CASE Tools and Software Engineering


Methodology Papers & Texts


Copyright © 1991 by Foresite Systems
Assessing Object-Oriented Technology:  
A Manager’s Perspective

Outline

☞ Understand the technology behind object-oriented stuff
☞ Understand your own development environment
☞ Understand the potential benefits and drawbacks to using object-oriented technology
☞ Determine if object-oriented technology might help and where it could be best applied
☞ Identify potential object-oriented products/techniques
☞ Sell the organization on object-oriented technology
☞ Develop and implement a transition plan for object-oriented technology
☞ Questions

What is an object?

Object: something that is or is capable of being seen, touched or otherwise sensed; something physical or mental of which a subject is cognitively aware; a thing that forms an element of or constitutes the subject matter of an investigation or science; a noun or noun equivalent denoting in a verb constructions that on or toward which the action of a verb is directed.

From Webster’s Dictionary

Examples of objects and attributes:

Airplane: speed, altitude, weight, # of passengers
Automobile: speed, weight, # of doors
Telephone: last # dialed, stored #’s (1-10)
Window (GUI): location, size, handles

Objects have characteristics (attributes), communicate with other objects (messages) and have predictable behaviors (methods). An instance of an object exists when the object possesses all the attributes of a specific thing.

Grady Booch on identifying objects:

“Read the specification of the software you want to build. Underline the verbs if you are after procedural code, the nouns if you aim for an object-oriented program.”

A notation for Objects

Copyright © 1991 by Foresite Systems
Assessing Object-Oriented Technology:  
* A Manager's Perspective

Outline
- Understand the technology behind object-oriented stuff
- Understand your own development environment
- Understand the potential benefits and drawbacks to using object-oriented technology
- Determine if object-oriented technology might help and where it could be best applied
- Identify potential object-oriented products/techniques
- Sell the organization on object-oriented technology
- Develop and implement a transition plan for object-oriented technology
- Questions

---

What is an object?

Object: something that is or is capable of being seen, touched or otherwise sensed; something physical or mental of which a subject is cognitively aware; a thing that forms an element of or constitutes the subject matter of an investigation or science; a noun or noun equivalent denoting in a verb constructions that on or toward which the action of a verb is directed.

*From Webster's Dictionary*

Examples of objects and attributes:
- Airplane: speed, altitude, weight, # of passengers
- Automobile: speed, weight, # of doors
- Telephone: last # dialed, stored #’s (1-10)
- Window (GUI): location, size, handles

Objects have characteristics (attributes), communicate with other objects (messages) and have predictable behaviors (methods). An instance of an object exists when the object possesses all the attributes of a specific thing.

Grady Booch on identifying objects:

"Read the specification of the software you want to build. Underline the verbs if you are after procedural code, the nouns if you aim for an object-oriented program."

---

A notation for Objects

```
Object
    Message Centers
    Data
    Methods
```

Copyright © 1991 by Foresite Systems
Examples of Objects

Copyright © 1991 by Foresite Systems

History of object-oriented technology

1967 Simula-67 is developed at the Norwegian Computing Centre by Kristen Nygaard and Ole-Johan Dahl from Algol

1972 Smalltalk is written at the Xerox PARC by Dan Ingalls and Smalltalk-72 is the first version of smalltalk available; Smalltalk was originally aimed at teaching elementary school students how to program

1976 MIT develops Flavors, a set of object-oriented extensions for Lisp

1980 The Xerox PARC defines Smalltalk-80 and licenses it to Apple, DEC, HP and Tektronix

1984 Brad Cox develops Objective-C

1985 Digitalk ships Methods, a PC-based Smalltalk language, Bjarne Stroustrup of AT&T defines C++, and Apple Computer begins shipping Object Pascal and MacApp for the Macintosh

1986 The first OO conference, OOPSLA, is held and Interactive Software Engineering delivers Eiffel

1987 Digitalk delivers Smalltalk/V for the PC and Parc Place Systems releases Smalltalk-80 for the Macintosh

1989 Borland introduces Object Pascal for the PC, Apple releases C++ for the Macintosh and the Next computer ships
History of object-oriented technology

1967 Simula-67 is developed at the Norwegian Computing Centre by Kristen Nygaard and Ole-Johan Dahl from Algol

1972 Smalltalk is written at the Xerox PARC by Dan Ingalls and Smalltalk-72 is the first version of smalltalk available; Smalltalk was originally aimed at teaching elementary school students how to program

1976 MIT develops Flavors, a set of object-oriented extensions for Lisp

1980 The Xerox PARC defines Smalltalk-80 and licenses it to Apple, DEC, HP and Tektronix

1984 Brad Cox develops Objective-C

1985 Digitalk ships Methods, a PC-based Smalltalk language, Bjarne Stroustrup of AT&T defines C++, and Apple Computer begins shipping Object Pascal and MacApp for the Macintosh

1986 The first OO conference, OOPSLA, is held and Interactive Software Engineering delivers Eiffel

1987 Digitalk delivers Smalltalk/V for the PC and Parc Place Systems releases Smalltalk-80 for the Macintosh

1989 Borland introduces Object Pascal for the PC, Apple releases C++ for the Macintosh and the Next computer ships
### Object-oriented terms defined

**Object-orientation**: using abstraction beyond data and procedures by considering the world as a set of entities or objects that are related to and communicate with each other.

**Class**: a group or category of objects that all share the same functions and data.

**Inheritance**: a property that allows child objects to inherit data and functions from parent objects.

**Methods**: procedures that reside in an object and determine how the object will act when it receives messages.

**Message**: a means of communicating between objects.

**Encapsulation**: the process of combining data and function into an object.

**Information hiding**: methods carried out by objects are hidden from other objects. Because of information hiding, objects can be modified or enhanced without changing the way they are used by other objects.

**Abstraction**: the process of considering the world in terms of abstract objects. Different levels of abstraction can exist within the SDLC and for specific projects.

**Polymorphism**: results when a single message causes different actions when received by different objects.

**Persistence**: the permanence of an object or the period of time that space is allocated and available for the object in memory.

**Static vs. Dynamic binding**: the mechanism for associating the address of a called procedure with a caller. Static binding occurs at compile and link time while dynamic binding occurs at run time.

