26- Analysis & Process

Rick Adrion

Reminders, etc.

- Schedule a Project#3 review
  - adrion@cs.umass.edu, cooper@cs.umass.edu
  - I am out 12/8-9, 12/16
- Rachel Smith, guest lecture 12/8
- All lecture notes, assignments through 12/1 are posted

Approaches

- Static Analysis
  - Inspections ✓
  - Software metrics ✓
  - Symbolic execution ✓
  - Dependence Analysis
  - Data flow analysis ✓
  - Software Verification ✓
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing ✓
  - Coverage criteria ✓
  - Fault-based testing ✓
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

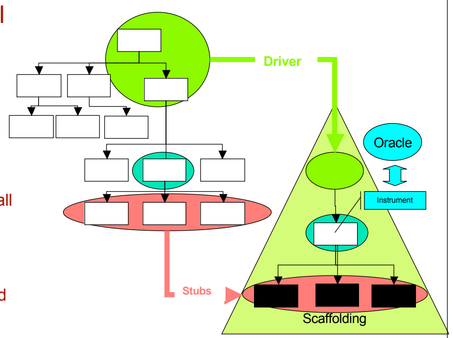
Putting it all together

- unit testing
- integration & system testing
- regression testing

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

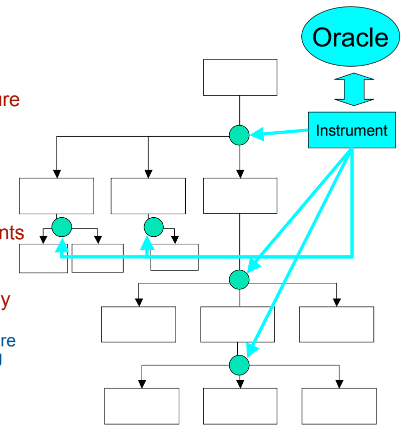
Unit testing

- test scaffolding
  - can be created for general or for specific tests
  - is composed of
    - one or more drivers
      - provide a prototype activation environment
      - drivers initialize non-local variables and parameters and call the unit
    - one or more stubs
      - provide a prototype of the units used by the program to be tested
    - one or more oracles
      - identify the tests that cause failures.

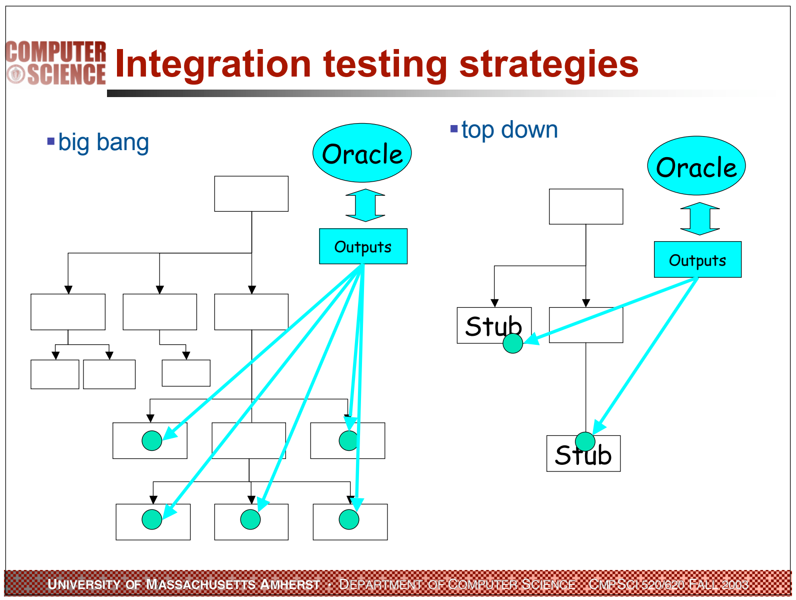


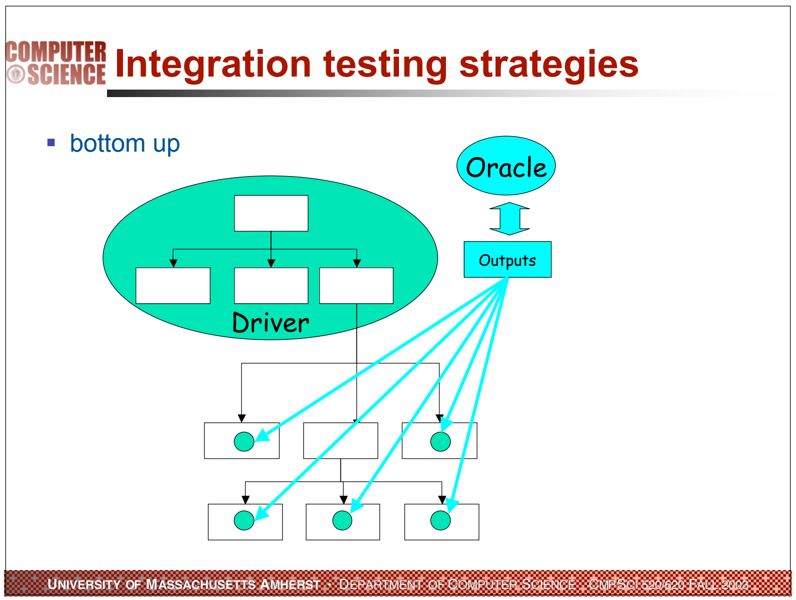
Unit vs. Integration vs. System Testing

- Integration testing
  - focuses on communication and interface problems
  - tests derived from module interfaces and detailed architecture specifications
  - some scaffolding is required
- System testing
  - focuses on the behavior of the system as a whole
  - tests are derived from requirements specifications
  - code is seen as a black box
  - support of scaffoldings not usually needed
    - exception is embedded code, where some simulation of the embedding environment may be required

