Lecture 11: Evolving Requirements

Last Week:
Agreeing Requirements
Negotiation
Conflict Resolution

This Week:
Evolving Requirements
Change management
Product Families
Viewpoints and Inconsistency management
Paraconsistent Logics

Next Week:
Integrated RE processes
selecting methods
method engineering
problem frames

Outline

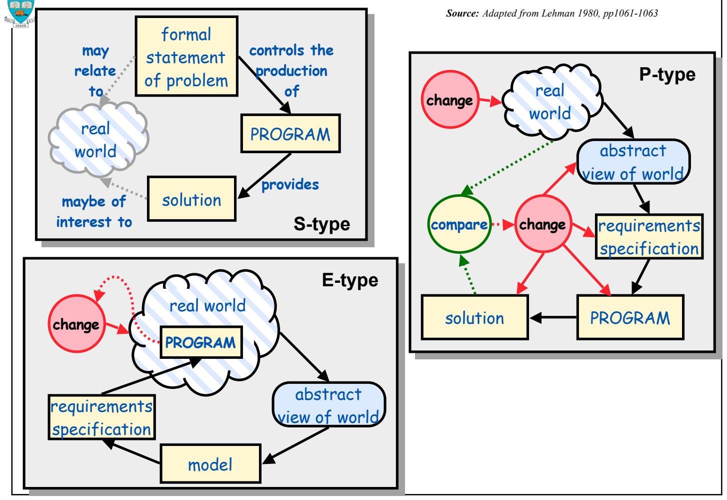
- **Software evolves because requirements evolve**
  - Laws of software evolution
  - Beyond specification singularity
- **Traditional change management**
  - Baselines and Change Requests
  - Configuration Management
- **Software Families**
  - The product line approach
- **Viewpoints**
  - …as a framework for understanding requirements evolution
  - Managing Inconsistency
  - Reasoning about change
  - Feature Interaction

Program Types

Source: Adapted from Lehman 1980, pp1061-1063

- **S-type Programs ("Specifiable")**
  - problem can be stated formally and completely
  - acceptance: Is the program correct according to its specification?
  - This software does not evolve.
    - A change to the specification defines a new problem, hence a new program
- **P-type Programs ("Problem-solving")**
  - imprecise statement of a real-world problem
  - acceptance: Is the program an acceptable solution to the problem?
  - This software is likely to evolve continuously
    - because the solution is never perfect, and can be improved
    - because the real-world changes and hence the problem changes
- **E-type Programs ("Embedded")**
  - A system that becomes part of the world that it models
  - acceptance: depends entirely on opinion and judgement
  - This software is inherently evolutionary
    - changes in the software and the world affect each other

Source: Adapted from Lehman 1980, pp1061-1063



Laws of Program Evolution

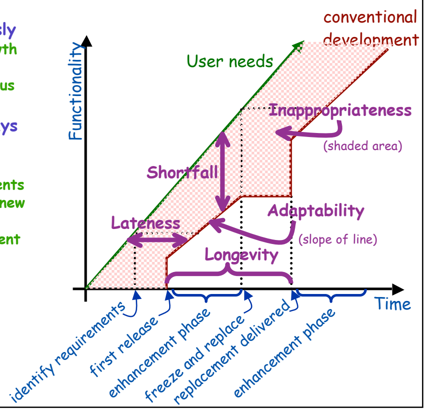
Source: *Adapted from Lehman 1980, pp1061-1063*

- **Continuing Change**
  - Any software that *reflects some external reality* undergoes continual change or becomes progressively less useful
    - change continues until it is judged more cost effective to replace the system
- **Increasing Complexity**
  - As software evolves, its *complexity* increases…
    - …unless steps are taken to control it.
- **Fundamental Law of Program Evolution**
  - Software evolution is self-regulating
    - …with statistically determinable trends and invariants
- **Conservation of Organizational Stability**
  - During the active life of a software system, the work output of a development project is roughly constant (regardless of resources!)
- **Conservation of Familiarity**
  - The amount of change in successive releases is roughly constant