**Class Library**: a collection of generic classes that can be adapted (inherited) and tailored for a new application.

---

### Categories of Object-Oriented Tools

**Object-Oriented Analysis products**
**Object-Oriented Design products**
**Object-Oriented Programming languages**
**Object-Oriented Database Management Systems**
**Object-Oriented Environments**

Comparing OO techniques with traditional structured techniques

<table>
<thead>
<tr>
<th>Traditional techniques</th>
<th>OO techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures, functions or subroutines</td>
<td>Methods</td>
</tr>
<tr>
<td>Data</td>
<td>Instance variables</td>
</tr>
<tr>
<td>Procedure/function calls</td>
<td>Messages</td>
</tr>
<tr>
<td>Abstract data types</td>
<td>Classes</td>
</tr>
<tr>
<td>?</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Calls under program control (Program-driven)</td>
<td>Calls under OS control (Event-driven)</td>
</tr>
</tbody>
</table>
Object-oriented terms defined

Object-orientation: using abstraction beyond data and procedures by considering the world as a set of entities or objects that are related to and communicate with each other.

Class: a group or category of objects that all share the same functions and data.

Inheritance: a property that allows child objects to inherit data and functions from parent objects.

Methods: procedures that reside in an object and determine how the object will act when it receives messages.

Message: a means of communicating between objects.

Encapsulation: the process of combining data and function into an object.

Information hiding: methods carried out by objects are hidden from other objects. Because of information hiding, objects can be modified or enhanced without changing the way they are used by other objects.

Abstraction: the process of considering the world in terms of abstract objects. Different levels of abstraction can exist within the SDLC and for specific projects.

Polymorphism: results when a single message causes different actions when received by different objects.

Persistence: the permanence of an object or the period of time that space is allocated and available for the object in memory.

Static vs. Dynamic binding: the mechanism for associating the address of a called procedure with a caller. Static binding occurs at compile and link time while dynamic binding occurs at run time.

Class Library: a collection of generic classes that can be adapted (inherited) and tailored for a new application.

Categories of Object-Oriented Tools

Object-Oriented Analysis products
Object-Oriented Design products
Object-Oriented Programming languages
Object-Oriented Database Management Systems
Object-Oriented Environments

Comparing OO techniques with traditional structured techniques

<table>
<thead>
<tr>
<th>Traditional techniques</th>
<th>OO techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures, functions or subroutines</td>
<td>Methods</td>
</tr>
<tr>
<td>Data</td>
<td>Instance variables</td>
</tr>
<tr>
<td>Procedure/function calls</td>
<td>Messages</td>
</tr>
<tr>
<td>Abstract data types</td>
<td>Classes</td>
</tr>
<tr>
<td>?</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Calls under program control</td>
<td>Calls under OS control</td>
</tr>
<tr>
<td>(Program-driven)</td>
<td>(Event-driven)</td>
</tr>
</tbody>
</table>
Comparing Objects to traditional subroutines

Objects reflect a deeper level of abstraction
Objects respond only to certain strictly defined messages passed to them
Objects receive messages and take full control of the system until it passes control to some other object via another message
Objects interface to other objects in a clearly defined manner and cannot be subverted

Major Object-Oriented techniques

<table>
<thead>
<tr>
<th>OOA</th>
<th>Shlaer/Mellor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Coad/Yourdon</td>
</tr>
<tr>
<td></td>
<td>Synthesis</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Modeling Technique (OOMT)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OOD</th>
<th>Booch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Buhr</td>
</tr>
<tr>
<td></td>
<td>Synthesis</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Structured Design (OOSD)</td>
</tr>
<tr>
<td></td>
<td>General Object-Oriented Design (GOOD)</td>
</tr>
<tr>
<td></td>
<td>Hierarchical Object-Oriented Design (HOOD)</td>
</tr>
</tbody>
</table>

Getting started with Object-Oriented Technology

**OO Evaluation Questionnaire**

1. What type of hardware & software are you currently developing on (A) and for (B)?
   [Example: A - IBM PC/AT & B - IBM 3091 VM using SQL/DS, A - VAX VMS & B - Same, etc.]
   
<table>
<thead>
<tr>
<th>A:</th>
<th>B:</th>
</tr>
</thead>
</table>

2. List any structured methodologies or object-oriented tools/techniques your company is currently using or plans to use in the future. [Example: Ward/Mellor SA/SD, Hatley/Pirbhai, ESML, etc.]

3. What phases of the Software Development Life Cycle (SDLC) are you initially planning to support using automated tools? [Example: Systems Analysis & Design, all phases, Programming & Testing, etc.]

4. Are you developing in a multi-tasking, GUI or event-driven environment? [Example: Real-Time embedded O/S, XWindows, OS/2 PM, etc.]

5. Will you be attempting to support a group development environment with automated tools in the short term? If so, on what hardware platform? [Example: Yes - on a Local Area Network, Yes - using minicomputers, Yes - using Unix workstations]

6. What programming language will you be developing with? [Example: Assembler, C, Pascal, etc.]
Comparing Objects to traditional subroutines

Objects reflect a deeper level of abstraction
Objects respond only to certain strictly defined messages passed to them
Objects receive messages and take full control of the system until it passes control to some other object via another message
Objects interface to other objects in a clearly defined manner and cannot be subverted

Major Object-Oriented techniques

OOA  
Shlaer/Mellor
Coad/Yourdon
Synthesis
Object-Oriented Modeling Technique (OOMT)

OOD  
Booch
Buhr
Synthesis
Object-Oriented Structured Design (OOSD)
General Object-Oriented Design (GOOD)
Hierarchical Object-Oriented Design (HOOD)

Getting started with Object-Oriented Technology

OO Evaluation Questionnaire

1. What type of hardware & software are you currently developing on (A) and for (B) ?
[Example: A - IBM PC/AT & B - IBM 3091 VM using SQL/DS, A - VAX VMS & B - Same, etc.]

A :
B :

2. List any structured methodologies or object-oriented tools/techniques your company is currently using or plans to use in the future. [Example : Ward/Mellor SA/SD, Hatley/Pirbhai, ESML, etc.]

3. What phases of the Software Development Life Cycle (SDLC) are you initially planning to support using automated tools ? [Example : Systems Analysis & Design, all phases, Programming & Testing, etc.]

4. Are you developing in a multi-tasking, GUI or event-driven environment? [Example : Real-Time embedded O/S, XWindows, OS/2 PM, etc.]

5. Will you be attempting to support a group development environment with automated tools in the short term ? If so, on what hardware platform? [ Example : Yes - on a Local Area Network, Yes - using minicomputers, Yes - using Unix workstations]

6. What programming language will you be developing with? [Example : Assembler, C, Pascal, etc.]
Roger Pressman: A Software Engineering Life Cycle

Roger Pressman has written on the process of selecting and implementing structured methods and CASE tools. In his books, Dr. Pressman introduces a Software Engineering Life Cycle (SELC) for the transition to a CASE environment.