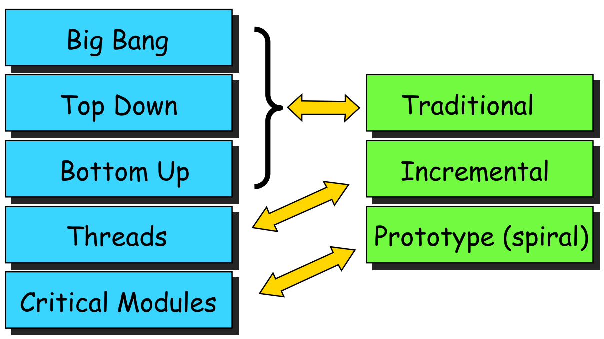


University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003


Integration testing strategies
big bang
Oracle
Outputs
top down
Oracle
Outputs
Stub
Stub
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003


Integration testing strategies
bottom up
Oracle
Outputs
Driver
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Relation to design



O-O Programs are Different

- High Degree of Reuse
  - Does this mean more, or less testing?
- Unit Testing vs. Class Testing
  - What is the right “unit” in OO testing?
- Review of Analysis & Design
  - Classes appear early, so defects can be recognized early as well

Testing OOA and OOD Models

- Correctness (of each model element)
  - Syntactic (notation, conventions)
    - review by modeling experts
  - Semantic (conforms to real problem)
    - review by domain experts
- Consistency (of each class)
  - Revisit Class Diagram
  - Trace delegated responsibilities
  - Examine / adjust cohesion of responsibilities
- Evaluating the Design
  - Compare behavioral model to class model
  - Compare behavioral & class models to the use cases
  - Inspect the detailed design for each class (algorithms & data structures)

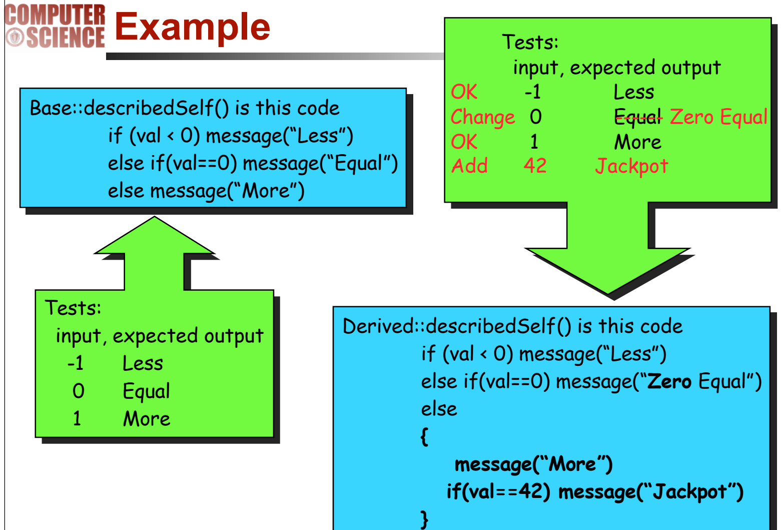
Unit Testing

- What is a “Unit”?
  - Traditional: a “single operation”
  - O-O: encapsulated data & operations
- Smallest testable unit = class
  many operations
- Inheritance
  - testing “in isolation” is impossible
  - operations must be tested every place they are used

Issues in O-O testing

- Need to re-examine all testing techniques and processes
  - Primary Issues:
    - implications of encapsulation
    - implications of inheritance
    - implications of genericity
    - implications of polymorphism
  - Changes concerns
    - State of instance variables
    - Sequences of methods calls
    - Must test a class and its specializations

Example



White-box vs. Black-box Testing

- The distance between object-oriented specification and implementation is typically small
  - gap (and therefore usefulness) of the white-box/black-box distinction is decreasing
- But object-oriented specification describes functional behavior, while the implementation describes how that is achieved
- These techniques can be adapted to method testing, but are not sufficient for class testing
- Conventional flow-graph approaches
  - may be inconsistent the object-oriented paradigm
  - method-level control faults are not likely

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Black-box O-O Testing

- Conventional black-box methods are useful for object-oriented systems
  - error-guessing strategies
  - verification of ADTs can be adapted to object-oriented systems
- Other approaches
  - utilize specifications integrated with the implementation as assertions
    - specification integrated with the implementation as dynamic assertions
    - C++ assertions or Eiffel pre/post-conditions offer similar self-checking
  - Utilize method (event) sequence information
    - usually don't have history of method invocations so can't do this with assertions