Requirements Growth

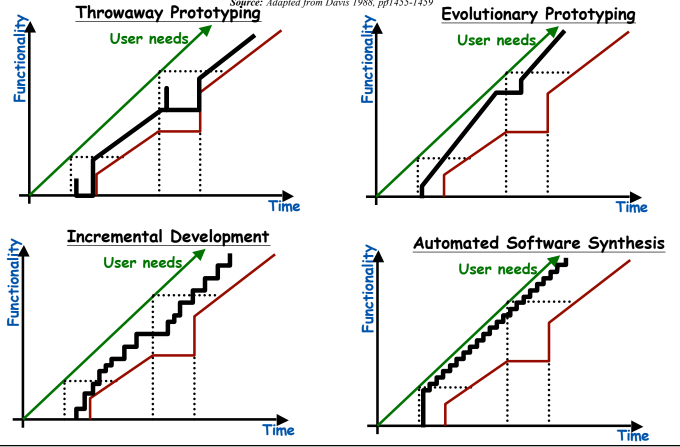
Source: *Adapted from Davis 1988, pp1453-1455*

- **Davis's model:**
  - User needs evolve continuously
    - Imagine a graph showing growth of needs over time
    - May not be linear or continuous (hence no scale shown)
  - Traditional development always lags behind needs growth
    - first release implements only part of the original requirements
    - functional enhancement adds new functionality
    - eventually, further enhancement becomes too costly, and a replacement is planned
    - the replacement also only implements part of its requirements,
    - and so on...



Alternative lifecycle models

Source: *Adapted from Davis 1988, pp1455-1459*



Software "maintenance"

Source: *Adapted from Blum, 1992, p492-495*

- **Maintenance philosophies**
  - "throw-it-over-the-wall" - someone else is responsible for maintenance
    - investment in knowledge and experience is lost
    - maintenance becomes a reverse engineering challenge
  - "mission orientation" - development team make a long term commitment to maintaining/enhancing the software
- **Basili's maintenance process models:**
  - Quick-fix model
    - changes made at the code level, as easily as possible
    - rapidly degrades the structure of the software
  - Iterative enhancement model
    - Changes made based on an analysis of the existing system
    - attempts to control complexity and maintain good design
  - Full-reuse model
    - Starts with requirements for the new system, reusing as much as possible
    - Needs a mature reuse culture to be successful

Traditional Change Management

- **Managers need to respond to requirements change**
  - Add new requirements during development
    - But not succumbing to feature creep
  - Modify requirements during development
    - Because development is a learning process
  - Remove requirements during development
    - requirements "scrub" for handling cost/schedule slippage
- **Elements of Change Management**
  - Configuration Items
    - Each distinct product during development is a configuration item
    - version control of each item
    - control which version of each item belongs in which build of the system
  - Baselines
    - A baseline is a stable version of a document that can be shared among the team
    - Formal approval process for changes to be incorporated into the next baseline
  - Change Management Process
    - All proposed changes are submitted formally as change requests
    - A review board reviews change requests periodically and decides which to accept
    - Review board considers interaction between change requests

Beyond "Product Singularity"

- **Most RE techniques focus on individual models**
  - "Build a model, get it consistent and complete, then validate it"
  - Assumes that RE is a process with a single definite output
    - The output is a complete, consistent, valid specification of the requirements.
- **This ignores reality!**
  - Requirements Engineering isn't just about obtaining a specification
    - Requirements are volatile; changes need to be managed continuously
    - The specification is never complete anyway!
  - There is never just one model:
    - There are multiple versions of models over time
    - There are multiple variants of models that explore different issues
    - There are multiple components of models representing different decompositions
    - Families of models evolve over time (add, delete, merge, restructure the family)
  - RE must address requirements evolution
    - How do we manage incremental change to requirements models?
    - How can multiple models (specifications) be compared?
    - How will changes to a model affect the properties established for it?
    - How do you capture the rationale for each change?
    - How do we reason about inconsistent and incomplete models?

Towards Software Families

- **Software reuse aims to cut costs**
  - Developing software is expensive, so aim to reuse for related systems
    - Successful approaches focus on reusing knowledge and experience rather than just software products
    - Economics of reuse are complex as it costs more to develop reusable software
- **Libraries of Reusable Components**
  - domain specific libraries (e.g. Math libraries)
  - program development libraries (e.g. Java AWT, C libraries)
- **Domain Engineering**
  - Divides software development into two parts:
    - domain analysis - identifies generic reusable components for a problem domain
    - application development - uses the domain components for specific applications.
- **Software Families**
  - Many companies offer a range of related software systems
    - Choose a stable architecture for the software family
    - identify variations for different members of the family
  - Represents a strategic business decision about what software to develop

