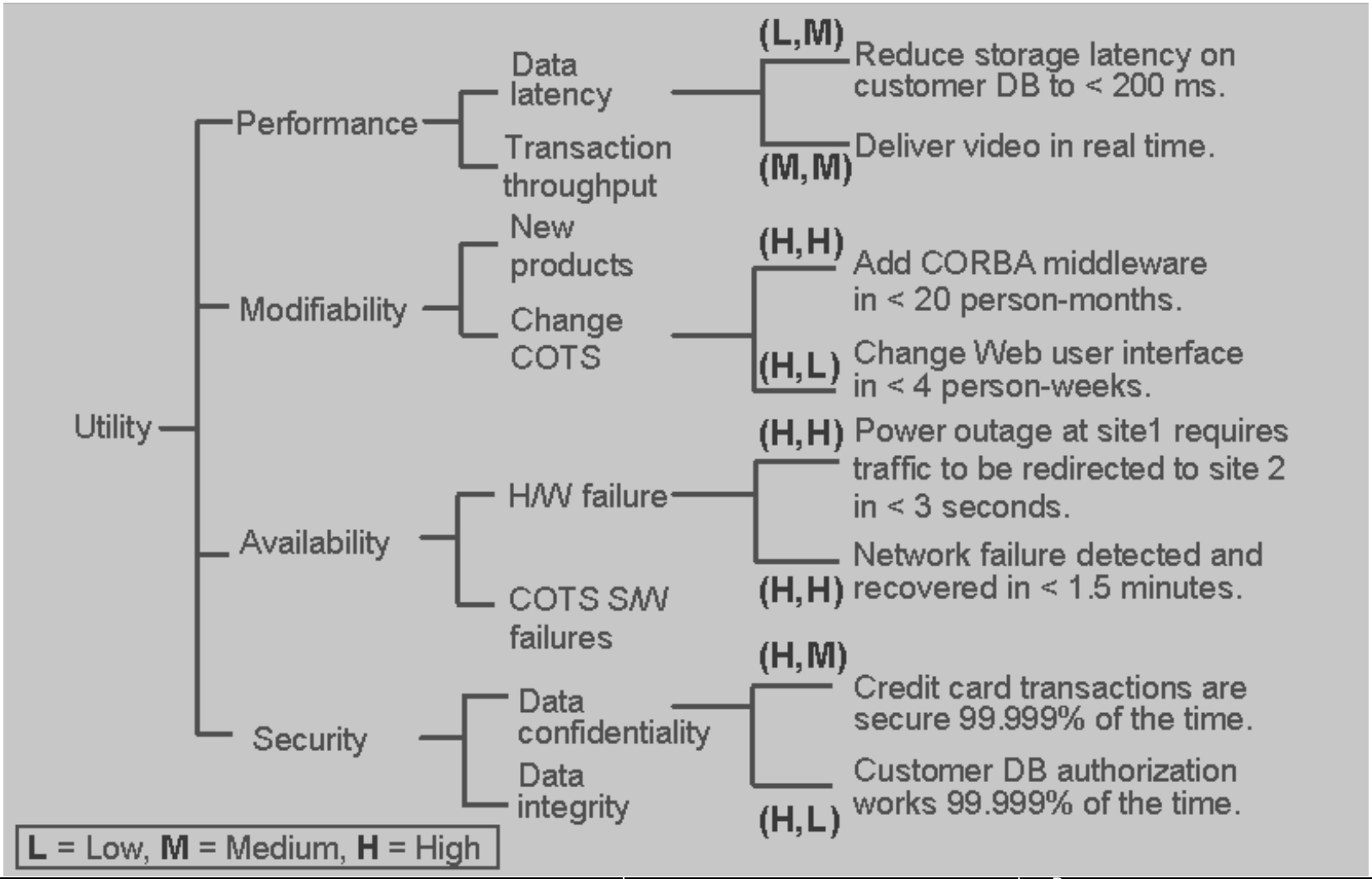
Utility Tree - 2

The leaves are prioritized in two dimensions

- The importance to the business of the scenario (H, M, L)
- The pervasiveness within the architecture of the requirements (H, M, L)

Those scenarios rated high importance and high difficulty provide the most critical context against which the architecture can be analyzed. These scenarios are candidates for the ASRs.

Utility Tree - 3



Quality Attribute Data from SEI ATAMs[1]

	QAs	
	QA	Dist.
1	Modifiability	14.1%
2	Performance	13.6%
3	Usability	11.4%
4	Maintainability	8.5%
5	Interoperability	7.8%
6	Security	7.3%
7	Configurability	6.9%
8	Availability	6.8%
9	Reliability	5.7%
10	Scalability	3.2%
11	Testability	2.6%
12	Affordability	2.0%
13	Reusability	1.9%
14	Integrability	1.9%
15	Safety	1.1%
16	User data management	1.0%
17	Portability	0.8%
18	Assurance	0.8%
19	Product line	0.8%
20	Net-centric operation	0.5%