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Encapsulation

- not a source of errors but may be an obstacle to testing
- how to get at the concrete state of an object?
- use the abstraction
  - state is inspected via access methods
  - equivalence scenarios
    - comparing sequences of events
    - state is implicitly inspected by comparing related behavior
  - examine sequences of events
    - need to be able to define what are equivalent sequences and need to determine equal states
- use or provide hidden functions to examine the state
  - useful for debugging throughout the life of the system
    - but modified code, may alter the behavior
    - especially true for languages that do not support strong typing
- proof-of-correctness techniques
  - a proved method could be excused from testing to bootstrap testing of other methods
  - state reporting methods tend to be small and simple, they should be relatively easy to prove

Implications of Inheritance

- rule rather than the exception?
- inherited features usually require re-testing
  - because a new context of usage results when features are inherited
  - multiple inheritance increases the number of contexts to test
- specialization relationships
  - implementation specialization should correspond to problem domain specialization
  - reusability of superclass test cases depends on this

University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

Which fns must be tested

Base class contains:
inherited(int x)
redefined() - returns a number in range 1 to 10 inclusive

Derived class contains:
redefined() - returns a number in range 0 to20 inclusive
//inherited() is inherited

- derived::redefined has to be tested afresh
- does derived::inherited() have to be retested?

```
inherited contains the code:
        if (x<0)
            x = x/redefined()
        return x
```

have to test when x<0, could divide by 0

- derived::inherited() may not have to be completely tested
  - if code in inherited() doesn't depend on redefined(), doesn't call it nor call any code that indirectly calls it

Inheritance Testing

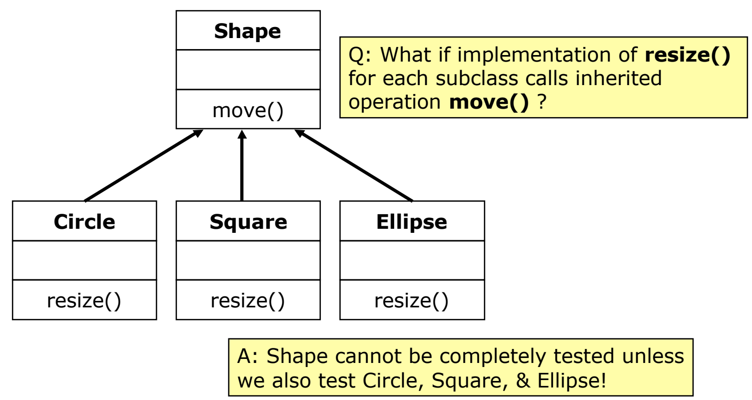
- flattening inheritance
  - each subclass is tested as if all inherited features were newly defined
  - tests used in the super-classes can be reused
  - many tests are redundant
- incremental testing
  - reduce tests only to new/modified features
  - determining what needs to be tested requires automated support

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Polymorphism

- in procedural programming, procedure calls are statically bound
- each possible binding of a polymorphic component requires a separate set of test cases
  - many server classes may need to be integrated before a client class can be tested
- may be hard to determine all such bindings
- complicates integration planning and testing

Testing under Inheritance



Integration Testing

- O-O Integration: Not Hierarchical
  - Coupling is not via subroutine
  - "Top-down" and "Bottom-up" have little meaning
- Integrating one operation at a time is difficult
  - Indirect interactions among operations

O-O Integration Testing

- Thread-Based Testing
  - Integrate set of classes required to respond to one input or event
  - Integrate one thread at a time
  - Example: Event-Dispatching Thread vs. Event Handlers in Java
    - Implement & test all GUI events first
    - Add event handlers one at a time
- Use-Based Testing
  - Implement & test independent classes first
  - Then implement dependent classes (layer by layer, or cluster-based)
  - Simple driver classes or methods sometimes required to test lower layers

Test Case Design

- Focus: “Designing sequences of operations to exercise the states of a class instance”
- Challenges
  - Observability - Do we have methods that allow us to inspect the inner state of an object?
  - Inheritance - Can test cases for a superclass be used to test a subclass?
- Test Case Checklist
  - Identify unique tests & associate with a particular class
  - Describe purpose of the test
  - Develop list of testing steps:
    - Specified states to be tested
    - Operations (methods) to be tested
    - Exceptions that might occur
    - External conditions & changes thereto
    - Supplemental information (if needed)

Software Processes

- Software processes are:
  - the set of activities, methods, and practices that are used in the production and evolution of software
  - **devices** for creating, modifying, analyzing, understanding software artifacts and products

➲ **Hypothesis: Processes are software**

- Improve quality by improving processes
  - Build in quality in, don’t “test in” quality (manufacturing)
  - Use processes to manage complex activities
  - Many observed “process errors”
- Proposed approach
  - Use computers to help perform processes
  - Analyze processes to determine and eliminate defects
  - Use demonstrably superior processes to identify risks, mitigate their consequences, demonstrate quality

Software Process as Software

- Software processes should be developed using a (Software development process) development process
  - Process Requirements
    - Key to designing suitable process
    - Basis for evaluation and improvement of process
  - Process Specification/Modeling/Design
    - Helps conceptualization, communication, visualization
    - Can be management aid
  - Process Code
    - Provides rigor and complete details
    - Basis for execution/tool support and integration

