An Anecdote

RPI instruction tells the processor to READ PROGRAMMER INTENT

Wouldn’t it be nice if the computer could understand what we are trying to do?

An Overly Simplified View of Intent

Does the action match what we intended?
Expressing Intent to a Computer

- We have spent decades trying to build programming systems for the purpose of telling a computer what we want it to do.
  - Programming languages define operational behavior
  - Compilers translate intent into instructions
  - Contracts and assertions enforce correct behavior
- Advanced programming systems:
  - Abstract away concepts specific to the platform
  - Allow for partitioning of sub-tasks and sub-goals
  - Restrict control and data flow
  - Create a virtual environment for problem-solving
- Errors are behaviors where:
  - Intent is incorrectly expressed to the computer
  - Intent is incompletely understood by the programmer

Modern Software Design

- Primarily a structuring and abstraction problem
  - For functional intent we know:
    - Group into procedures, classes, modules, etc.
    - Coupling and cohesion affect:
      - Performance
      - Reliability - fewer unexpected or undefined interactions
  - For new functionality:
    - Low impact of change through good modularity
    - New interactions are limited in scope
- Abstraction yields cognitive benefits
  - Fine-grain details are abstracted away
  - Large-scale design becomes possible
  - Abstracted designs can be inspected for design qualities
  - Complex systems can be generally understood quickly

Design is for Humans

- CLAIM: There are some technical benefits of certain design strategies, but comprehensibility is the primary objective of modern design and analysis.
  - Code elements are given "intentional" names
  - Organization makes "clear" the intent of a set of instructions
  - Modularity (coupling, cohesion) → abstract complexity within an interface
  - The computer has no use for the programmer’s "intent"
    - Counter-example: Expert system?
- CLAIM: Flexibility, elegance, testability, adaptability, etc. are all aspects of comprehensibility
  - Spaghetti code executes nicely, thank you
  - Counter-example: Distributed or replicated enterprise app.
- CLAIM: We have spent considerably less time studying how to express intent to people
Problem Structuring

→ Well-Structured Problems:
  - Relationship between problem, solution methods, and criteria
    - Coding a well-defined algorithm

→ Ill-Structured Problems:
  - Not well-structured (i.e., no domain guidance on solution methods or evaluation)
    - Deciding what to build (requirements selection)

→ Problem Structuring:
  - The act of turning ISPs into WSPs
  - Software Analysis and Design:
    - Select requirements to implement
    - Given a requirement, decompose into a set of goals
    - Transform goal into a detailed design
    - Treat design as a WSP, and abstract its complexity, and use to solve another goal

Software Design Decisions

→ What is a design decision?
  - Separating functional units into procedures (methods)
  - Defining interfaces for procedures
  - Grouping procedures into classes / modules
  - Defining interfaces for modules
  - Etc.
  - Prescriptive approaches provide strategies or methods for problem structuring
    - Top-down, Bottom-up, Stepwise refinement
    - OOAD
    - CBSP

How Do Software Designers Think?

→ Opportunistic Decision Making
  - Decisions made with partial knowledge influence later decisions as fact
  - Emergent knowledge and partial solutions
    - Discovery of partial WSPs from domain knowledge
  - Emergent requirements need attention
    - Immediate Structuring ISP into WSP
  - Drifting
    - Explore dependencies and assumptions
  - Scenario exploration
    - Make ill-structured requirements concrete
    - Verify partial solutions
    - Confirm inferred requirements
  - Early design activities are opportunistic, rather than prescriptive

Rational Decision Making

→ A decision is made based on criteria and rationale
→ Consequential choice of an alternative
  - Possible actions and outcomes
  - Utility function assigns value to options
  - Probabilities of outcomes
  - Assumptions behind Rational Decision Making
    - Set of possible options are known
    - Probabilities of outcomes are known
    - Optimality is desirable
    - Cost of decision process is not a concern or is less than the cost of a sub-optimal decision
  - Useful for WSPs
Naturalistic Decision Making

- Situational decisions
  - Made on partial knowledge + personal expertise
  - Preserved until they are invalidated
- Characteristics of Naturalistic Decision Making
  - Dynamic or volatile situations
  - Incomplete knowledge and ill-defined tasks and goals
  - Knowledgeable and experienced decision makers
  - Situational assessment over consequential choice
  - Alternatives not considered until rejection
  - Satisficing solutions
- Useful for ISPs

Problem Structuring and Decision Making

- Software design is a combination of:
  - Well-structured and Ill-structured problems
  - Opportunistic and Prescriptive structuring methods
  - Rational and Naturalistic decision making
- Structuring Methods:
  - Personal Experience
  - Opinion, Ideas
  - Domain Knowledge
  - Group Knowledge
  - External Influences
  - Existing Models of the Problem
  - Existing Processes
  - Preferred Evaluation Criteria

Decision-Making Cycle

- What Were We Trying To Do?
  - In the life of a piece of software
    - Some decisions were rational
    - Some decisions were naturalistic
    - Some decisions were arbitrary
    - Some decisions were deferred
- Over time:
  - As rationale is lost, distinction between decision types is lost
    - Rational decisions relate to well-structuredness and optimality
    - Naturalistic decisions were situationally satisficing based on partial solutions and incomplete knowledge
  - Assumptions and Dependencies are forgotten or ignored
What Are We Trying to Do?