QA Concerns		
QA	QA Concern	Dist.
Modifiabillity	new/revised functionality/components	6.4%
Usability	operability (e.g. can do)	4.1%
Modifiability	upgrade/add hardware components	3.9%
Performance	response time/deadline	3.6%
Performance	latency	3.2%
Modifiability	portable to other platforms	3.1%
Interoperability	operate intra-service (e.g. ship-to-ship)	2.8%
Usability	ease of operation: can do within a time limit	2.7%
Performance	throughput	2.1%
Performance	resource utilization	1.9%
Availability	failure recovery/containment	1.9%
Configurability	flexibility (range of operation scenarios)	1.7%
Availability	graceful degradation	1.6%
Interoperability	compliance to standards/protocols	1.5%
Affordability	affordability of various decisions (e.g. opennes)	1.5%
Modifiability	replace COTS	1.4%
Performace	real time	1.4%
Availability	fault tolerance	1.3%
Configurability	discovery (new configuration)	1.3%
Security	authentication	1.3%

[1] Ipek Ozkaya, Len Bass, Raghvinder Sangwan and Robert Nord. Making Practical Use of Quality Attribute Information. IEEE Software, March/April 2008, pp. 25-33.

Design as Generate and Test

Design is the process of

- generating a hypothesis,
- testing that hypothesis,
- generating a new hypothesis, and
- repeating until hypothesized design passes the tests

Several questions result from this view of the design process

- Where does the initial hypothesis come from?
- What does it mean to test a hypothesis?
- Where does the new hypothesis come from?

Initial Design Hypothesis - 1

The initial design hypothesis comes from one of several sources (in order of preference):

1. From similar successful systems to that being built. Successful systems similar to the one being constructed have dealt with most of the issues facing the current system.
2. From a legacy system. If the current system is an extension to a legacy system, then the initial hypothesis comes from the legacy system and the next hypothesis will deal with problems raised through the testing process.

Initial Design Hypothesis - 2

3. A collection of frameworks and pre-existing components. If the system is going to be largely created from frameworks and pre-existing components, then the initial hypothesis consists of these frameworks and components connected with empty connectors. The testing process will determine how the connectors get filled in.
4. A pattern. Multiple patterns exist both in books and on the web. These patterns present solutions to recurring problems. If a pattern exists that can satisfy one of the architecturally significant requirements, then this provides a starting place. Such patterns typically, however, will not address more than one of the architecturally significant requirements. Differrent patterns can be chosen to address each architecturally significant requirement but then the patterns must be composed to get an overall pattern.

Initial Design Hypothesis - 3

5. From first principles of quality attributes–*tactics*
6. From functional/logical view. In this case, the testing will disclose missing quality attribute requirements that need to be addressed.

Output of test stage

We will test the hypothesis with a collection of test cases.

The output of the tests will be

- Additional responsibilities that need to be addressed
- List of quality attribute problems

Source of test cases

- Architecturally significant requirements
- Quality attribute specific use cases
- Architectural design decisions

Testing technique

- Analytic models
- Simulation models
- Scenario walk throughs
- prototypes

Architecturally significant requirements as test cases

Architecturally significant requirements are the ones that the architecture design must satisfy

As such, they are obvious test cases for any design hypothesis.

Quality Attribute Specific Use Cases - 1

There are a collection of quality attribute use cases that should be used as test cases in addition to the architecturally significant requirements.

Based on consideration of functionality:

- expected operation exercising major capabilities
- exceptions
- growth and exploratory scenarios
- deferred binding time
- version upgrades
- modification scenarios

Look for

- Allocation of responsibilities to modules
- Responsibilities associated with exception management
- Responsibilities associated with deferred binding time.

Quality Attribute Specific Use Cases - 2

Based on consideration of parallelism:

- two users doing similar tasks simultaneously
- one user performing multiple activities simultaneously
- start-up (creating threads that must be in waiting mode, initializing connected devices, etc.)
- shutdown (cleaning up similar finishing activities, storing data, etc.)

Look for

- Points of resource contention (synchronization),
- Opportunities for parallelism (creation of new threads)
- Necessity for killing processes (deleting threads)
- Additional responsibilities to manage points of contention and clean up
- Management of user state

Quality Attribute Specific Use Cases - 3

Based on consideration of multiple processors

- installation
- initialization
- processing across processors
- messaging over the network
- disconnected operation
- failure of an element (e.g., process, processor, network)

For each use case

- Determine desired policy
- Determine mechanisms to achieve desired policy
- Determine responsibilities to implement chosen mechanisms

Architectural Design Decisions

Important structures to be used to test a design

- Allocation of Functionality
- Coordination Model
- Data and object Model
- Management of Resources
- Mapping Among Architectural Elements
- Binding Time Decisions
- Choice of Technology

Allocation of Functionality

Allocation of functionality decisions assign responsibilities to software elements. Decisions involving allocation of functionality include

Identifying the important responsibilities and their abstractions, and the operations that they provide.

Determining how these responsibilities are allocated to hardware and software, run-time and non run-time elements (components and modules), e.g. functional decomposition, decomposition based on frame rates, decomposition based on modeling real-world objects, information hiding decomposition, …

Identifying the major modes of operation and determining how they are realized. Examples of major modes include startup, normal processing, overload processing, backup/recovery, degraded operation, etc. They might also be application- or domain-specific, such as takeoff, landing, level flight, etc.