This SELC involves five basic steps:

Assessment: Assess where the organization is in its software development environment and identify strong and weak points in the process. Pressman recommends organizations perform a Software Engineering Audit (SEA) prior to moving forward with CASE.

Education: Educate everyone involved in the methods, techniques and tools to be used or already available.

Selection: Select any new techniques and/or tools that can help improve the software development process.

Installation: Install techniques and tools as selected.

Evaluation: Evaluate whether the software process has been improved by the tools and techniques installed and how the process can be further improved with other tools and techniques.

It is important to recognize that Pressman strongly suggests this SELC is a repetitive process and doesn’t stop. Once methods and tools have been successfully implemented, the need still exists to evaluate and continue education.

Pressman Software Engineering Life Cycle

As part of the Software Engineering Life Cycle (SELC), Pressman recommends a software engineering audit to help determine what the current state of software development is within an organization.

A software engineering audit (SEA) is used to recognize the current software engineering environment, look into strengths and weaknesses, investigate organizational structure and dynamics and serve as a starting point for making recommendations for the next step.

During a Software Engineering Audit, organizations should focus on:

A. Consistency of development philosophy
B. Sophistication of tracking and control mechanisms
C. Acceptance of the current environment
D. Commitment of people involved
E. Politics of the MIS arena
F. Strengths and weaknesses

Example outline for SEA report

1. Software technology - level of technology in use
2. Software development process - methods, procedures, tools, documentation, quality assurance, etc.
3. People - staff turnover, technical skills, experience, etc.
4. CASE tools - any tools in use?, if so, how used?
5. Training - recommendations for a training strategy
Roger Pressman: A Software Engineering Life Cycle

Roger Pressman has written on the process of selecting and implementing structured methods and CASE tools. In his books, Dr. Pressman introduces a Software Engineering Life Cycle (SELC) for the transition to a CASE environment.

This SELC involves five basic steps:

**Assessment:** Assess where the organization is in its software development environment and identify strong and weak points in the process. Pressman recommends organizations perform a Software Engineering Audit (SEA) prior to moving forward with CASE.

**Education:** Educate everyone involved in the methods, techniques and tools to be used or already available.

**Selection:** Select any new techniques and/or tools that can help improve the software development process.

**Installation:** Install techniques and tools as selected.

**Evaluation:** Evaluate whether the software process has been improved by the tools and techniques installed and how the process can be further improved with other tools and techniques.

It is important to recognize that Pressman strongly suggests this SELC is a repetitive process and doesn’t stop. Once methods and tools have been successfully implemented, the need still exists to evaluate and continue education.

---

As part of the Software Engineering Life Cycle (SELC), Pressman recommends a software engineering audit to help determine what the current state of software development is within an organization.

*A software engineering audit (SEA) is used to recognize the current software engineering environment, look into strengths and weaknesses, investigate organizational structure and dynamics and serve as a starting point for making recommendations for the next step.*

During a Software Engineering Audit, organizations should focus on:

A. Consistency of development philosophy
B. Sophistication of tracking and control mechanisms
C. Acceptance of the current environment
D. Commitment of people involved
E. Politics of the MIS arena
F. Strengths and weaknesses

**Example outline for SEA report**

1. Software technology - level of technology in use
2. Software development process - methods, procedures, tools, documentation, quality assurance, etc.
3. People - staff turnover, technical skills, experience, etc.
4. CASE tools - any tools in use?, if so, how used?
5. Training - recommendations for a training strategy
The Software Engineering Institute (SEI) at Carnegie Mellon University perform assessments of DoD subcontractors as part of its software process program. The SEI process assessment helps organizations characterize their software development environment and SEI makes recommendations to facilitate improvement in this area.

Over the past five years, SEI has done assessments on over 30 organizations.

A typical SEI-assisted assessment schedule

**Day One**
- Assessment overview with staff, site manager, etc.
- Briefing - review the assessment schedule, questions and answers and project discussion

**Day Two**
- Functional area interviews with experts in selected technical areas
- Preliminary findings are formulated

**Day Three**
- Project discussions clarifying any items and accepting comment on the preliminary findings
- Formal findings by the assessment team

**Day Four**
- Findings dry run by assessment team
- Findings review with project members
- Findings presentation to staff and site manager
- Senior management meeting to discuss findings and future steps
- Assessment postmortem to identify and provide for any changes to the assessment

** Hewlett Packard has also identified a software process improvement life cycle that consists of the following activities:**

- Define the overall (software development) process and determine whether it is under control
- An assessment team performs an assessment and finds major strengths and weaknesses with the current process and sets a baseline for measurement of future improvements
- A process improvement team collects and analyzes data to determine the most likely cause for problems, and discusses ways of eliminating these problems with the developers
- A process improvement approach is selected and implemented
- It is important to standardize improvements and ensure that change in the process is communicated to everyone involved
The Software Engineering Institute (SEI) at Carnegie Mellon University perform assess­ments of DoD subcontractors as part of its software-process program. The SEI process assessment helps organizations characterize their software development environment and SEI makes recommendations to facilitate improvement in this area.

Over the past five years, SEI has done assessments on over 30 organizations.

A typical SEI-assisted assessment schedule

Day One
Assessment overview with staff, site manager, etc.
Briefing - review the assessment schedule, questions and answers and project discussion

Day Two
Functional area interviews with experts in selected technical areas
Preliminary findings are formulated

Day Three
Project discussions clarifying any items and accepting comment on the preliminary findings
Formal findings by the assessment team

Day Four
Findings dry run by assessment team
Findings review with project members
Findings presentation to staff and site manager
Senior management meeting to discuss findings and future steps
Assessment postmortem to identify and provide for any changes to the assessment

Copyright © 1991 by Foresite Systems

Hewlett Packard has also identified a software process improvement life cycle that consists of the following activities:

Define the overall (software development) process and determine whether it is under control

An assessment team performs an assessment and finds major strengths and weaknesses with the current process and sets a baseline for measurement of future improvements

A process improvement team collects and analyzes data to determine the most likely cause for problems, and discusses ways of eliminating these problems with the developers

A process improvement approach is selected and implemented

It is important to standardize improvements and ensure that change in the process is communicated to everyone involved
Issues Related to Automating Software Development

★ Configuration Management
★ Standards and Procedures
★ Testing and Reviews
★ Quality Assurance
★ Training & Education
★ Project Management

Formalized

★ Written down
★ Measured
★ Rigorously followed
★ Taught
★ Automatable(?)
Issues Related to Automating Software Development

- Configuration Management
- Standards and Procedures
- Testing and Reviews
- Quality Assurance
- Training & Education
- Project Management

Formalized

- Written down
- Measured
- Rigorously followed
- Taught
- Automatable(?)
Configuration Management

Definition: identifying, organizing, and controlling modifications to the software under development. Can include source and executable code, specifications and models.

