Use Case Maps: A New Paradigm for Attributing Behaviour to System Architecture

R.J.A. Buhr
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
buhr@sce.carleton.ca
http://www.sce.carleton.ca/faculty/buhr

Key Points

- The ability to attribute behaviour to architecture is important for high-level understanding, designing, evolving, and reengineering all sorts of systems (from object-oriented programs to parallel and distributed computer systems).
- Scenarios are a good way of doing it, but popular scenario techniques, such as message sequence charts, that work well for small-scale systems do not scale up well.
- Use case maps provide a new, scenario-based way of attributing behaviour to architecture that solves the scaleup problem by the simple trick of representing scenarios in terms of cause-effect sequences of responsibilities along paths, instead of interaction sequences between components.
- The notation enables compact, composite maps to be drawn to represent behaviour patterns of whole systems in path terms.

This talk:

- Describes highlights of the approach, its relationship to other approaches, work in progress, and issues and directions for future work;
- Through examples, aims to convincesoftware and system engineers that the approach has depth and adds value, despite (and because of) its simplicity and deferment of detail.

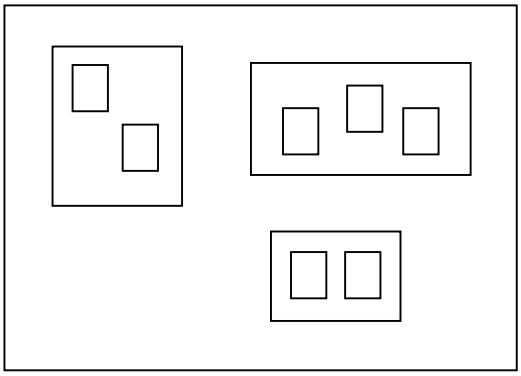
Complexity Factors in Systems

Setting the stage:

Definition: A system is a set of collaborating components, each of which may, through recursive decomposition, be a system.

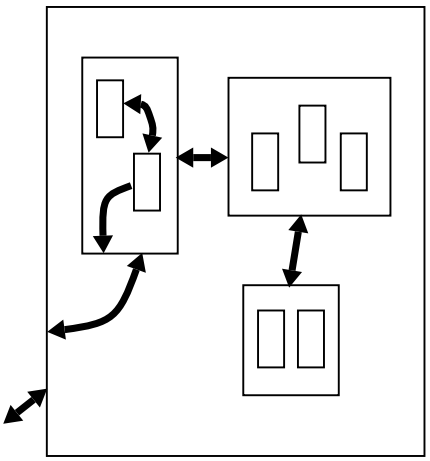
Component context diagram:

- *identifies and positions components*
- *an aspect of architecture.*



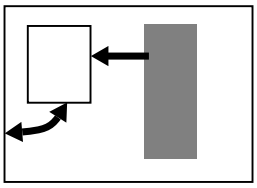
First Factor: Operation.

- Operational connective tissue: wiring.
- Wiring required both between peer components at the same level of recursive decomposition and between components at different levels of recursive decomposition, through all levels of decomposition.
- Wiring at level of calls, messages, or IPC does not scale up well.

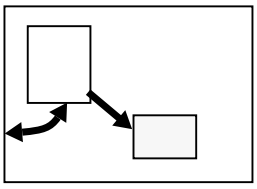


Second Factor: Assembly.

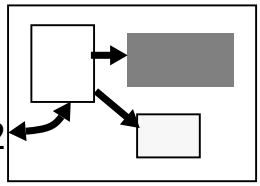
- Systems may change form (components and wiring) while they are running
- Routine property of systems and software, not unusual.
- Wiring in snapshots indicates *Operation*
- Intersnapshot gaps indicate *Assembly*
- Operation+Assembly=*Behaviour.*
- Described with wiring, this is heavyweight and complex.
- Contrast with lightweight way in which assembly happens in code (pointer manipulations).
- Complexity comes from viewing things in wiring terms.