Coordination Model

Software "works" by elements interacting with each other in useful and productive ways through designed mechanisms. Choosing those interaction mechanisms is the task associated with designing the coordination model. Decisions about the coordination model include

- Identifying the elements of the system that need to coordinate—directly or indirectly—and the properties of that coordination, such as timeliness, currency, completeness, correctness, consistency, etc.
- Choosing the coordination model (between systems, between our system and external entities, between elements of our system) and the communication mechanisms that realize this model.
- Understanding the information that system and external entities share and how consistent this information needs to be over time.
- Deciding the properties of the communication mechanisms; e.g. stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, latency, etc.

Data and object Model

Every system represents objects and artifacts of external interest in an internal fashion. Choosing the data and object model means choosing how the software will represent those items of interest. Here the major design decisions include

- Choosing the major data abstractions, their operations, their properties.
- Determining how the data items are created, initialized, persisted, manipulated, translated, destroyed.

Management of Resources

One of the critical responsibilities of an architecture is to arbitrate the usage of shared resources. Management of resources includes

- Identifying the resources that need to be managed: hard resources (e.g. CPU, memory, battery, hardware buffers, system clock, I/O ports, etc.) and soft resources (e.g. system locks, software buffers, thread pools, etc.)
- Determining the resource limits.
- Determining which system element(s) manage each resource.
- Determining the resources that are shared and how these are arbitrated; e.g., the process/thread models employed; the scheduling strategies employed

Mapping Among Architectural Elements

Whereas architecture comprises multiple structures, and each structure comprises multiple elements, those elements across structures must have predetermined associations with each other in order for the design to make holistic sense. Mapping among architectural elements involves

- Deciding what are the elements in different architectural structures and how they map to each other. Examples include
  - the mapping of modules and runtime elements to each other: the runtime elements that are created from each module; the modules that contain the code for each runtime element.
  - the assignment of runtime elements to processors.
  - the assignment of items in the data model to data stores.
  - the mapping of modules and runtime elements to units of delivery.

Binding Time Decisions

Not all architectural decisions are made on the design table; some are intentionally delayed, so as to bring about greater flexibility and facilitate change. Binding time decisions involve deciding how and when decisions in the other models are resolved. Possible answers include

- compile time (e.g., compiler switches)
- build time (e.g., replace modules, pick from library)
- load time (e.g., dynamic link libraries [dlls])
- initialization time (e.g., resource files)
- run time (e.g., load balancing)

Choice of Technology

The architect must often determine which available technologies will be utilized. This involves

- Knowing which technologies that are available to realize the decisions made in the other models.
- Investigating the available tools to support this technology choice (IDEs, testing tools, etc.)
- Knowing what external support is available for the technology, such as courses, tutorials, examples, internal familiarity, availability of contractors who can provide expertise in a crunch, etc.

Checklists

There is a checklist for each of the cells in the following matrix

	Availability	Modifiability	Security	…
Allocation of functionality	Checklist questions			
Coordination model				
Data and Object Model				
…..				

Sample checklist cell - Coordination x Availability

Determine the system responsibilities that need to be highly available.

With respect to those responsibilities, ensure that:

- coordination mechanisms exist to detect an omission, crash, incorrect timing, incorrect response.
- coordination mechanisms exist to log the fault, notify appropriate entities, disable the source of events causing the fault, fix or mask the fault, or operate in a degraded mode
- failures of system responsibilities, and the artifacts that support them (processors, communications channels, persistent storage, processes) can be communicated and replaced, e.g. does failure of an external entity cause the coordination to fail?

Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation, e.g. how much lost information the coordination model can withstand and with what consequences?

Summary of hypothesis testing

Use architecturally significant requirements to identify specific portions of the current hypothesis that are relevant to the satisfaction of these requirements

Use various quality attribute testing techniques to determine whether quality attribute requirements are satisfied

Use quality attribute specific use cases and architecturally significant requirements to determine additional responsibilities that should be included in the next hypothesis.