The software understanding problem is an attempt to reconstruct:

- The rationale for rational decisions
- The situational context and expert knowledge for naturalistic decisions

We want to:

- Evolve software
- Maintain software
- Reuse software
- Reuse and transfer design knowledge and expertise

We have spent the semester looking at ways to:

- Record design intent and rationale
- Design for comprehensibility
- Use design knowledge to recover or infer intent

General-Purpose Rationale Systems

QOC, IBIS/PHI, DRL, etc.

Rationale systems have their roots in argumentation

- Two or more sides (alternatives)
- Supporting and objecting arguments

Motivation:

- Support decision making through visualization
- Representation in semi-formal notation facilitates computer support

Two ways to use rationale system:

- Prescriptive: capture evolving arguments and use utility function on criteria to select among alternatives
- Descriptive: justify a made decision by recording considered alternatives and criteria

Problems with General Rationale Systems

Software design decisions are:

- Non-rational
- Opportunistic
- Ill-structured
- At different levels of abstraction

Cognitive complexity of argumentation systems occludes opportunistic thought

No prescriptive value to software domain

Documenting rationale provides little upstream value

Descriptive value only benefits later designers

General systems fail to leverage inherent structure of software design decisions

Fake It!

Because there is something satisfying about rational decisions, treat all decisions as rational

In mature engineering professions, many tasks are WSP

We want to believe that Software Engineering is an engineering profession

Express SE problems as WSP with well-defined goals and decision processes (i.e., that it is rational)

Emphasis on prescriptive methods of design

“We will never find a process that allows us to design software in a perfectly rational way... [but] we can present our system to others as if we had been rational designers and it pays to pretend do so during development and maintenance.”
Problems with Faking Design Rationale

- Naturalistic decisions are situational
  - Difficult to differentiate between essential domain criteria and dynamic or volatile criteria
- Faked rationale tends to be uniform
  - What level of abstraction / granularity to use?
- Does not necessarily reflect real alternatives
  - How many alternative solutions should be faked?
  - Are these alternatives realistic or practical?
- Bad or failed solutions are interesting
  - Faked rationale describes successful designs
  - "The best prototype is a failed project" (Curtis, et.al.)
- Faked rationale uses "timeless" inferential reasoning
  - See Potts & Bruns - infer rationale from an existing design, process description, and natural language documentation
  - If you can infer rationale, why document faked rationale?

Hybrid Software Rationale Systems

- General rationale systems are semi-formal
  - Content of nodes is informal
  - Link structure is formal
- Use SE design domain knowledge structure nodes
  - Scope definition of a design "decision"
  - Define abstraction
  - Associate decisions with design artifacts
- Associate with a prescriptive problem structuring process
  - Potts & Bruns
  - Archium
  - SEURat
- Provide upstream and downstream value

Potts & Bruns (1988)

- Argumentative rationale with design process
  - Modified IBIS
  - Liskov and Guttag (proto-OOD abstract data type design)
- Incorporates design artifacts into rationale model

Rationale Structure (P&B)

- Relationships between artifacts are defined
- Decisions are classified by type as issues
  - Issues correspond to specific steps in L&G process
- Node elements are structured with a semi-formal schema
  - Specific explanations are natural language
Problems with Potts & Bruns

- They fake it
  - Sample problem is taken from L&G book and rationale inferred from descriptive text and process knowledge
  - Do not evaluate cost of documenting process
  - Prototype hypertext tool for supporting the process
  - Problem definition does not allow exploring alternatives
- Who will use it?
  - They do not demonstrate the upstream design value of this form of design visualization
  - They do not demonstrate the queries downstream users might desire
- Many of the same usability problems as general IBIS

Archium

- Seems promising
- Incorporates design visualization with argumentation visualization
  - Architecture elements are first class entities with rationale
  - Explicitly supports design fragments and design evolution
- Still a ways to go
  - Empirical case studies
  - Tool support
  - Need to prove it can provide downstream value

SEURat

- Argumentation + SE decision ontology
  - Integration of knowledge base with IDE
    - Code elements can be associated with rationale elements
  - Core schemas use generalized argumentation concepts
    - Decision, alternative, claim, assumption, etc.
  - Support for some SE concepts
    - Change request, requirement, etc.
  - Rules describe common SE criteria and allow for inferencing
    - Adaptability, Dependability, Maintainability, Performance, etc.
  - Expert system identifies deficiencies in rationale
- Assumes expert approaches problem rationally