snapshot
at time t0



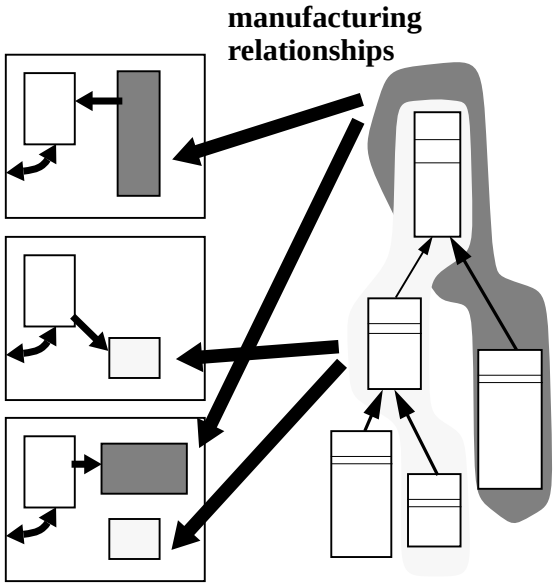
snapshot
at time t1



snapshot
at time t2

Third Factor: Manufacturing

- Behaviour (operation+assembly) intertwined with manufacturing—e.g., classes.
- In OOD, classes are where behaviour is defined, but the diagrams showing class relationships do not themselves give the system picture, which must be inferred from details in the class descriptions or diagrammed as a second-class abstraction in relation to classes.
- Messy, difficult-to-separate soup of details. Does not scale



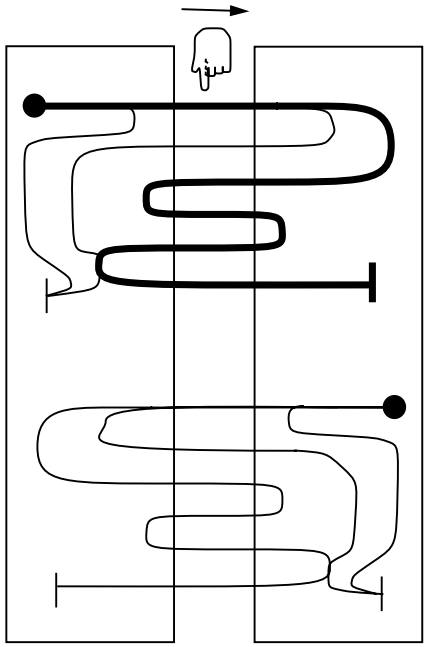
Essence of Use Case Maps

Use case maps raise the level of abstraction by simplifying all three of these complexity factors:

- **Operation.** Wires are replaced as connective tissue by cause-effect paths.
- **Assembly.** Sequences of snapshots are eliminated. Changing wirings between them are eliminated. Change is modeled as the movement of components into and out of slots.
- **Manufacturing.** Use case maps are first-class models in their own right that may be developed independently of manufacturing details, but related to them at any convenient time.

UCMs: Records of Finger-Pointing Sequences

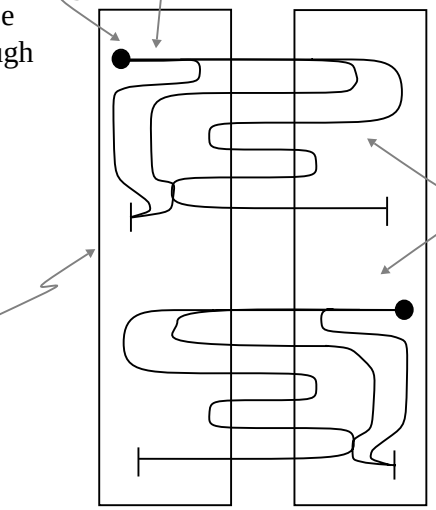
- Like a road map in which any end-to-end route may be traced with your finger.
- No wiring: sequences traced are cause-effect sequences, not interaction sequences.
- Basically record of routes traced by one or more such finger-pointing sequences (called “scenarios”).
- Concurrency represented, not by specific map notations, but by allowing multiple scenarios to be traced at the same time.