Functions of Configuration Management

- Controls access to code by more than one programmer (shared access to objects)
- Allows previous versions of code to be recreated
- Provides automatic journaling to document changes

Basic operations in Configuration Management

- Check-in/Check-out - for update or browsing (read-only)
- Storage of multiple versions - deltas (difference between two versions)
- Numbering/naming of versions (1.0, 1.1, 1.2, 2.0, ...)
- Scanning of files for keywords
- Branching (two variations of a set of code) and merging (combinations of more than one variation)
- Histories and journals of code
- Difference generators (diff)

Standards and Procedures

- Naming conventions
- Programming conventions
- Reuse procedures (formal?)

Testing and Reviews

- Unit testing
- Integration testing
- System testing
- Creation/generation/maintenance of test cases
- Monitoring control logic
- Test points and debugging
- Black-box testing
- Regression testing
- Test Reviews
- Structured Walk-throughs
- Operation Reviews

Software Quality Assurance

Software Quality Assurance (SQA) monitors the methods and standards used by software developers and verifies that they have properly applied their expertise.

SQA functions

- Establish and review quality assurance practices
- Evaluate the project plan
- Evaluate the requirements
- Evaluate the design process
- Evaluate coding practices
- Evaluate integration and testing
- Evaluate management processes
- Tailor QA procedures as needed

Training and Education

- Business training
- Technical training
- Development approach
- Techniques and methods
- Standards and procedures
- Tool use

Project Management

- Deliverables throughout the LC
- Metrics
- Estimation
- Collection of statistics
- Staff/resource scheduling
- Reporting
- GANTT, PERT, CPM etc.
Configuration Management

Definition: identifying, organizing, and controlling modifications to the software under development. Can include source and executable code, specifications and models.

Functions of Configuration Management

Controls access to code by more than one programmer (shared access to objects)
Allows previous versions of code to be recreated
Provides automatic journaling to document changes

Basic operations in Configuration Management

Check-in/Check-out - for update or browsing (read-only)
Storage of multiple versions - deltas (difference between two versions)
Numbering/naming of versions (1.0,1.1,1.2,2.0,...)
Scanning of files for keywords
Branching (two variations of a set of code) and merging (combinations of more than one variation)
Histories and journals of code
Difference generators (diff)

Software Quality Assurance

Software Quality Assurance (SQA) monitors the methods and standards used by software developers and verifies that they have properly applied their expertise.

SQA functions

- Establish and review quality assurance practices
- Evaluate the project plan
- Evaluate the requirements
- Evaluate the design process
- Evaluate coding practices
- Evaluate integration and testing
- Evaluate management processes
- Tailor QA procedures as needed

Training and Education

Business training
Technical training
Development approach
Techniques and methods
Standards and procedures
Tool use

Project Management

Deliverables throughout the LC
Metrics
Estimation
Collection of statistics
Staff/resource scheduling
Reporting
GANTT, PERT, CPM etc.
Potential benefits of OO technology

Increased reusability of code, specifications and models because of built-in support for inheritance, information hiding, encapsulation and classes
Increased quality because of reuse of tested code and easy to enhance
Increased productivity based on reuse (quicker development)
Increased predictability because of standard parts reused, less new development required
Complexity controlled through classification, better requirements modeling and separating function from data
Decreased maintenance because code will be more modular with isolated features

Problems with OO technology

Debugging OO systems: because control flow shifts from object to object, debugging is often difficult to follow (message propagation)
OOD can often result in more structurally complex modules than SD
The ripple effect: any change or problem that occurs in a parent object will have to be investigated in all child objects
Supporting concurrency in an OOA/OOD notation or OOPL
Potential benefits of OO technology

Increased reusability of code, specifications and models because of built-in support for inheritance, information hiding, encapsulation and classes
Increased quality because of reuse of tested code and easy to enhance
Increased productivity based on reuse (quicker development)
Increased predictability because of standard parts reused, less new development required
Complexity controlled through classification, better requirements modeling and separating function from data
Decreased maintenance because code will be more modular with isolated features

Problems with OO technology

Debugging OO systems: because control flow shifts from object to object, debugging is often difficult to follow (message propagation)
OOD can often result in more structurally complex modules than SD
The ripple effect: any change or problem that occurs in a parent object will have to be investigated in all child objects
Supporting concurrency in an OOA/OOD notation or OOPL
Real-Time Software Development Life Cycle (SDLC)

Planning & Investigation: As with the traditional SDLC, we need to identify the requirements (generally speaking) and make basic hardware decisions.

Outcomes or deliverables: System Scope document
General System requirements document
Cost Benefits, time & expected impact on external systems
Plan for next phase (resources)

Systems Analysis: Determine the feasibility of the project, model the data, process and control of the system and identify the subsystems. In addition, the interfaces between the subsystems should be modeled.

Outcomes or deliverables: Functional Requirements document
Data, Process & Control Models
Prototypes and simulations
Data Structure definitions

Preparation of proposal(s): If several subsystems are involved, proposals for each should be prepared and presented to management.

Outcomes or deliverables: Proposal
Go/No go decision

Logical Design: Complete the logical model of the system including data/control flow diagrams, entity-relationship diagrams, data definitions and process/control specifications.

Outcomes or deliverables: Data model (Completed)
Process & Control models (Completed)
Module Description documents

Analyze specification: Verify that the diagrams created above are complete and consistent.

Outcomes or deliverables: Verification of the design
Performance & machine testing information

Physical Design: Develop a physical model and allocate the processes to processors, develop the structure of the system (structure charts) and develop any additional hardware/software interfaces that are required.

Outcomes or deliverables: Module Architecture document
Module Description documents
Block diagrams (Optional)

Programming: Create and test the programs that will meet the physical design created above.

Outcomes or deliverables: The actual program code (source & executable)
Data structures and/or table definitions

Maintenance: Keep the production system running through its useful lifetime.

Outcomes or deliverables: Impact analysis reports
Cost-benefits analysis document
Modification schedule/plan
Real-Time Software Development Life Cycle (SDLC)

Planning & Investigation: As with the traditional SDLC, we need to identify the requirements (generally speaking) and make basic hardware decisions.