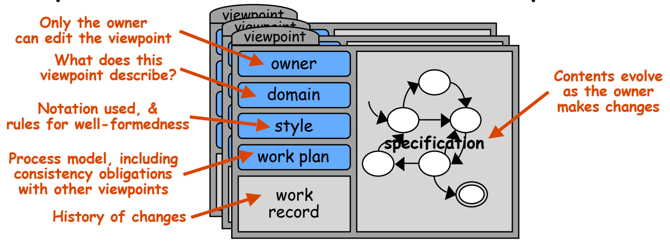
Viewpoints - Motivations

Multiple Perspectives	Distributed Modeling
Many different stakeholders	Collaborating analysts & stakeholders
Diverse kinds of Domain Knowledge	Multiple modeling methods
Conflicting views (& negotiation)	Continuous evolution of requirements
Many representation schemes	Imperfect communication links

- **Delaying Resolution of Inconsistency**
  - Inconsistency caused by:
    - Conflict between knowledge sources
    - Different interpretations
    - Communication problems between developers
    - Different development speeds
    - Divergence from prescribed method
    - Mistakes
  - Single model with consistency enforcement is too restrictive
    - Single model becomes a bottleneck for distributed modeling process
    - Consistency enforcement prevents entry of divergent/tentative ideas
  - Inconsistencies generally arise where there is the most uncertainty
    - Premature resolution may entail premature design decisions
    - Inconsistency implies more knowledge acquisition needed!
    - *More radically:* Some inconsistencies never get fixed…

The basic framework

→ Requirements model is a collection of viewpoints:

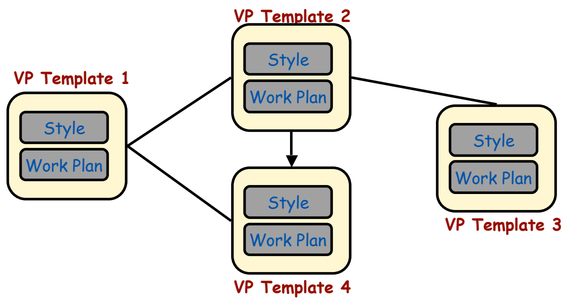

Only the owner can edit the viewpoint
What does this viewpoint describe?
Notation used, & rules for well-formedness
Process model, including consistency obligations with other viewpoints
History of changes
viewpoint
viewpoint
viewpoint
owner
domain
style
work plan
work record
specification
Contents evolve as the owner makes changes

- Viewpoints are instantiated from viewpoint templates
  - Template only has style and work plan slots filled
  - Development of templates is a separate "method engineering" task
  - A method provides a set of templates designed to be used together
- Viewpoints contain consistency rules (no central control)
  - Internal consistency rules for checking a viewpoint's specification
  - External consistency rules for inter-viewpoint checks
  - Work plan provides guidance for when to apply each consistency rule

Method Engineering with VP Templates

→ Method = Configuration of ViewPoint Templates

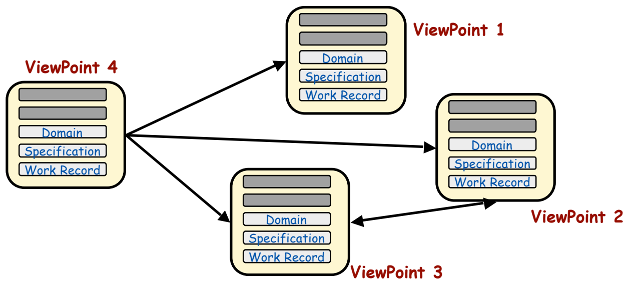
- *A method provides a set of templates designed to be used together*
- Development of templates is a separate "method engineering" task



Method Use with ViewPoints

→ Specification = Configuration of ViewPoints

- *A system specification is a collection of Viewpoints related by "inter-Viewpoint rules"*



Advantages of the approach

→ Stakeholder buy-in and Traceability

- Viewpoint owners can be roles, people, teams,…
- Each stakeholder's contribution is modeled in an appropriate notation
  - Stakeholders can identify and validate their own contributions
  - Increases stakeholder 'ownership' of the requirements process
- Requirements can be traced back to a source/authority

→ Structuring the development process

- Each viewpoint is an independent 'workpiece'
  - viewpoints as a distributed, loosely-coupled, suite of development tools
- No global control, no global enforcement of consistency
  - supports synchronous and asynchronous working
  - consistency checking rules act as explicit re-synchronization points

→ Structuring the descriptions

- Different stakeholders' contributions are modeled separately
  - Separation of concerns
  - Richer models through the use of multiple problem structures
- Resolution of inconsistency can be delayed
  - Supports negotiation by allowing detailed comparison of viewpoints
  - Encourages early modeling and expression of divergent views

Inconsistency Management

→ Inconsistency arises from:
- Conflict between knowledge sources
- Different interpretations
- Communication problems between developers
- Different development speeds
- Divergence from prescribed method
- Mistakes

→ Definition of inconsistency
- "two parts of a specification do not obey some relationship that should hold between them". (Easterbrook & Nuseibeh, 1995)
- Relationships may link
  - syntactic elements of partial specifications;
  - semantics of elements in partial specifications;
  - sub-processes of the overall development process.
- Relationships arise from:
  - definition of the method;
  - practical experience with the method;
  - local contingencies during development.