Notational Elements

2. responsibilities
• labeled points on paths
• coarse-grained units of activity
• bound to components
• high-level view

1. paths
• of use case scenarios
• from stimulus to response
• cause-effect chains through system
• routes for slot occupants
• first class models



3. interpath couplings
• pre/post-conditions
• constraints
• responsibilities
• state
• components
• asynchronous/ synchronous

4. components
• characterized by responsibilities
• no internal logic
• no interfaces
• fixed & slots

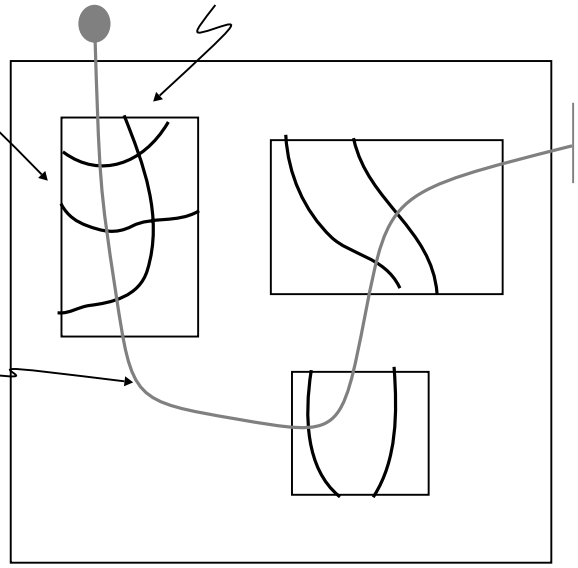
Paths

A component-centric view gives a complete small-scale picture but loses sight of the large-scale picture.

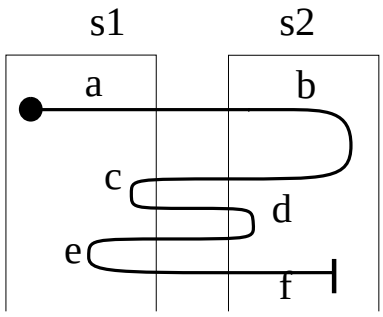
A path-centric view gives a large-scale picture, but loses sight of small-scale details.

Both views needed.

Other paths.



Sequences of Responsibilities

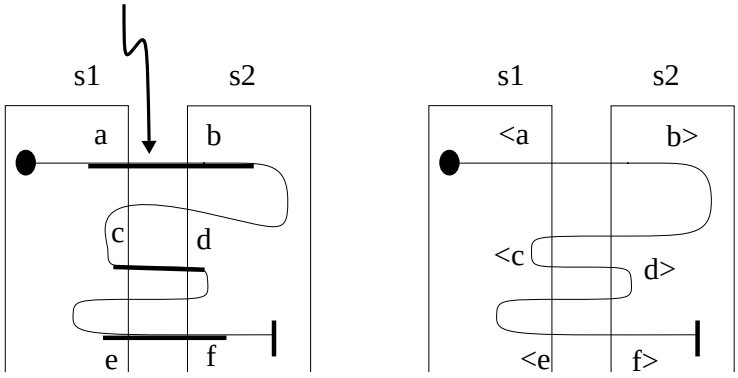


path over structure

- close to architecture
- easily composable to give composite pattern in compact form

Responsibilities may be Coarse-Grained

may be many
interactions here

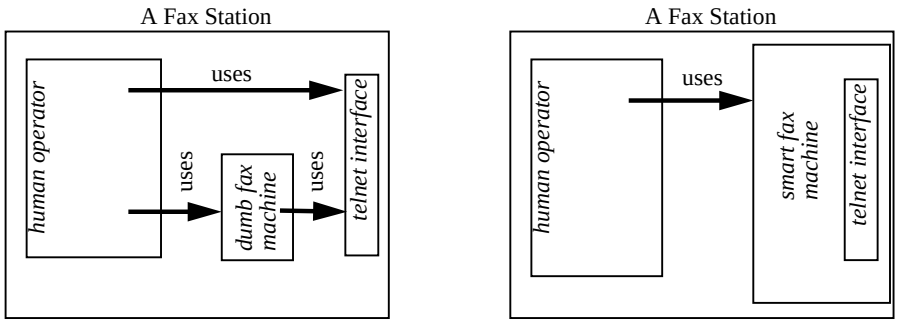


shared responsibilities

- big picture requires responsibilities to be coarse-grained
- coarse-grained responsibilities chained between components may have to be viewed as shared