Outcomes or deliverables: System Scope document
General System requirements document
Cost Benefits, time & expected impact
on external systems
Plan for next phase (resources)

Systems Analysis: Determine the feasibility of the project, model the data, process and control of the system and identify the subsystems. In addition, the interfaces between the subsystems should be modeled.

Outcomes or deliverables: Functional Requirements document
Data, Process & Control Models
Prototypes and simulations
Data Structure definitions

Preparation of proposal(s): If several subsystems are involved, proposals for each should be prepared and presented to management.

Outcomes or deliverables: Proposal
Go/No go decision

Logical Design: Complete the logical model of the system including data/control flow diagrams, entity-relationship diagrams, data definitions and process/control specifications.

Outcomes or deliverables: Data model (Completed)
Process & Control models (Completed)
Module Description documents

Analyze specification: Verify that the diagrams created above are complete and consistent.

Outcomes or deliverables: Verification of the design
Performance & machine testing information

Physical Design: Develop a physical model and allocate the processes to processors, develop the structure of the system (structure charts) and develop any additional hardware/software interfaces that are required.

Outcomes or deliverables: Module Architecture document
Module Description documents
Block diagrams (Optional)

Copyright © 1991 by Foresite Systems

Programming: Create and test the programs that will meet the physical design created above.

Outcomes or deliverables: The actual program code (source & executable)
Data structures and/or table definitions

Maintenance: Keep the production system running through its useful lifetime.

Outcomes or deliverables: Impact analysis reports
Cost-benefits analysis document
Modification schedule/plan
Object-orientation in the SDLC

**OO Analysis:** the process of analyzing or modeling the requirements of a system with objects. Traditional OOA identifies system components (objects), their relationships and behaviors.

**OOA Techniques**

Shlaer & Mellor, Project Technologies, Inc. from their book, *Object-Oriented Systems Analysis: Modeling the World in Data*, defined the following OOA/OOD technique:

**Systems Analysis phase**

1. Start by building an information model that identifies the objects, relationships and attributes, along with the multiplicity and conditionality specifications. Start at the 'center' of the problem domain, and work until all objects have been identified.

Output: Information Structure Diagram (ISD)

2. For each active object in the ISD, create a state model that contains all the potential states in the object's life cycle and the events that cause transitions from one state to another.

Output: State transition diagrams (STDs)

(Optional) State Transition tables/matrices

3. Create an Object Communication Diagram (OCD) that illustrates all the object state models and communication between objects.

Output: Object Communication Diagram (OCDs)

4. For each state transition in the STDs, create a Data Flow diagram and describe the processing that causes the state to change. Processes in the DFDs can create events that appear in the STDs.

Output: Data Flow diagrams (DFDs)

5. Generate reports from the diagrams in the form of an Objects/Attributes list, an event list and a relationship list. Each object in the ISD becomes a data store in a DFD.

**External Specification phase**

1. State the boundaries of the system - the imaginary line between the abstract system and the portion outside the system. This distinction should be based on external events that cross this boundary.

2. Create an external event list

3. Create a functional requirements document and back this document up with the abstract system models created in the Systems Analysis phase. This document can also contain screen & report layouts, descriptions of interfaces between the system and external systems and operator procedures.

**Model Integration**

STDs are built for objects that exhibit a life cycle or operational cycle

STDs cause actions to take place which are represented as processes in a DFD

Each object in the ERD becomes a data store in a DFD

Processes accept and produce only the data which is defined in the ERD

Process may create events which appear in STDs

**Systems Design phase**

1. Design the system architecture
2. Design the system information content
3. Design data structures
4. Partition the system into programs

**Implementation phase**

1. Collect the data (identify sources, develop entry procedures, prepare data structures, enter the data and verify that it is correctly entered)
2. Design programs
3. Code and test the programs
4. Integrate the system
Object-orientation in the SDLC

**OO Analysis:** the process of analyzing or modeling the requirements of a system with objects. Traditional OOA identifies system components (objects), their relationships and behaviors.

**OOA Techniques**

Shlaer & Mellor, Project Technologies, Inc. from their book, *Object-Oriented Systems Analysis: Modeling the World in Data*, defined the following OOA/OOD technique:

**Systems Analysis phase**

1. Start by building an information model that identifies the objects, relationships and attributes, along with the multiplicity and conditionality specifications. Start at the 'center' of the problem domain, and work until all objects have been identified.

   Output: Information Structure Diagram (ISD)

2. For each active object in the ISD, create a state model that contains all the potential states in the object's life cycle and the events that cause transitions from one state to another.

   Output: State transition diagrams (STDs)

   (Optional) State Transition tables/matrices

3. Create an Object Communication Diagram (OCD) that illustrates all the object state models and communication between objects.

   Output: Object Communication Diagram (OCDs)

4. For each state transition in the STDs, create a Data Flow diagram and describe the processing that causes the state to change. Processes in the DFDs can create events that appear in the STDs.

   Output: Data Flow diagrams (DFDs)

5. Generate reports from the diagrams in the form of an Objects/Attributes list, an event list and a relationship list. Each object in the ISD becomes a data store in a DFD.

**External Specification phase**

1. State the boundaries of the system - the imaginary line between the abstract system and the portion outside the system. This distinction should be based on external events that cross this boundary.

2. Create an external event list

3. Create a functional requirements document and back this document up with the abstract system models created in the Systems Analysis phase. This document can also contain screen & report layouts, descriptions of interfaces between the system and external systems and operator procedures.

**Model Integration**

STDs are build for objects that exhibit a life cycle or operational cycle.

STDs cause actions to take place which are represented as processes in a DFD.

Each object in the ERD becomes a data store in a DFD.

Processes accept and produce only the data which is defined in the ERD. Process may create events which appear in STDs.

**Systems Design phase**

1. Design the system architecture

2. Design the system information content

3. Design data structures

4. Partition the system into programs

**Implementation phase**

1. Collect the data (identify sources, develop entry procedures, prepare data structures, enter the data and verify that it is correctly entered)

2. Design programs

3. Code and test the programs

4. Integrate the system
Another popular technique for Object-oriented Analysis and Design is from Peter Coad and Ed Yourdon. The Coad/Yourdon OOA/OOD approach is defined in Object-Oriented Analysis and Object-Oriented Design, both from Prentice Hall.