Prescriptive Design Methods

- Designs and design processes are non-rational
  - Naturalistic decision making uses incomplete knowledge and relies on the reuse of expertise
- Goal: methodical, prescriptive approach that relates domain, design, and constraints, reusing design knowledge
  - For a set of known inputs, structure them in some methodical way
  - Evaluate against a criteria, and either iterate or terminate
- Observation:
  - Design rationale is "because the method told me so"
  - Documentation is:
    - Process model (a priori)
    - Input knowledge (method by-product)
    - Intermediate and final models (method by-product)
    - Justification for overriding method where appropriate
**Goal-Oriented KAOS**

- Systematic process for refinement and transformation
  - Each step has defined entry and exit criteria
  - Each step is an ISP with guidance on how to begin and solve common problems
- Intermediate models are used for partial reasoning and evaluation
- Non-Functional goals constrain solutions and are used as evaluation criteria
- Research Question: What kinds of information would need to be stored to justify intuitive leaps?

**CBSP Revisited**

- Transform requirements into architectural elements
  - Refining requirements into allocatable properties is an ISP
  - How can we prescribe requirements refinement?
    - Is there a manageable set of heuristics for each transformation step?
    - Can we document those changes with pseudo-rationale?
- What information would we need to store with our design to capture our design decisions?
  - Input requirements
  - Refined requirements
  - Voting results
  - Dimensions \(\rightarrow\) Properties table
- Additional upstream value
  - Task prioritization
  - Traceability from arch. elements to requirements and back

**Design Maintenance Systems**

- Given a specification, apply transformations to yield a program
  - Transformation trace and stepwise justifications form rationale
  - Functional specification defines functional intent
  - Performance specifications define design constraints
- Upstream value of this process is limited
  - Cost to implement for trivial problems is high
  - Might not scale
- However, process prescriptively handles evolving functional specification
  - And provides change rationale associated with the original derivation

**Reusing Design Knowledge**

- Much of design involves solving the same problems over and over again
- Styles, patterns, idioms, cliches represent solutions to these recurrent patterns
  - They have been selected and refined over time by experts
  - They standardize solution vocabulary
    - Solution patterns can be abstracted to meaningful terms
    - Documentation can be recorded centrally and referred to
  - Identify relationships with other participating elements
    - Can’t identify pattern’s role in larger problem
    - Can express the problem domain in terms of patterns
- Patterns rarely appear unmodified in code, or may be named for domain concepts
  - SPQR—decompose patterns into elemental design patterns and identify patterns by observing localized EDPs in code
  - Identifying cliches through reverse engineering
Reusing Process By-Products

- Relate process by-products to the design context
  - Evolutionary annotations
    - Associate project communications to change logs
  - Technology books
    - Bind code and domain documentation
    - Difficult to query, but creates contextual relationship
    - No need to compile and maintain separate documentation
- Code analysis techniques to infer intent
  - Lackwit: static type inference to understand variable usage
  - Dependency Structure Matrix
    - Can be used to analyze a design's modularity
    - Or understand modularity in an existing program

Explicit Models of Intent

- Code is complex, and inadequate for effectively expressing functional intent
  - Code is a sequence of low-level imperative commands
- Contracts and specifications are descriptive statements of functional intent
  - You would have to read code to find the error conditions
    - That could be easily stated in a single logic sentence
  - Obligations are not local to the code they require them
  - Inscape extends contract-based specifications with obligations
    - A logic for reasoning over semantic interconnections
- Intentional programming and Domain languages
  - Express domain concepts in domain terms and lets the programming system transform intent into the imperative code that implements it

Back Where We Started

- We have a variety of methods for expressing intent (design and functional) to human designers
- How do we express intent to a computer so it can do what we want it to do?
  - Define functional and design intent in formal terms
  - Associate intent to architectural elements
  - Systems can be dynamically reconfigurable on the basis of changing requirements and environment
    - Express new requirement to self-managing system and let it choose a configuration to meet the new needs
  - Software design becomes a problem of the effective expression of intent to a configuration system
    - WSP?

Summary

- We've looked at the major issues covered this semester in the context of:
  - Problem structuring
  - Prescriptive vs. descriptive modeling
  - Opportunistic problem solving
  - Rational vs. naturalistic decision making
- Software design is not mature enough to be rational
  - Reliance on designer experience and knowledge
  - Prescriptive methods in limited use (WHY?)
  - We should consider upstream and downstream value
  - SE domain has a limited number of design transformations and justifications for them - general systems too complex
  - Faking rationality occludes actual design justification
  - Rationality is overrated

’T’he road to Hell is paved with bad intent.”
Credits

- C. Zannier and F. Maurer. Decisions in Agile Design. (Submitted to FSE'06)
- J. Grudin. Evaluating Opportunities for Design Capture
- Everybody’s very fine presentations and All the other papers we’ve covered this semester!