Generating alternatives for next hypothesis

Tactics

Tactics -> responsibilities

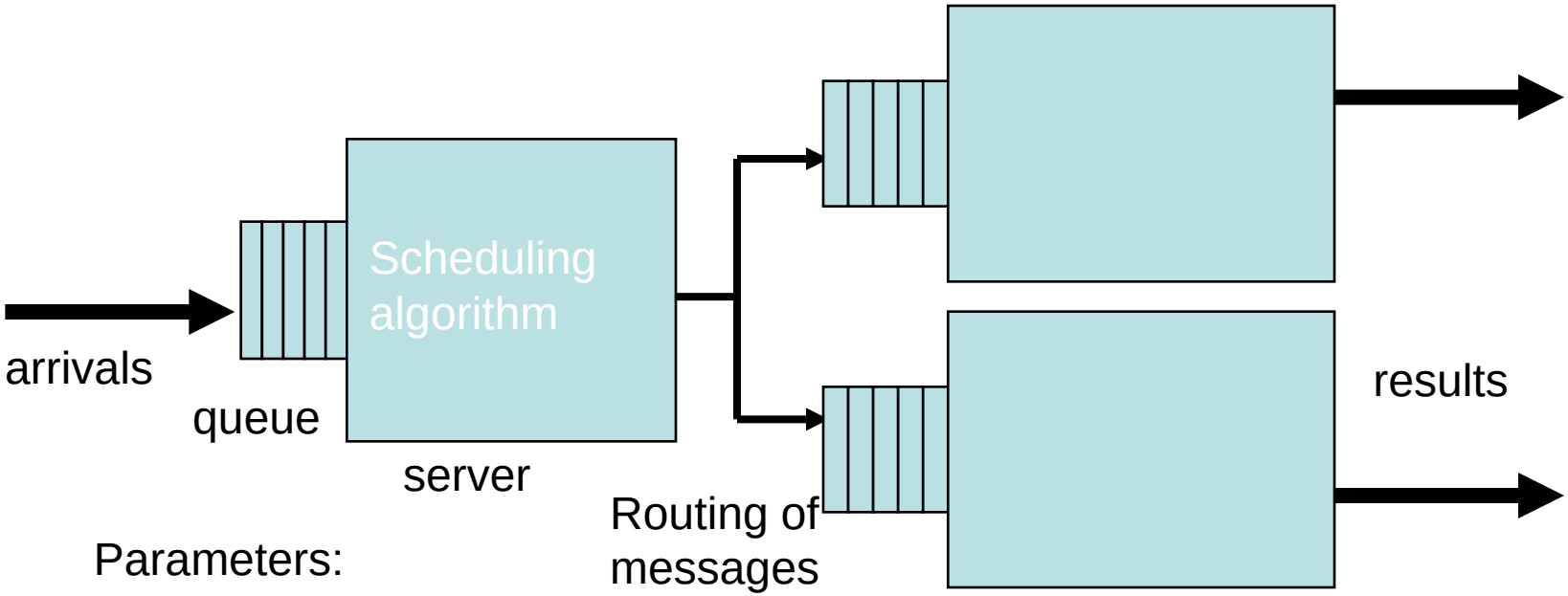
Generating new hypothesis from tactics and responsibilities

Tactics

Tactics are architectural means of controlling parameters of a model of a quality attributes.

We will explore models and tactics for performance. Lists of tactics exist for other quality attributes as well.

Queuing Model for Performance



Parameters:

- Arrival rate
- Queuing discipline
- Scheduling algorithm
- Service time
- Topology
- Network bandwidth
- Routing algorithm

Latency can be affected only by changing one of the parameters.

Controlling Performance Parameters

Architectural means (tactics) for controlling the parameters of a performance model

- *Arrival rate* – restrict access, differential rate/charging structure, constrain message size
- *Queuing discipline* – first-come first served (FCFS), priority queues, etc.
- *Service time*
  - Increase efficiency of algorithms.
  - Cut down on overhead (reduce inter-process communication, use thread pools, use pool of DB connections, etc.).
  - Use faster processor.
- *Scheduling algorithm* – round robin, service last interrupt first, etc.
- *Topology* – add/delete processors
- *Network bandwidth* – faster networks
- *Routing algorithm* – load balancing

Tactics are transformations on responsibilities and structure- 1

A tactic is one (or more) of the following types of transformations

- modify responsibility. The tactic increase message size can be achieved by modifying the responsibilities that construct messages to construct larger messages.
- introduce new responsibilities. The tactic introduce concurrency requires that responsibilities for forking the concurrent threads and joining those threads together be introduced.
- Introduce new structural elements. The tactic maintain multiple copies requires elements to store the new copy and maintain consistency among the copies.

Tactics are transformations on responsibilities and structure- 2.

- modify the properties of a responsibility. The tactic reduce execution time will result in a modification of a property (execution time) of the responsibility that is being made more efficient.
- decompose responsibilities. The tactic maintain multiple copies will result in the responsibility of storing information into one location being decomposed (and augmented) into responsibilities that store the information and synchronize the information with other locations.
- reallocate responsibilities. The tactic reduce computational overhead may result in responsibilities being reallocated from one process into another to reduce interprocess communication.

Generate next hypothesis

At this point we have as a result of the test phase

- Additional responsibilities as a result of responsibilities discovered through test cases or through tactics
- Revised responsibilities as a result of decomposition.
- Constraints on the allocation of responsibilities to modules as result of tactics.
- Other constraints on responsibilities such as budgeted execution time

These responsibilities and their constraints are merged with the current hypothesis to generate the next hypothesis.

Summary of design

Design is the process of generate and test.

The initial design is generated from existing or similar systems, frameworks and components, or patterns.

The design is tested against the architecturally significant requirements, a collection of quality attribute use cases, architectural decision categories to derive additional responsibilities and constraints on responsibilities.

The design is analyzed against quality attribute models to discover shortcomings.

Tactics are used to propose alternatives for improving the design

The next hypothesis is generated based on additional responsibilities and constraints discovered during test and analysis.

Curriculum question

Where in your curriculum do you teach architectural design (not O-O design)?