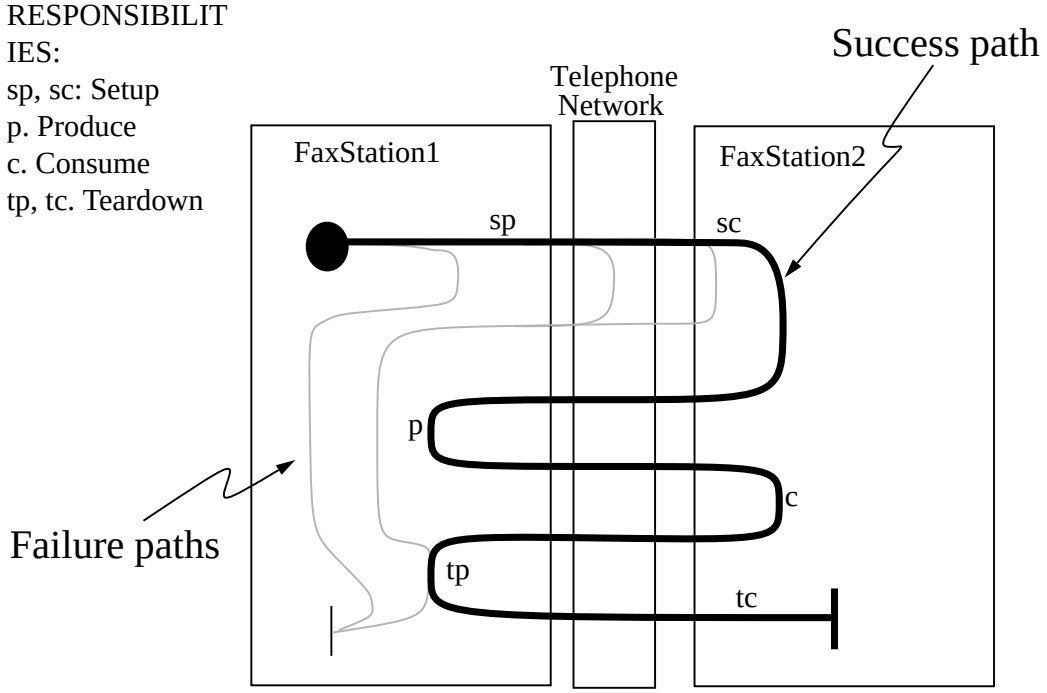
A Fax System Example

- Develop high-level behaviour patterns without making commitment to internal structures.
- Then bind them to any internal structure.
- Specific example of a general issue.