Example Consistency Rules

→ E.g 1: in structured analysis:
- In a data flow diagram, if a process is decomposed in a separate diagram, then the input flows into the parent process must be the same as the input flows into child data flow diagram.

→ E.g. 2: Use of domain concepts:
- For a particular Library System, the concept of operations document states that "User" and "Borrower" are synonyms. Hence, the list of user actions described in the help manuals must correspond to the list of borrower actions in the requirements specification.

→ E.g. 3: Process rules:
- Coding should not begin until the Systems Requirement Specification has been signed off by the Project Review Board (PRB). Hence, the program code repository should be empty until the SRS has the status 'approved by PRB'.

Lessons about inconsistency in practice

→ some inconsistencies never get fixed
- because the cost of changing the documentation outweighs the benefit
- humans are good at inventing workarounds

→ living with inconsistency is a risky decision
- risk factors change, so the risk must be constantly re-evaluated

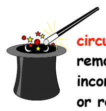
→ some consistency checks are not worth performing
- waste of money to establish consistency where change is anticipated
- … also where documents are early drafts, or are full of known errors

→ inconsistency is deniable
- e.g. because of face saving and defensiveness - inconsistency seen as bad!
- e.g. because you can always question the formalization!

Living With Inconsistency

→ Toleration?
- Detection is vital:
  - living with inconsistency is only safe if you know what inconsistencies exist

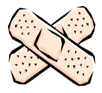
→ Handling:


circumvent - remove/replace inconsistent elements, or remove/modify the rule that was broken;

ignore - if the inconsistency can be isolated and is not important;

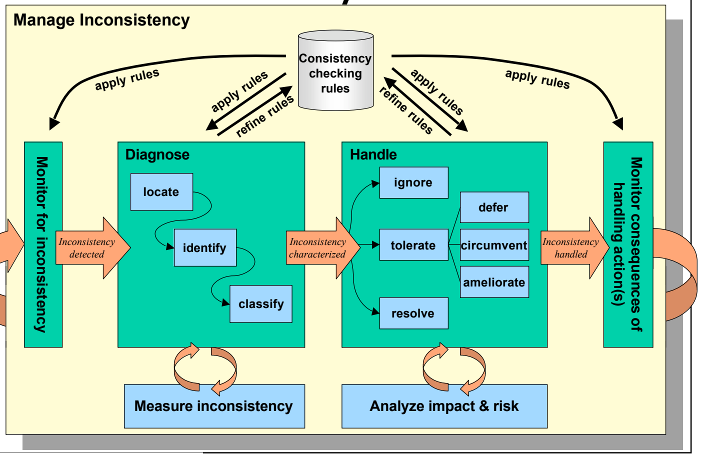
delay - if needed information is not immediately available;

resolve - negotiate a new solution or find a compromise.


ameliorate - take actions that improve the situation but which don't resolve the inconsistency;

→ Inconsistencies usually indicate that more information is needed.

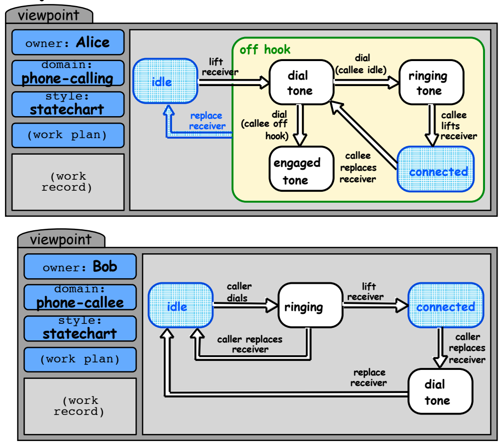
An Inconsistency Process Model



Sample Inconsistencies

- Rule 1:
  "If a transition between two states is described in one ViewPoint, and both states are described in the second ViewPoint, then the transition should also be described in the second ViewPoint".
  (e.g. see the elements shaded blue)
- Rule 2:
  "If a state is shown as belonging to a super-state in one ViewPoint, and the same state is included in the second ViewPoint, then the super-state must also be included in the second ViewPoint".
  (e.g. see the affected elements shaded yellow)