Analysis phase
1. Identify Objects (Object layer ERD)
2. Identify Structures
3. Define Subjects (Subject layer ERD)
4. Define Attributes (and instance connections) Attribute layer ERD
5. Define Services (and message connections) Service layer ERD

Design phase
1. Define and design the problem domain component
2. Design the human interaction component
3. Design the task management component
4. Design the data management component
5. Select an OOPL and implement the design

Coad/Yourdon Entity-relationship notation

Synthesis - Meilir Page-Jones & Steven Weiss

Analysis Phase
1. Create an augmented Information Model
2. Create an events dictionary
3. Create a context diagram for the system
4. Create state-transition diagrams
5. Create neighborhood diagrams

Design Phase
1. Object classes are mapped to processors
2. Implement a software architecture
3. Package the objects
4. Design internal aspects of the objects
5. Code the objects in an OOPL

Synthesis uses a Unified Object Notation (UON) which includes four types of diagrams in a single object-structure diagrams.

Class-definition diagram: provides the external or abstract definition of the features of a class that are available in a normal way to external clients. These diagrams show messages begin passed between classes and defined as a list of formal object names with the object couple.

Instance diagrams: model inheritance as a directed relationship from the inheriting object module to the parent or superclass from which it inherits. Notation is not required on the connections since current languages support all or nothing inheritance.

Object-communication diagrams: shows the interaction of selected objects through message passing. These diagrams show only the methods of each object that are called into play by the messages shown. Events are drawn and labeled.

Method-internals diagrams: shows the internal or detailed design of an object module. It is derived from a structure chart and has been extended to handle object orientation - message passing, hierarchy of components, etc.

Benefit: Using OOA allows models to be created using objects identified in the system using inheritance, aggregation, encapsulation, information hiding and messaging.
Another popular technique for Object-oriented Analysis and Design is from Peter Coad and Ed Yourdon. The Coad/Yourdon OOA/OOD approach is defined in Object-Oriented Analysis and Object-Oriented Design, both from Prentice Hall.

**Analysis phase**
1. Identify Objects (Object layer ERD)
2. Identify Structures
3. Define Subjects (Subject layer ERD)
4. Define Attributes (and instance connections) Attribute layer ERD
5. Define Services (and message connections) Service layer ERD

**Design phase**
1. Define and design the problem domain component
2. Design the human interaction component
3. Design the task management component
4. Design the data management component
5. Select an OOPL and implement the design

---

**Coad/Yourdon Entity-relationship notation**

```
Book

Author

Copy
```

---

**Synthesis - Meilir Page-Jones & Steven Weiss**

**Analysis Phase**
1. Create an augmented Information Model
2. Create an events dictionary
3. Create a context diagram for the system
4. Create state-transition diagrams
5. Create neighborhood diagrams

**Design Phase**
1. Object classes are mapped to processors
2. Implement a software architecture
3. Package the objects
4. Design internal aspects of the objects
5. Code the objects in an OOPL

Synthesis uses a Unified Object Notation (UON) which includes four types of diagrams in a single object-structure diagrams.

- **Class-definition diagram**: provides the external or abstract definition of the features of a class that are available in a normal way to external clients. These diagrams show messages begin passed between classes and defined as a list of formal object names with the object couple.

- **Instance diagrams**: model inheritance as a directed relationship from the inheriting object module to the parent or superclass from which it inherits. Notation is not required on the connections since current languages support all or nothing inheritance.

- **Object-communication diagrams**: shows the interaction of selected objects through message passing. These diagrams show only the methods of each object that are called into play by the messages shown. Events are drawn and labeled.

- **Method-internals diagrams**: shows the internal or detailed design of an object module. It is derived from a structure chart and has been extended to handle object orientation - message passing, hierarchy of components, etc.

**Benefit**: Using OOA allows models to be created using objects identified in the system using inheritance, aggregation, encapsulation, information hiding and messaging.
**OO Design:** the process of translating or specifying a physical structure composed of software objects from the logical structure defined in Analysis.

OOD consists of four basic activities:

1. Identifying and defining objects and classes
2. Organizing relationships among the classes
3. Cultivating frameworks in a hierarchy of classes
4. Building reusable class libraries and application frameworks (AF)
   (AF = a set of classes that express a design for a family of related applications)

Several approaches to object-oriented design currently exist, and two of these techniques are briefly described below.

### The Booch technique (Object-Oriented Design)

Grady Booch, in his book *Software Engineering with Ada*, describes a technique for the design and implementation of systems using Ada and a structured diagram. Booch defines an object as a model of a real world entity that combines both data and operations on that data. These objects, once identified, are the basis for the modules of a system or set of related objects.

The Booch technique uses diagrams to represent the constructs of the Ada language and derives a set of objects from a specification and informal design. The Booch diagramming notation shows dependencies between the Ada packages and tasks which implement the objects.

While this approach only focuses on the design and implementation phases, it is sometimes used together with a System Analysis technique such as Structured Analysis, Object-Oriented Analysis or the Ward/Mellor or Hatley/Pirhbai real-time Analysis techniques.

Booch's OOD technique is a cyclical process of refining and reorganizing classes involving the following steps:

1. Identify the objects (nouns = potential classes of objects)
2. Associate attributes with each object
3. Identify the methods (verbs = potential methods)
4. Collect methods with their objects
5. Define the interfaces between the objects

Booch has also written a book with less of an emphasis on Ada, *Object Oriented Design with Applications*, from Addison-Wesley.

---

### The Buhr technique - Ada Structure Graphs (ASG)

Raymond Buhr, in his book *Systems Design with Ada*, extends the Booch diagram by adding notations including symbols for task activation order, guarded task activation, package nesting and other symbols that represent Ada constructs.

The Buhr approach is also aimed at Ada development, with inspiration from the Structured Design technique and an orientation towards objects, graphical notation and a conceptual model.

The Buhr technique combines some aspects of (DeMarco) Structured Analysis and (Yourdon/Constantine) Structured Design techniques. This approach uses Structure charts along with Ada structure graphs (ASGs) but is closely tied to the Ada language. Buhr is currently working on a new method, using Machine Charts that provide a more task-oriented approach.
OO Design: the process of translating or specifying a physical structure composed of software objects from the logical structure defined in Analysis.

OOD consists of four basic activities:

1. Identifying and defining objects and classes
2. Organizing relationships among the classes
3. Cultivating frameworks in a hierarchy of classes
4. Building reusable class libraries and application frameworks (AF)
   (AF = a set of classes that express a design for a family of related applications)

Several approaches to object-oriented design currently exist, and two of these techniques are briefly described below.

The Booch technique (Object-Oriented Design)

Grady Booch, in his book *Software Engineering with Ada*, describes a technique for the design and implementation of systems using Ada and a structured diagram. Booch defines an object as a model of a real world entity that combines both data and operations on that data. These objects, once identified, are the basis for the modules of a system or set of related objects.