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

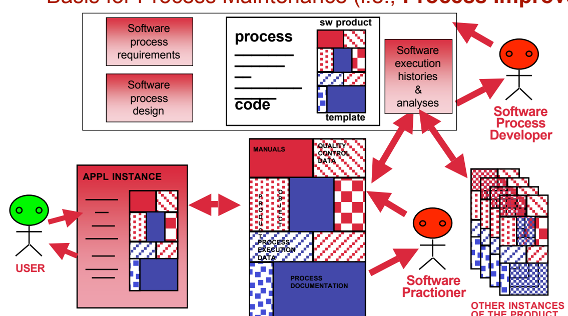
Software Process Code

- Provides details and elaborations upon process design
- Tries to include details omitted from model/design
- Supports more detailed, precise, definitive reasoning
- Vehicle for meshing process control with product data at arbitrarily low levels of detail
- Provides superior visibility enabling better control
- Basis for better predictability
- Basis for process enaction/execution
- Blueprint for tool integration

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Software Process as Software

- Software processes should be developed using a (Software development process) development process
  - Process Measurements and Evaluation
  - Results of Static Analysis and Dynamic Measurement ⇒ Basis for Process Maintenance (i.e., **Process Improvement**)



Software Process Formalisms

- Techniques
  - Languages
    - procedural
    - rule-based
    - object-oriented
  - Modeling formalisms
    - Data flow diagrams
    - Petri Nets
    - Flow graphs
- Key considerations
  - concurrency
  - exception handling
  - resource specification
  - self-modification/long lifetime
  - constraint management
  - artifact specification/management
  - real-time
  - visualizability

Language-based Formalisms

- More traditional coding languages:
  - Procedural (Sutton's Appl/A)
  - Rule-based (Kaiser's Marvel)
  - Functional Hierarchy (Katayama's HFSP)
  - Law based (Minsky)
  - Object Oriented (schema definition languages)
- Key issue: developing abstractions to facilitate process definition

HFSP design model

(a) JSD(Real_World | Design_Spec) =>
(1)Develop_Spec(Real_World_Desc |System_Spec_Diagram)
(2)Develop_Impl(System_Spec_Diagram |System_Impl_Diagram)
(3)Where Real_World_Desc = Interview(Users, Developers,Real_World)
(4) Design_Spec = union(System_Spec_Diagram, System_Impl_Diagram)

Second_level

(b) Develop_Spec(Real_World_Desc |System_Spec_Diagram) =>
(1)Develop_System_Model(Real_World_Desc |Real_World_Model, Init_System_Spec_Diagram)
(2)Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram |System_Spec_Diagram)

Third_level

(c) Develop_System_Model(Real_World_Desc |Real_World_Model, Init_System_Spec_Diagram) =>
(1)Model_Reality(Real_World_Desc |Real_World_Model)
(2)Model_System(Real_World_Model |Init_System_Spec_Diagram)

(d) Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram |System_Spec_Diagram)
(1)Define_Func(Real_World_Model, Init_System_Spec_Diagram |System_Function, Function_Process)
(2)Define_Timing(Init_System_Spec_Diagram, System_Function |Timing)
(3)Where System_Spec_Diagram = is_composed_of(Init_System_Spec_Diagram, System_Function, Function_Process, Timing)

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE CMPSCI 520/620 FALL 2003

HFSP design model

(a) JSD(Real_World | Design_Spec) =>
(1)Develop_Spec(Real_World_Desc |System_Spec_Diagram)

Second_level

(b) Develop_Spec(Real_World_Desc |System_Spec_Diagram) =>
(1)Develop_System_Model(Real_World_Desc |Real_World_Model, Init_System_Spec_Diagram)

Third_level

(c) Develop_System_Model(Real_World_Desc |Real_World_Model, Init_System_Spec_Diagram) =>
(1)Model_Reality(Real_World_Desc |Real_World_Model)
(2)Model_System(Real_World_Model |Init_System_Spec_Diagram)

Fourth_level

(e)Model_Reality(Real_World_Desc | Real_World_Model) =>
(1)Identify_Entity_Action(Real_World_Desc | Entity_Action_List)
(2)Draw_Entity_Structure(Entity_Action_List | Entity_Structure_List)
Where Real_World_Model = is(Entity_Structure_List)
Real_World_Process = is(Entity_Structure)

(f) Model_System(Real_World_Model | Init_System_Spec_Diagram) =>
(1)Identify_Model_Process(Real_World_Model | M_Proc_Name_List)
(2)Connect(Real_World_Model, M_Proc_Name_List | Connection_List)
(3)Specify_Model_Process(Connection_List, Real_World_Model, M_Proc_Name_List |Model_Process_List)
(4)Where Init_System_Spec_Diagram = is(Model_Process_List)
(5)Connection = is(State_Vector) or is(Data_Stream)

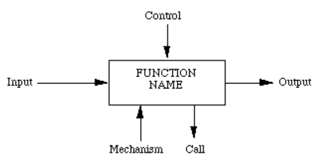
Software Process Formalisms

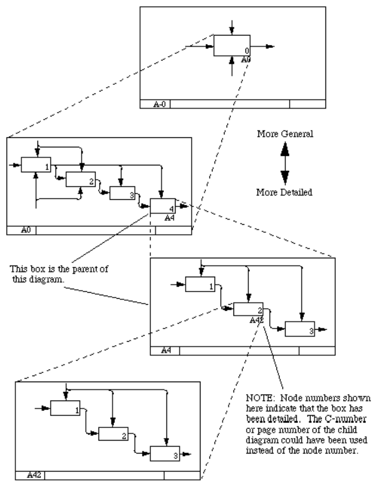
- Techniques
  - Languages
    - procedural
    - rule-based
    - object-oriented
  - Modeling formalisms
    - Data flow diagrams
    - Petri Nets
    - Flow graphs