These rules ensure the notation is used correctly



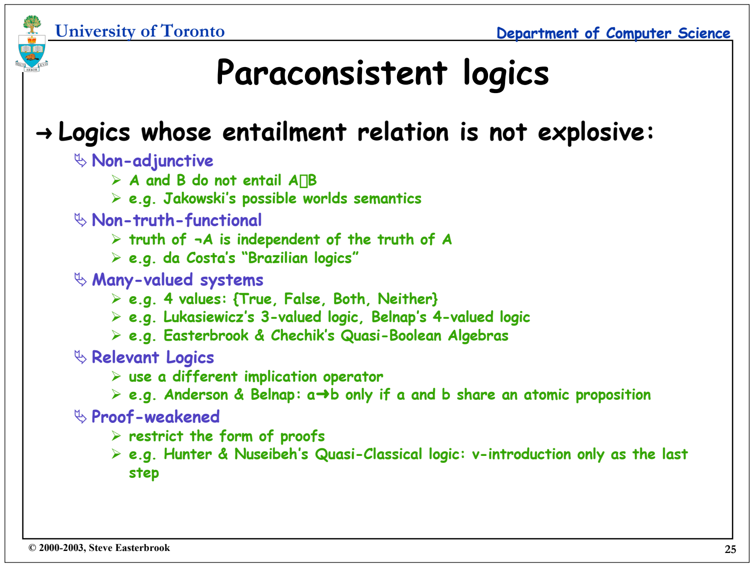
Viewpoint Consistency Checking

1. Where does responsibility lie?
   - ViewPoint owners are responsible for changes local to their own ViewPoints
     - May post requests/suggestions to others.
     - Not forced to synchronize ViewPoints
2. How are relationships expressed?
   - Consistency rules express relationships that should hold between ViewPoints
     - Each ViewPoint has its own list of rules
     - no central control
3. When should ViewPoint relationships be checked?
   - ViewPoint owners check rules whenever they need to…
     - … with guidance from local process model
4. How are relationships between ViewPoints checked?
   - Transaction management system between ViewPoints
   - Both ViewPoints notified of outcome
5. How are inconsistencies resolved?
   - List of actions associated with each rule
     - Actions don't necessarily make a complete resolution
     - Actions arise from method design, and experience with method use
6. What if inconsistencies are not resolved?
   - Unresolved inconsistencies are recorded in the work record.
     - Subsequent changes that affect known inconsistencies are tracked
7. What if future changes interfere with a resolution?
   - Successful check does not guarantee the relationship will continue to hold
     - Each rule may need to be applied a number of times during development
   - Changes that affect resolved inconsistencies are tracked
   - Actions and rationale involved in resolution are also stored.

Reasoning in the Presence of Inconsistency

→ Dialetheism - "some contradictions are true"
- dialetheia
  - a true contradiction, I.e. that A and not(A) are both true
- trivialism
  - the view that all contradictions are true
- explosive inference
  - an inference relation is explosive if a contradiction entails everything
- Law of non-contradiction has been orthodoxy in Western Philosophy since Aristotle (but not in Eastern Philosophy!)
  - LNC is often taken as a precondition for rationality

→ Dialetheists question this orthodoxy for a number of reasons:
- inability to handle self-referential paradoxes (e.g. the liar paradox)
- problems in handling legal reasoning - laws are contradictory for special cases
- quantum physics - a particle may be in two places at once
- epistemological reasoning - people can (rationally) hold contradictory beliefs
- Kuhnian paradigms - scientific theories often have undiscovered exceptions…
- reasoning with vague predicates - an adolescent is both a child and not a child



Paraconsistent logics

→ **Logics whose entailment relation is not explosive:**

- Non-adjunctive
  - A and B do not entail A∧B
  - e.g. Jakowski's possible worlds semantics
- Non-truth-functional
  - truth of ¬A is independent of the truth of A
  - e.g. da Costa's "Brazilian logics"
- Many-valued systems
  - e.g. 4 values: {True, False, Both, Neither}
  - e.g. Lukasiewicz's 3-valued logic, Belnap's 4-valued logic
  - e.g. Easterbrook & Chechik's Quasi-Boolean Algebras
- Relevant Logics
  - use a different implication operator
  - e.g. Anderson & Belnap: a→b only if a and b share an atomic proposition
- Proof-weakened
  - restrict the form of proofs
  - e.g. Hunter & Nuseibeh's Quasi-Classical logic: ∨-introduction only as the last step