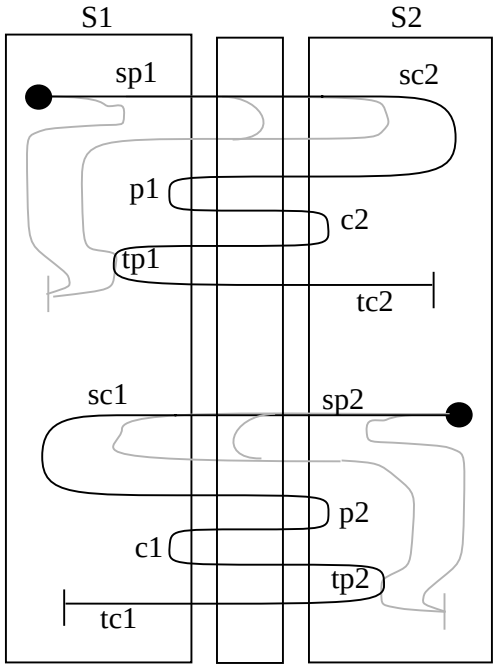
Negotiated Producer Consumer

RESPONSIBILITIES:
sp, sc: Setup
p. Produce
c. Consume
tp, tc. Teardown



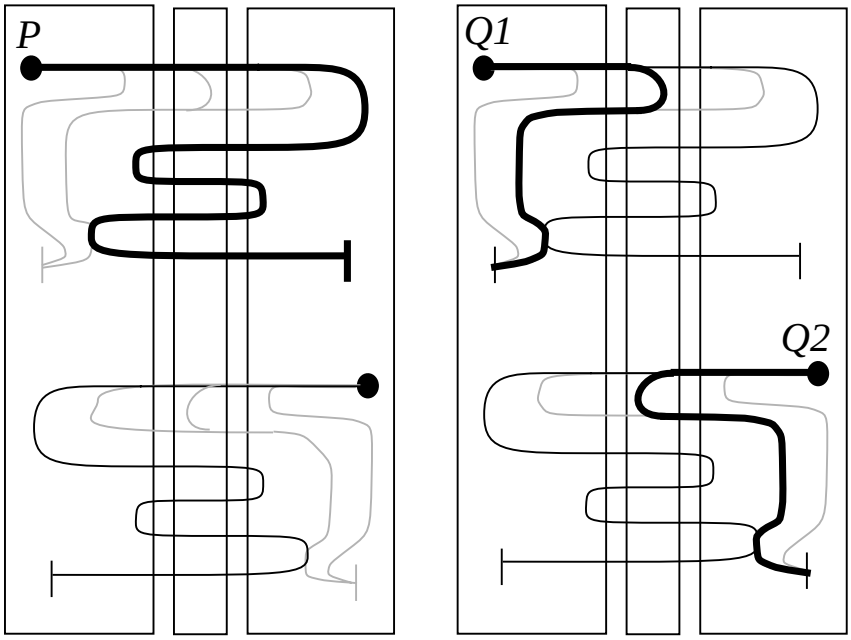
Composite Map Expresses Many Possible Scenarios

RESPONSIBILITIES:
sp, sc: Setup
p. Produce
c. Consume
tp, tc. Teardown



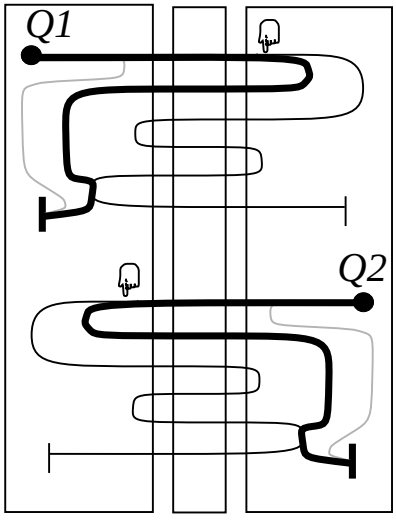
Interpath coupling
- telnet constraints
- shared responsibilities between S1 and S2
- local state in S1 and S2
- **explain by example —>**

Selected success (*P*) and failure (*Q*)scenarios

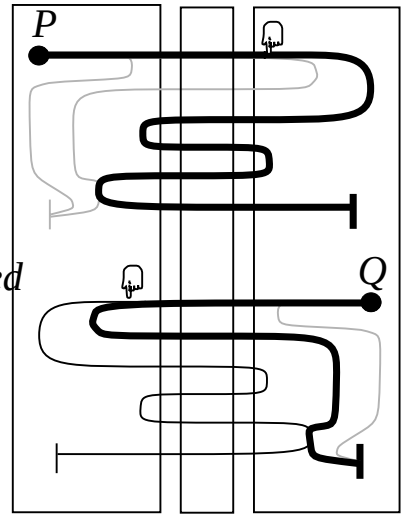


What-If Analysis

Can both fail this way?



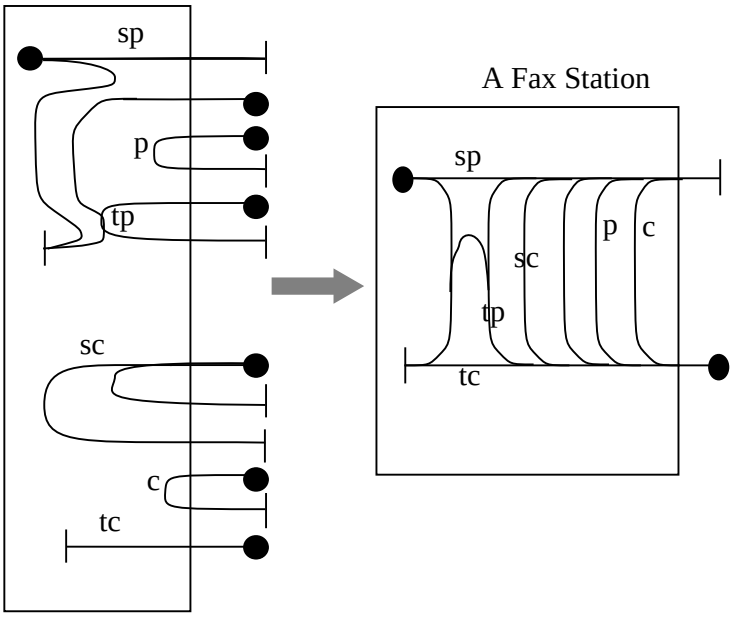
Can one succeed this way?