COMPUTER SCIENCE

IDEF 0

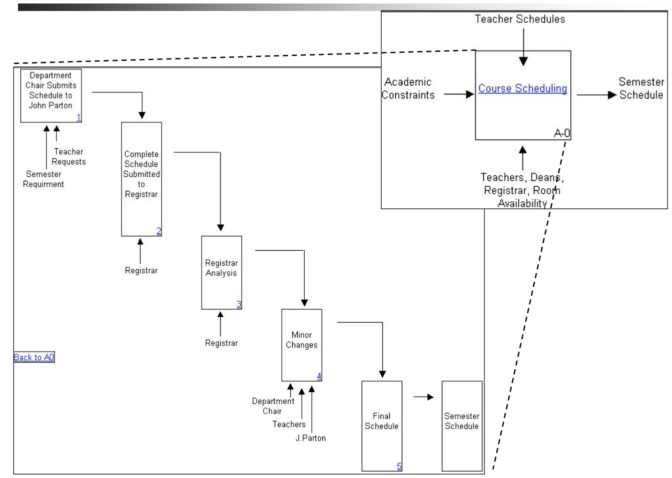
- Integration definition for function modeling
  - Adapted from SADT
  - Very widespread usage industry
  - Overburdened DFD
  - Overly constrained visualization





COMPUTER SCIENCE

IDEF 0 Course Scheduling



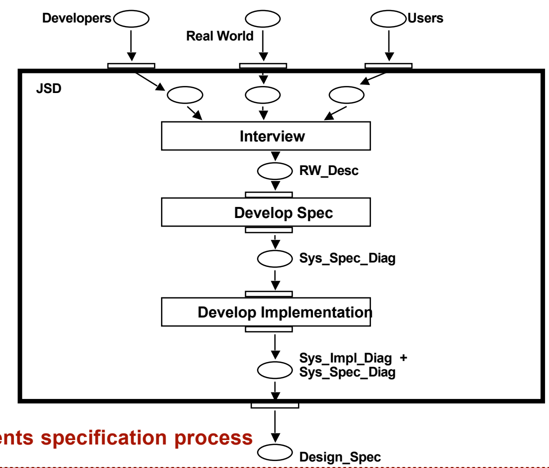
Other "DFD"'s

- Many different adaptations of the basic idea
- Add control flow in
- Add various annotations on
- Add timing information
- Etc.

Petri Net-like representations

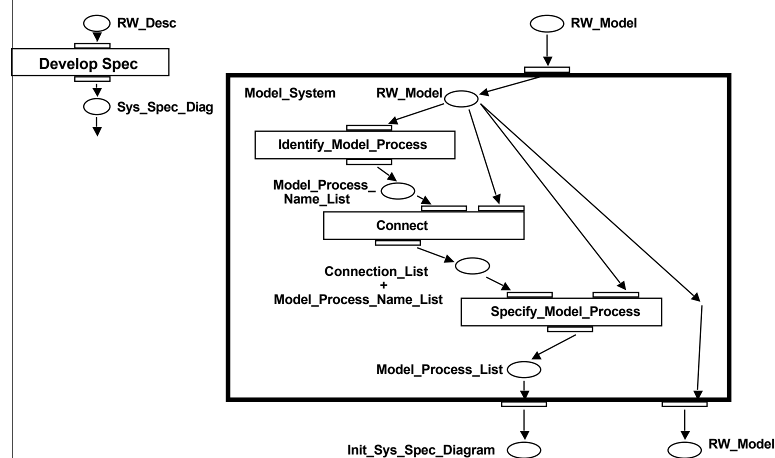
- Particularly effective for showing concurrency in processes
- Weak in dealing with artifacts
- Weak in dealing with exception flow

Petri net-like formalisms

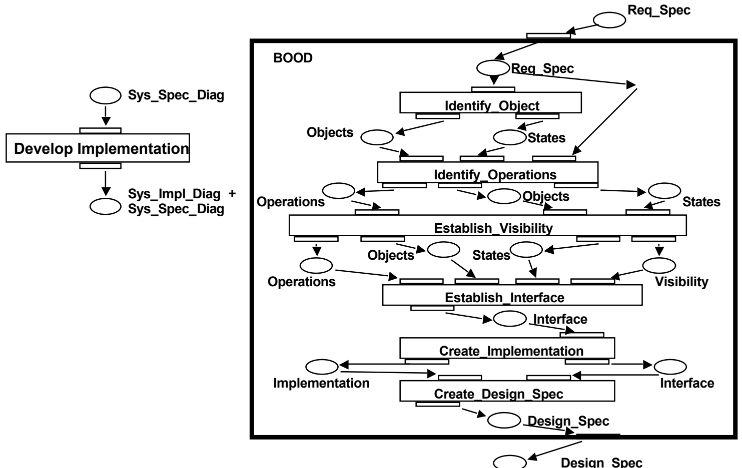


Requirements specification process

Decomposition



Design Process Petri net



Little-JIL

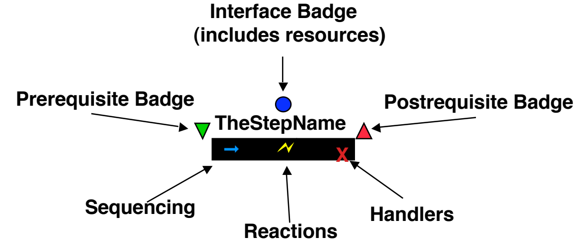
- Little-JIL
  - an agent coordination language. Programs describe the co-ordination and communication among agents that enables them to perform the process.
  - an executable, high-level language with formal yet graphical syntax and
- Hypothesis:
  - Co-ordination structure is separable from other process language issues.
  - Processes are executed by agents that know how to perform their tasks but benefit from co-ordination support.
- Design Principles
  - Simplicity
  - Expressiveness
  - Precision