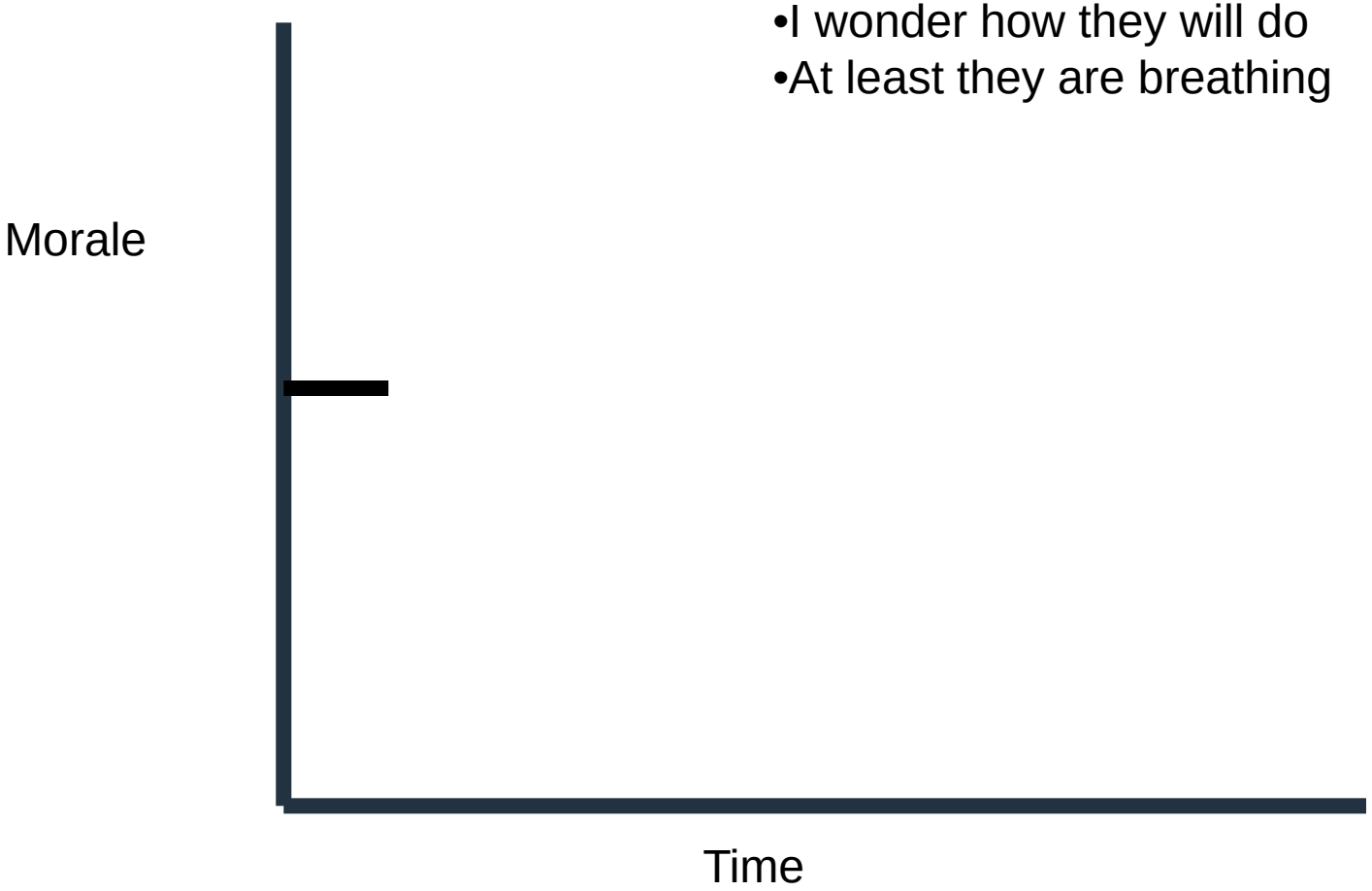
Experiences with being a client

I have been a client for a MSE studio three times:

- ArchE core. ArchE is a tool intended to support quality attribute oriented design.
- ArchE adding a reasoning framework. ArchE is intended to be extensible and we defined a language for adding capability relative to additional quality attributes.
- Usability Supporting Architectural Patterns checklist. This is a tool that allows an architect to review their own design with respect to a collection of responsibilities necessary to support particular usability features – e.g. customization