Scenarios would have to get through the telnet *concurrently* to the points marked by the fingers (how?) and then either discover the collision and fail or invoke a tie-breaking rule so one can succeed (must avoid both succeeding). Possible to get this far if one side performs off-hook just as the other causes the phone to ring. Human operators can then resolve the collision and negotiate. What about fax machines?

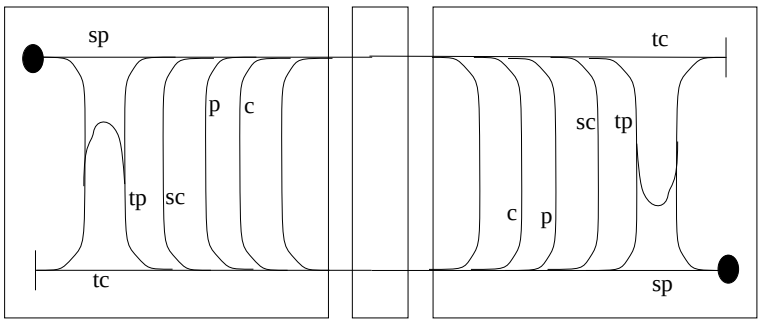
Factor to get Component-Centric View of One Fax Station

RESPONSIBILITIES:
sp, sc: Setup
p. Produce
c. Consume
tp, tc. Teardown



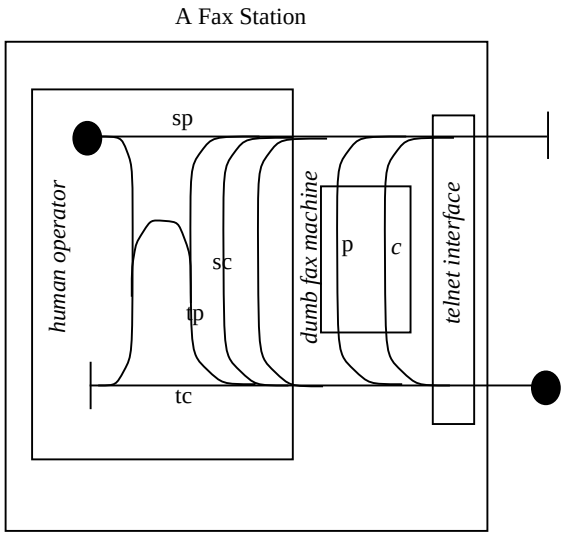
Reconnecting the Factored Maps

RESPONSIBILITIES:
sp, sc: Setup
p. Produce
c. Consume
tp, tc. Teardown



More compact but less visually informative

Internal Components of Factored Part

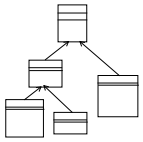


- this is only one possible choice (dumb fax machine)
- could do a different choice (intelligent fax machine) and then continue on with software decomposition for that choice
- same approach applies

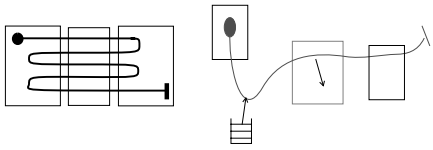
A Suite of Design Models with Use Case Maps