Little JIL

- Features
  - Explicit agent specifications
  - Explicit resource specification
  - Agent communication via agendas
  - Visualization
  - Proactive and reactive control constructs
  - Explicit data flow
  - Precondition and postcondition guards
- Coordination Paradigm
  - Coordination is the process of building of program by gluing together active pieces and is a vehicle for building programs that can include "human and software processes".
  - Collection of agents, communication mechanism, distribution mechanism, assignment of tasks to agents.

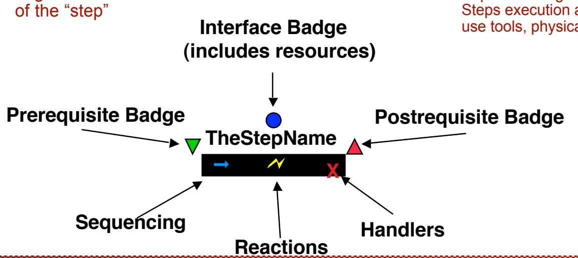
A "step" of Little-JIL

- provides scoping mechanism for control, data & exception flow, and for agent and resource assignment.
- organized into static hierarchy, but can have a highly dynamic execution structure including the possibility of recursion and concurrency.
- is a specification of a unit of work that is assigned to an agent. "unit of encapsulation"



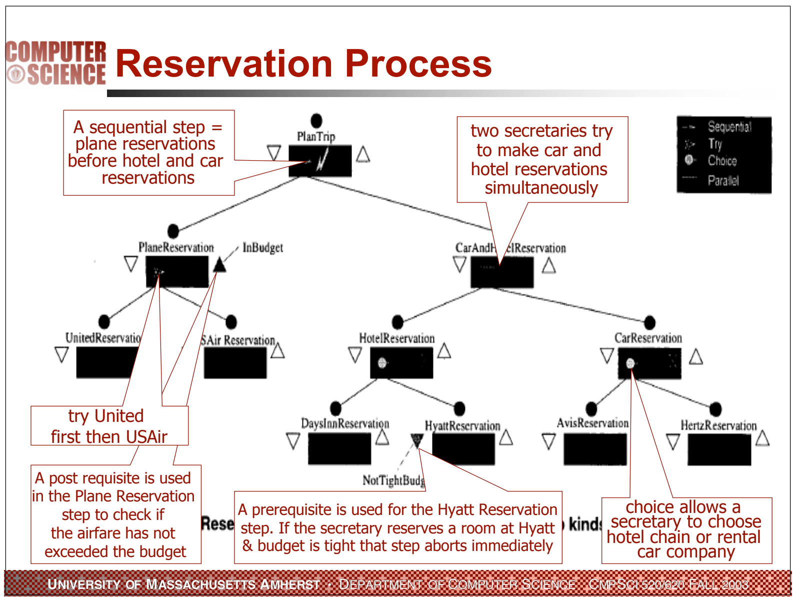
Little-JIL step's Badge

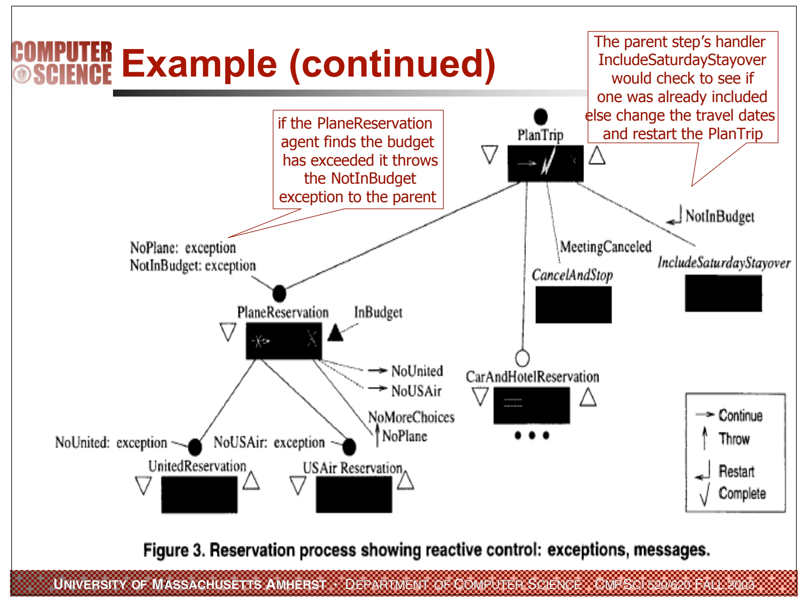
- Control flow :
  - 4 non-leaf steps
  - sequential, parallel, try, choice
- Requisites :
  - Mechanism to add checks before and after a "step" is executed
  - pre-requisite, post-requisite
- Exception & handlers:
  - augment the control flow construct of the "step"
- Messages & Reactions:
  - reactive power and expressive power
- Parameters:
  - passed between steps allow communication of information necessary for the execution of a step and for the return of step execution results.
- Resources:
  - are representations of entities that are required during step's execution e.g.. Steps execution agents, permission to use tools, physical artifacts