The Booch technique uses diagrams to represent the constructs of the Ada language and derives a set of objects from a specification and informal design. The Booch diagramming notation shows dependencies between the Ada packages and tasks which implement the objects.

While this approach only focuses on the design and implementation phases, it is sometimes used together with a System Analysis technique such as Structured Analysis, Object-Oriented Analysis or the Ward/Mellor or Hatley/Pirhbai real-time Analysis techniques.

Booch's OOD technique is a cyclical process of refining and reorganizing classes involving the following steps:

1. Identify the objects (nouns = potential classes of objects)
2. Associate attributes with each object
3. Identify the methods (verbs = potential methods)
4. Collect methods with their objects
5. Define the interfaces between the objects

Booch has also written a book with less of an emphasis on Ada, *Object Oriented Design with Applications*, from Addison-Wesley.

The Buhr technique - Ada Structure Graphs (ASG)

Raymond Buhr, in his book *Systems Design with Ada*, extends the Booch diagram by adding notations including symbols for task activation order, guarded task activation, package nesting and other symbols that represent Ada constructs.

The Buhr approach is also aimed at Ada development, with inspiration from the Structured Design technique and an orientation towards objects, graphical notation and a conceptual model.

The Buhr technique combines some aspects of (DeMarco) Structured Analysis and (Yourdon/Constantine) Structured Design techniques. This approach uses Structure charts along with Ada structure graphs (ASGs) but is closely tied to the Ada language. Buhr is currently working on a new method, using Machine Charts that provide a more task-oriented approach.
**Real-Time & Object-Oriented Techniques**

<table>
<thead>
<tr>
<th>Methods to diagrams matrix</th>
<th>Ward Mellor</th>
<th>Hatley Pirbhai</th>
<th>Harel</th>
<th>Shlaer Mellor OOA OOD</th>
<th>Coad Yourdon OOD</th>
<th>Object Oriented Structured Design</th>
<th>Booch OOD</th>
<th>Buhr OOD</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control flow diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity Relationship diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Transition diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure charts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control specs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decision tables or trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity, State and Module charts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architecture diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ada Structure diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other diagrams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOSD</td>
<td>Uniform Object Notation</td>
</tr>
</tbody>
</table>

**OO Programming:** creating programs using collections of self-sufficient objects with encapsulated data and behavior by interacting with each other via messages.

Construction in an Object-oriented development life cycle (OODLC) should be a process of assembling existing objects to meet the needs of the problem domain or system under development. To describe an object, we must specify its attributes, its behavior and its relationship with other objects. To use an object, the data structures and process structures don't need to be understood.

**Object-oriented programming languages**

- True OO languages
  - Eiffel
  - Smalltalk
  - Actor
- Hybrid languages
  - C++
  - Object Pascal
  - Common LISP Object System (CLOS)
  - Objective-C

**Benefits of hybrid OOPLs vs. true OOPLs**

- Existing applications can still be compiled with hybrid OOPLs
- Hybrid are usually faster and more memory efficient than true OOPLs
- Hybrid OOPLs can usually be customized and are more flexible
Construction in an Object-oriented development life cycle (OODLC) should be a process of assembling existing objects to meet the needs of the problem domain or system under development. To describe an object, we must specify its attributes, its behavior and its relationship with other objects. To use an object, the data structures and process structures don't need to be understood.

**Object-oriented programming languages**

- **True OO languages**
  - Eiffel
  - Smalltalk
  - Actor

- **Hybrid languages**
  - C++
  - Object Pascal
  - Common LISP Object System (CLOS)
  - Objective-C

**Benefits of hybrid OOPLs vs. true OOPLs**

Existing applications can still be compiled with hybrid OOPLs

Hybrid are usually faster and more memory efficient than true OOPLs

Hybrid OOPLs can usually be customized and are more flexible
### Comparing Object-oriented programming languages

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Actor</th>
<th>Ada</th>
<th>C++</th>
<th>CLOS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC</td>
<td>Many</td>
<td>Many</td>
<td>HP, Sun, Apollo &amp; VAX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Whitewater</th>
<th>DoD</th>
<th>AT&amp;T</th>
<th>Luicd, Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Franz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Features:</th>
<th>Instance Variables</th>
<th>Instance Methods</th>
<th>Inheritance</th>
<th>Binding</th>
<th>Class Variables</th>
<th>Class Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Yes</td>
<td>Yes</td>
<td>Single</td>
<td>Static/Dynamic</td>
<td>FR</td>
<td>FR</td>
</tr>
<tr>
<td>Ada</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Static</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>C++</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
<td>Static/Dynamic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CLOS</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eiffel</th>
<th>Objective-C</th>
<th>Smalltalk-80</th>
<th>Smalltalk/V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Unix</td>
<td>Apollo, HP, and PC (386)</td>
<td>Macintosh, and PC/DOS/OS/2</td>
</tr>
<tr>
<td></td>
<td>VAX</td>
<td></td>
<td>Digital</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Interactive Software Engineering</th>
<th>ἴ</th>
<th>ParcPlace</th>
<th>Digital</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stepstone</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Benefits of true OOPLs

Tightly integrated environments (closed)
Offer flexible class libraries, debuggers and editors
Not easily adaptable to all problems and systems

### OO and GUI

Graphical User Interfaces (GUI's) came out of the work performed at the Xerox Palo Alto Research Center (PARC) in the 1970's. A GUI can often be characterized as having windows, icons, dialog boxes and a mouse or other input device. GUI's support windowing, imaging, an application programming interface (API) and a framework for developers.

Examples of GUI's include XWindows, NextStep, OSF/Motif, OpenLook, OS/2 Presentation Manager, Windows and the Macintosh.