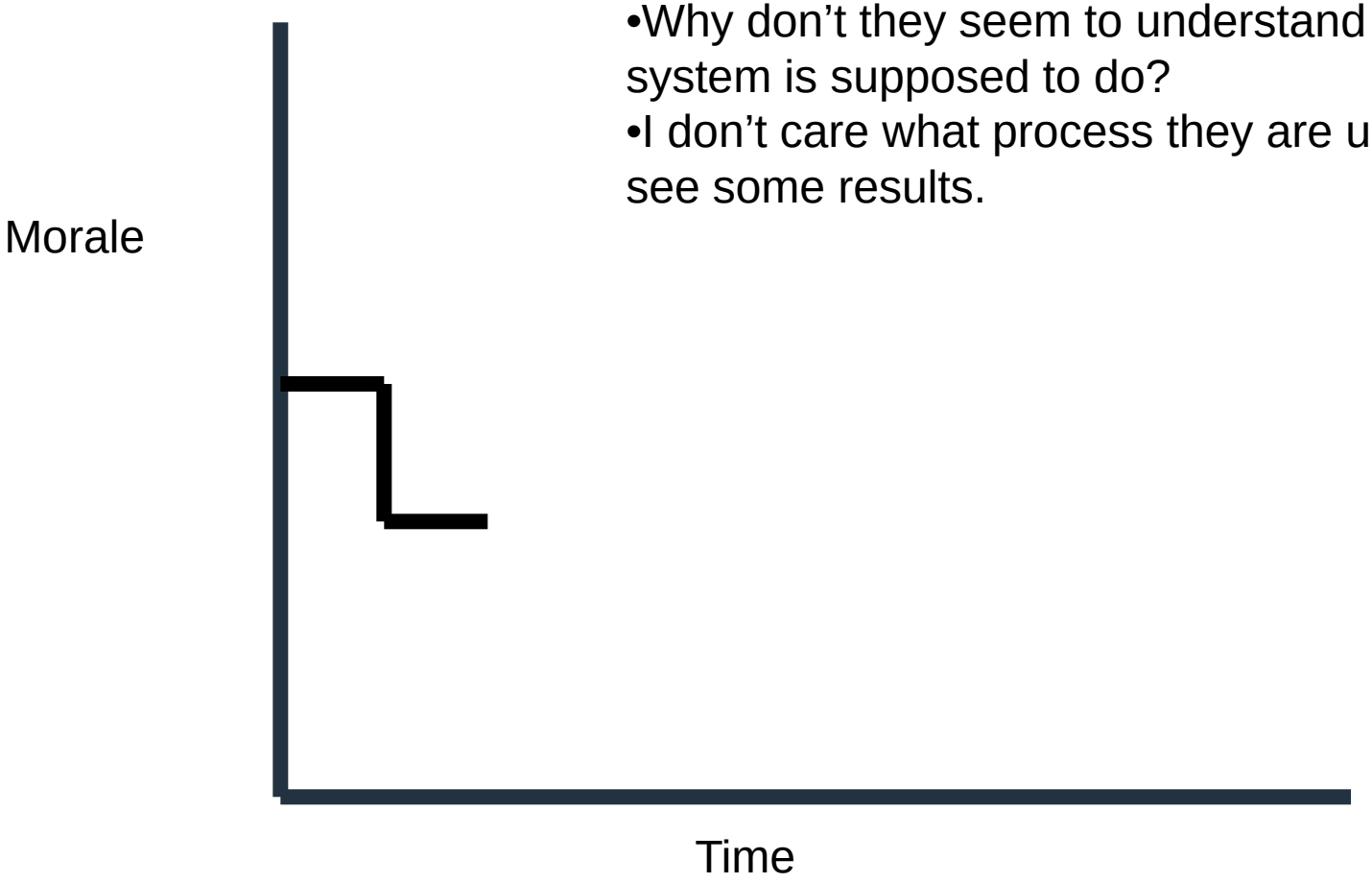
Client morale with respect to a project

Project inception
•I wonder how they will do
•At least they are breathing

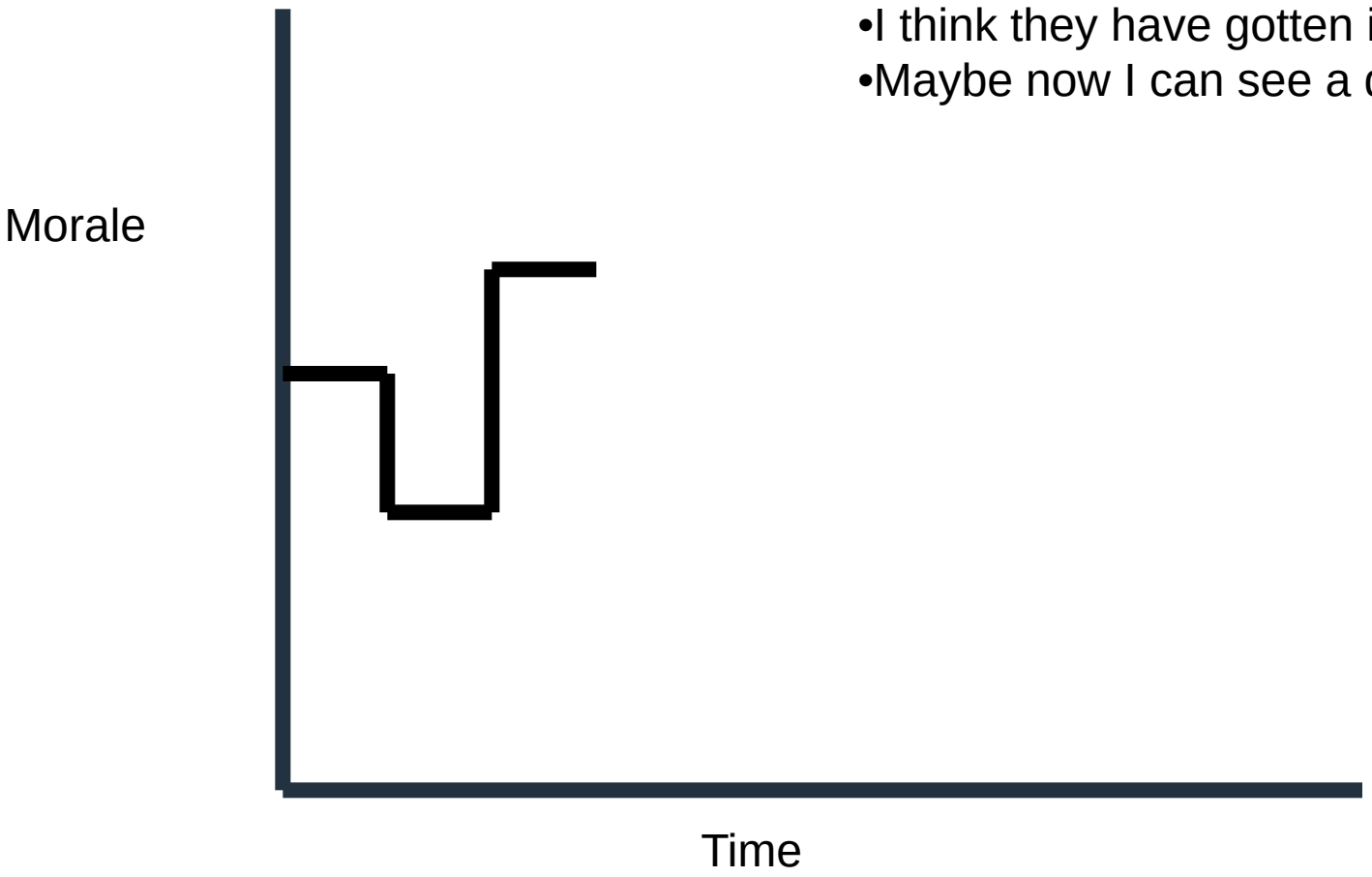


Client morale with respect to a project

Month 1:
•Why don't they seem to understand what the system is supposed to do?
•I don't care what process they are using, I want to see some results.