Example

- The process involves 4 people:
  - Traveler; Travel agent; Two secretaries
- "Rules"
  - We try United first then USAir
  - If the traveler has gone over budget, and a Saturday stay over was not included, the dates should be changed to include a Saturday stay over and other attempts should be made.
  - After the airline reservations are made and the travel date and times are set, car and hotel reservations should be made.
  - The hotel reservations may either at Days Inn or, if budget is not tight, a Hyatt.
  - The car reservations may be made with either Avis or Hertz.


COMPUTER SCIENCE
Reservation Process
A sequential step = plane reservations before hotel and car reservations
PlanTrip
two secretaries try to make car and hotel reservations simultaneously
Sequential
Try
Choice
Parallel
PlaneReservation
InBudget
UnitedReservation
HotelReservation
CarReservation
DaysInnReservation
HyattReservation
AvisReservation
HertzReservation
try United first then USAir
A post requisite is used in the Plane Reservation step to check if the airfare has not exceeded the budget
A prerequisite is used for the Hyatt Reservation step. If the secretary reserves a room at Hyatt & budget is tight that step aborts immediately
choice allows a secretary to choose hotel chain or rental car company
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003


COMPUTER SCIENCE
Example (continued)
The parent step's handler IncludeSaturdayStayover would check to see if one was already included else change the travel dates and restart the PlanTrip
if the PlaneReservation agent finds the budget has exceeded it throws the NotInBudget exception to the parent
PlanTrip
NotInBudget
MeetingCanceled
IncludeSaturdayStayover
CancelAndStop
NoPlane: exception
NotInBudget: exception
PlaneReservation
InBudget
NoUnited
NoUSAir
NoMoreChoices
NoPlane
CarAndHotelReservation
NoUnited: exception
NoUSAir: exception
UnitedReservation
USAir Reservation
Continue
Throw
Restart
Complete
Figure 3. Reservation process showing reactive control: exceptions, messages.
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

More

- If there was already a Saturday stay over, the handler could throw another exception that would go higher up the tree or terminate the process.
- Different continuation badges would create different executions.
- For example, if Include SaturdayStayover were to rewritten to make alternate plans, then the continuation badge would be changed to “complete” indicating that the exception step had provided an alternate implementation of PlanTrip.

Data flow

- 3 parameter passing modes defined in little JIL.
- Arrows attached to the parameters indicate whether a parameter is copied into the sub steps scope from the parent, copied out or both.

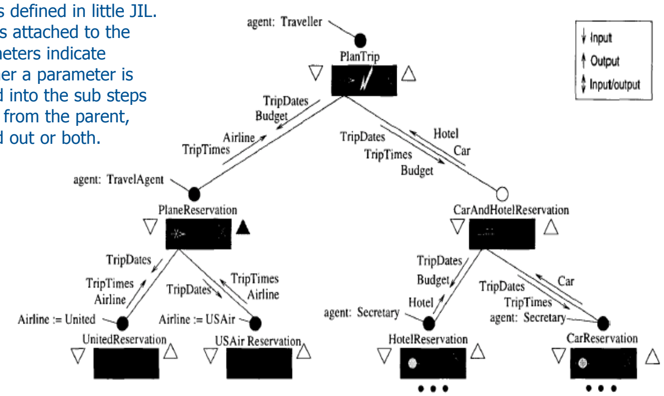


Figure 4. Reservation process showing data flow.

Little-JIL

- Advantages
  - Semantically rich and yet easy to use
  - Formal yet graphical syntax
  - Independent agent can benefit from the coordination with other agents
  - Flexibility to operate/level of details
  - Resource bounded recursion and parallelism
- Disadvantages
  - No data type model for parameters and resources
  - Omits expressions and commands
  - Relies of the agents to know how the tasks represents by leaf steps are performed
  - Specifies coordination and not execution, computation

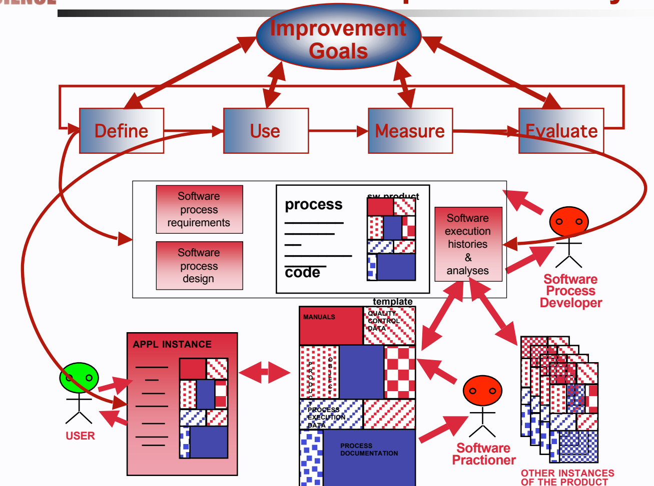
Juliette: The Little-JIL Interpreter

- Powerful substrate required in order to execute Little-JIL
- Architecture of Juliette is distributed
- Components include:
  - Step Interpreter (one for each step)
  - Object Management
  - Resource Management
  - Constraint Management
  - Agenda Management
  - Scheduler