Stylized Symbols for Design Models

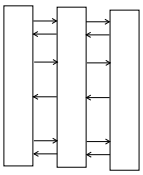
class relationship diagrams



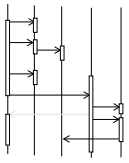
use case maps



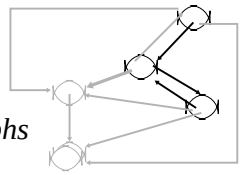
collaboration graphs



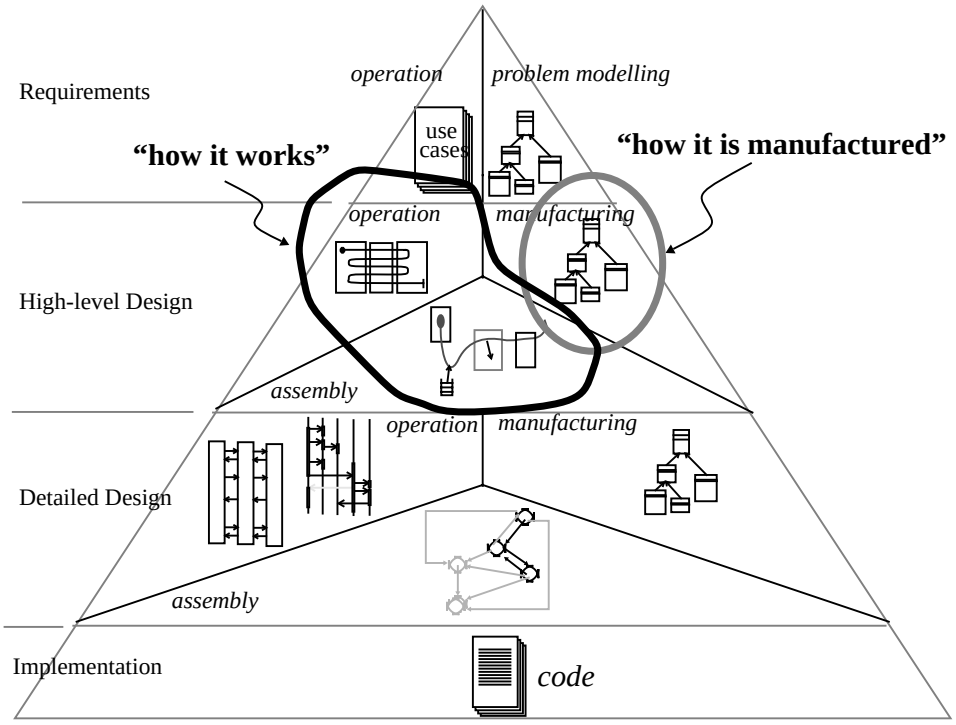
message sequence charts



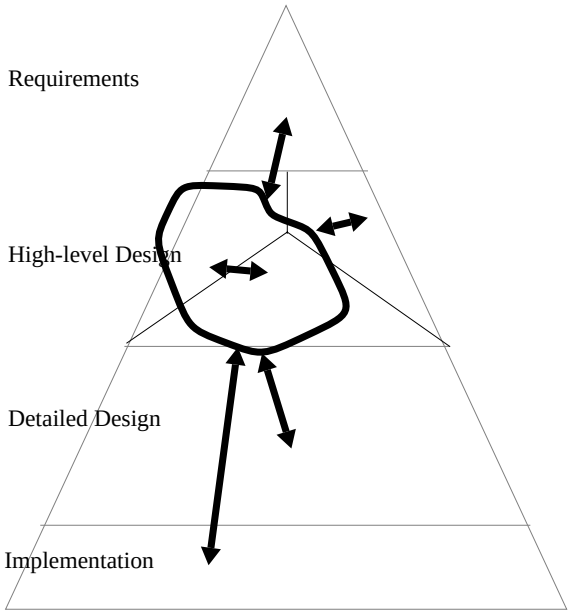
visibility (uses) graphs



Filling a Gap



Directions for Development



Tool support for UCMs

- First-class models
- Plain ordinary map creation/maintenance
- Linked to other models in same and different levels
- Formalized, analyzable, executable
- Time budgets along paths
- Performance estimates
- Tradeoff analysis
- Visualization
- ...

Properties of Use Case Maps

- visual composition: u.c.+architecture (not just u.c. notation)
- first class
- compact big picture
- exploit human pattern-recognition
- rational, traceable progression from requirements to solutions
- wireless —> lightweight operation+assembly
- scale up
- point-to-point stimulus-response patterns (with some ripple)
- technology-independent
- concrete work products
- no free lunch —> high level view loses sight of details
- starting point for detailed design with popular methods/tools
- add value to existing methods/tools by replacing magic
- have semantic depth