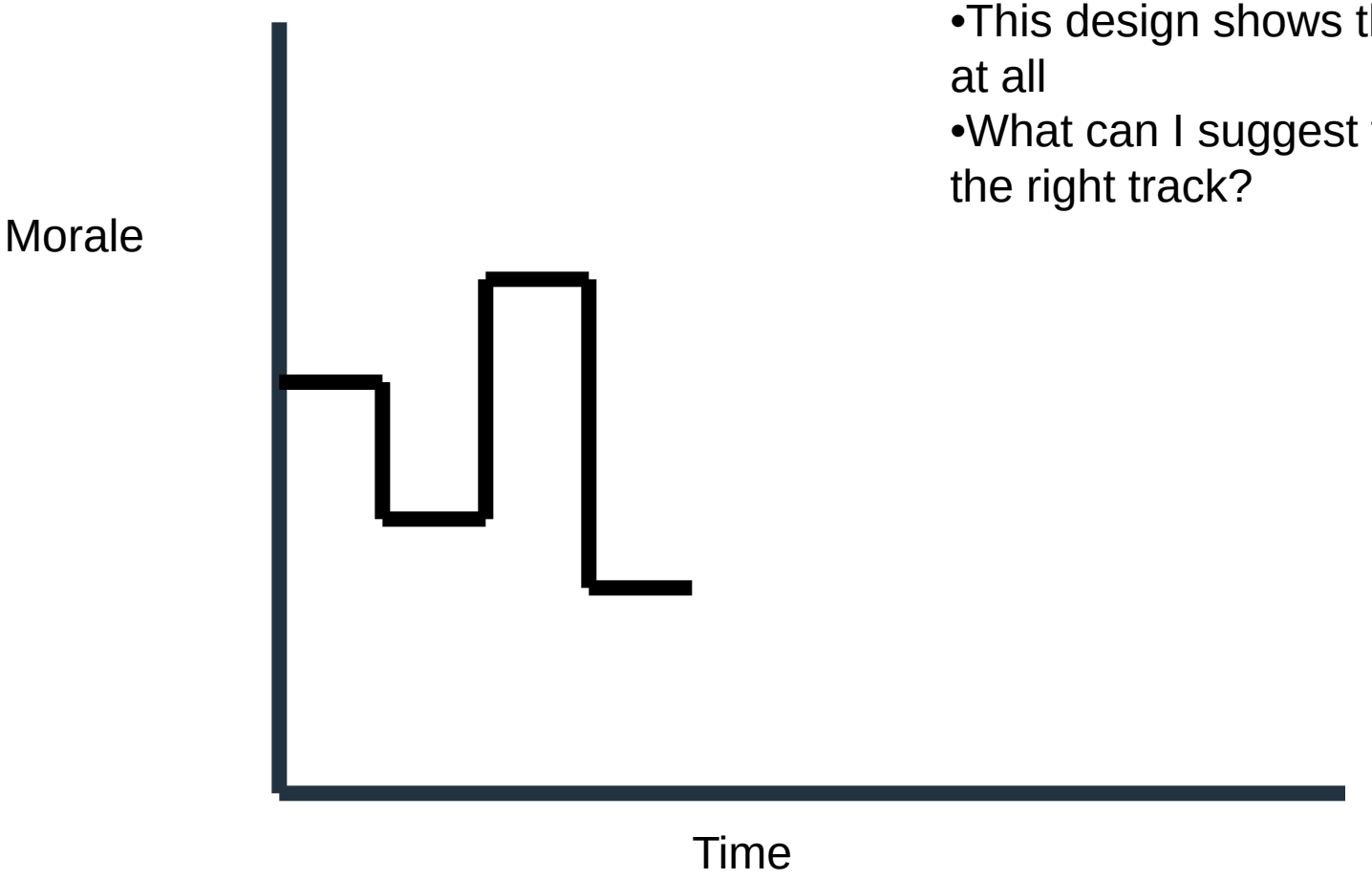


Client morale with respect to a project



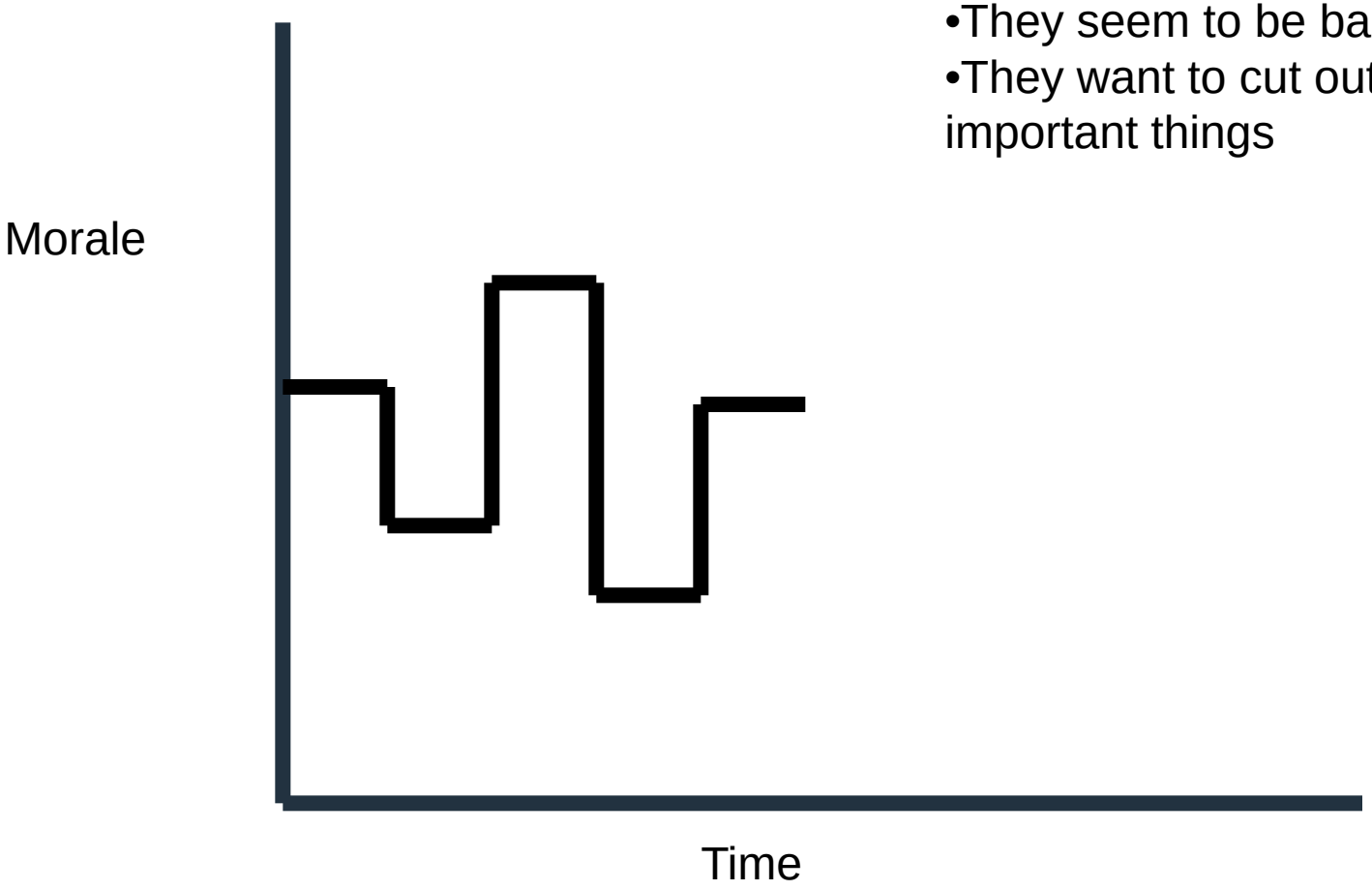
Month 3:
- I think they have gotten it, finally.
- Maybe now I can see a design

Client morale with respect to a project



Month 4:
- This design shows they don't get it at all
- What can I suggest to get them on the right track?

Client morale with respect to a project



Month 5:
- They seem to be back on track
- They want to cut out the most important things

Results of projects

ArchE core – unqualified success

ArchE extension – project was successful but we never used it because the language approach wasn't the right approach

Usability Supporting Architectural Pattern checklist – at project close I thought it was a success but then I tried to make a simple performance enhancement. The internals were abysmal. The students had no understanding of what should go in the front end (browser side) and what should be in the back end (content management, data base side). I ended up rewriting the whole thing.

Discussion question with respect to projects

Who is responsible for quality control?

- Clearly not the clients.
- In the last project, the system underwent an architectural review but the architecture was the standard three tier architecture and did not expose the students lack of understanding of what to put in each tier.
- When I was a mentor, I did not get into the details of the code the students wrote. Should the mentors be in charge of quality control?

Final questions??