Software Process

- Measurement and Evaluation
  - analogy to application software measurement and evaluation
  - dynamic monitoring of process execution is analogous to interactive debugging of application software
- Process Maintenance
  - takes place over an extended period of time
  - can be expected to be more costly and important than process development
- Process Improvement
  - aimed at progress towards process requirements and improvement goals
  - progress must be measured to assure progress is made and improvement is underway
- All of these argue for
  - process requirements specification and precise process measurement
  - greater rigor that can lead to more effective improvement

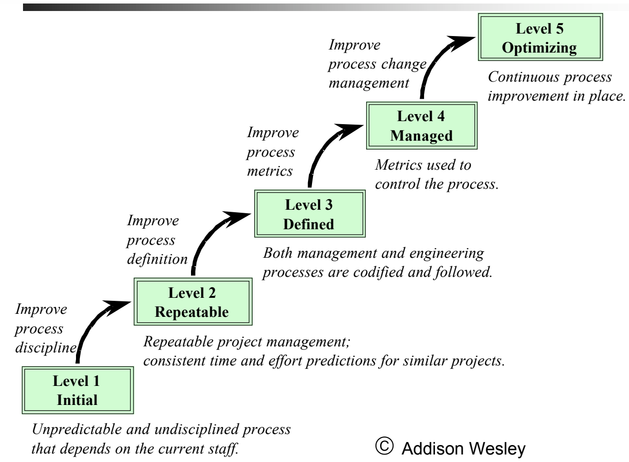
Software Process Improvement Cycle



Capability Maturity Model (CMM)

- Specific Approach to Software Process Improvement
  - model the effectiveness of organizations in developing software
  - developed and promulgated by Watts Humphrey at the CMU Software Engineering Institute
  - based on work on industrial statistical process control by Deming and Juran (decades ago)
- Hypothesizes a "normative model" of how software should be developed, using a comprehensive profile of activity areas
- Hypothesizes five levels of process maturity

CMM



© Addison Wesley

CMM Attempts to Evaluate Predictability

- Highly mature processes are those that offer assurance of predictable results
- Highest levels of process maturity also demonstrably offer expectation of continuous process improvement
- Higher maturity seems easiest to attain when software development is in a restricted domain

CMM and Process Formalisms

- Greater rigor and formality in the specification of the CMM can reduce confusion and ambiguity
  - Use of natural language (English) is always problematical
    - ambiguous, imprecise, incomplete
  - Software formalisms address these problems, e.g.,
    - requirements specification formalisms to make CMM more rigorous
    - testing formalisms and notations to solidify the acceptance testing processes implied by the Software Capability Evaluation (SCE)
- Developing Software Processes that Earn Superior CMM Evaluations
  - CMM does not offer any guidance on how to develop superior processes or on how to improve current processes
  - Process modeling and process coding techniques can be used to materialize the process.
  - Tangible process representation can be studied, analyzed evaluated using computer science techniques
  - Tangible processes can be used as solid bases for demonstrable improvement

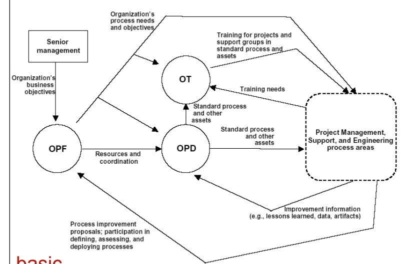
CMM Integration Models

- The CMMI Product Suite
  - includes multiple models and associated training and appraisal materials
  - content from bodies of knowledge (e.g., systems engineering, software engineering, IPPD)
  - helps set process improvement objectives and priorities, improve processes, and provide guidance for ensuring stable, capable, and mature processes..
- Four Categories of CMMI Process Areas
  - Process Management
  - Project Management
  - Engineering
  - Support

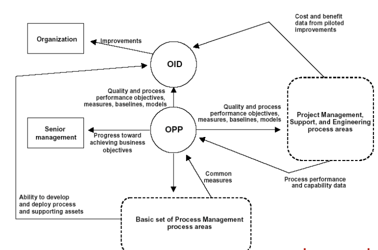
UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

Process Management examples

- Process Management process areas of CMMI are as follows:
  - Organizational Process Focus
  - Organizational Process Definition
  - Organizational Training
  - Organizational Process Performance
  - Organizational Innovation and Deployment



basic



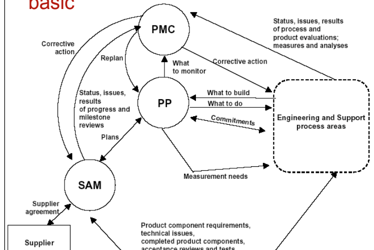
advanced

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

Product Management Examples

- The Project Management process areas of CMMI are as follows:
  - Project Planning
  - Project Monitoring and Control
  - Supplier Agreement Management
  - Integrated Project Management for IPPD
  - Risk Management
  - Integrated Teaming
  - Integrated Supplier Management
  - Quantitative Project Management

basic



advanced