### Components of UNIX-based GUI's

<table>
<thead>
<tr>
<th>API</th>
<th>Open Look</th>
<th>Next Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP XWidgets</td>
<td>XView</td>
<td>XUI</td>
</tr>
<tr>
<td>X Windows</td>
<td></td>
<td>NextStep</td>
</tr>
<tr>
<td>X Windows</td>
<td>X Windows X11/News</td>
<td></td>
</tr>
<tr>
<td>Display Postscript</td>
<td>Display Postscript</td>
<td></td>
</tr>
<tr>
<td>Unix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 1991 by Foresite Systems
Comparing Object-oriented programming languages

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Actor</th>
<th>Ada</th>
<th>C++</th>
<th>CLOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Many</td>
<td>Many</td>
<td>HP, Sun, Apollo &amp; VAX</td>
<td></td>
</tr>
<tr>
<td>Vendor</td>
<td>Whitewater</td>
<td>DoD</td>
<td>AT&amp;T Borland ZorTech</td>
<td>Luicd, Inc. Franz</td>
</tr>
</tbody>
</table>

**Features:**
- **Instance Variables:**
  - Actor: Yes
  - Ada: Yes
  - C++: Yes
  - CLOS: Yes

- **Instance Methods:**
  - Actor: Yes
  - Ada: Yes
  - C++: Yes
  - CLOS: Yes

- **Inheritance:**
  - Actor: Single
  - Ada: None
  - C++: Multiple
  - CLOS: Multiple

- **Class Variables:**
  - Actor: FR
  - Ada: No
  - C++: Yes
  - CLOS: Yes

- **Class Methods:**
  - Actor: FR
  - Ada: No
  - C++: Yes
  - CLOS: Yes

- **Features:**
  - **Eiffel:**
    - Platforms: Unix
    - Vendor: Interactive Software Engineering
  - **Objective-C:**
    - Platforms: Many
    - Vendor: Stepstone
  - **Smalltalk-80:**
    - Platforms: Apollo, HP, Sun & PC (386)
    - Vendor: ParcPlace
  - **Smalltalk/V:**
    - Platforms: Macintosh, PC DOS & OS/2
    - Vendor: Digitalk

**Benefits of true OOPLs**
Tightly integrated environments (closed)
Offer flexible class libraries, debuggers and editors
Not easily adaptable to all problems and systems

**OO and GUI**

Graphical User Interfaces (GUI’s) came out of the work performed at the Xerox Palo Alto Research Center (PARC) in the 1970’s. A GUI can often be characterized as having windows, icons, dialog boxes and a mouse or other input device. GUI’s support windowing, imaging, an application programming interface (API) and a framework for developers.

Examples of GUI’s include XWindows, NextStep, OSF/Motif, OpenLook, OS/2 Presentation Manager, Windows and the Macintosh.

**Components of UNIX-based GUI’s**

<table>
<thead>
<tr>
<th>Component</th>
<th>CI</th>
<th>Motif</th>
<th>DEC-Windows</th>
<th>Open Look</th>
<th>Next Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>HP</td>
<td>X Windows</td>
<td>XUI</td>
<td>X View</td>
<td>Kks</td>
</tr>
<tr>
<td>Windowing System</td>
<td>X Windows</td>
<td>X Windows X11/News</td>
<td>Display Postscript</td>
<td>Display Postscript</td>
<td></td>
</tr>
<tr>
<td>Imaging Model</td>
<td>Proprietary</td>
<td>?</td>
<td>Display Postscript</td>
<td>Display Postscript</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Unix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 1991 by Foresite Systems
Object-Oriented DBMS

OODBMS’s: provide facilities for defining and manipulating objects and relationships. OODBMS’s are usually better suited to support complex data structures since their data types are highly variable (bit-mapped images, etc.) and versioning is fairly easy in OODBMS’s.

The need for OODBMS’s

Object-oriented tools can map their object models to the OODBMS more easily than traditional DBMS’s since OODBMS’s support encapsulation, polymorphism, and inheritance. If these characteristics are built into the DBMS, the application can be simpler and more easily changed. OODBMS’s also allow storage of reusable objects.

Elements of an Object-Oriented DBMS (OODBMS)

Mandatory Object Features:
1. Support for complex objects
2. Support for object identity
3. Support for object encapsulation
4. Support for object types or classes
5. Support for class or type inheritance
6. Support for late binding with overriding and overloading
7. Support for computational completeness
8. Support the ability to add new types (extensibility)

Mandatory Database Features:
1. Support the persistence of objects
2. Support the ability to manage very large DBMS’s
3. Support concurrent user access/update
4. Support recovery from hardware/software failure
5. Support ad hoc query

Optional Features:
1. Support multiple inheritance
2. Support type checking and inferencing
3. Support distributed systems
4. Support long transactions
5. Support versioning

Source: The Object-Oriented Database System Manifesto

Problems with OODBMS’s

Query optimization
Distributed DBMS
Concurrency
Performance
Language support

Object-Oriented Environment

OO Environment: A set of integrated tools often including class hierarchy browsers, inspectors, profilers, debuggers, incremental compilers and linkers and class libraries.

Components of OOE’s

Browsers: interactive programs that permit navigation through class libraries and show relationships between classes, messages, methods for a class, variables, senders of a message, implementors of a message and implementation code of a method.

Inspectors: allow examination and editing of values of an object’s instance variables.

Profilers: a tool for analyzing a method’s performance - shows where the method spends its time; in OO terms, traces messages and their action within methods.

Debuggers: control the execution and examination of a running program.

Incremental compilers and linkers: support repetitive compile and link steps and include affected objects/methods.
Object-Oriented DBMS

OODBMS’s: provide facilities for defining and manipulating objects and relationships. OODBMS’s are usually better suited to support complex data structures since their data types are highly variable (bit-mapped images, etc.) and versioning is fairly easy in OODBMS’s.

The need for OODBMS’s

Object-oriented tools can map their object models to the OODBMS more easily than traditional DBMS’s since OODBMS’s support encapsulation, polymorphism, and inheritance. If these characteristics are built into the DBMS, the application can be simpler and more easily changed. OODBMS’s also allow storage of reusable objects.

Elements of an Object-Oriented DBMS (OODBMS)

Mandatory Object Features:
1. Support for complex objects
2. Support for object identity
3. Support for object encapsulation
4. Support for object types or classes
5. Support for class or type inheritance
6. Support for late binding with overriding and overloading
7. Support for computational completeness
8. Support the ability to add new types (extensibility)

Mandatory Database Features:
1. Support the persistence of objects
2. Support the ability to manage very large DBMS’s
3. Support concurrent user access/update
4. Support recovery from hardware/software failure
5. Support ad hoc query

Optional Features:
1. Support multiple inheritance
2. Support type checking and inferencing
3. Support distributed systems
4. Support long transactions
5. Support versioning

Source: The Object-Oriented Database System Manifesto

Problems with OODBMS’s

Query optimization
Distributed DBMS
Concurrency
Performance
Language support

Object-Oriented Environment

OO Environment: A set of integrated tools often including class hierarchy browsers, inspectors, profilers, debuggers, incremental compilers and linkers and class libraries.

Components of OOE’s

Browsers: interactive programs that permit navigation through class libraries and show relationships between classes, messages, methods for a class, variables, senders of a message, implementors of a message and implementation code of a method.

Inspectors: allow examination and editing of values of an object’s instance variables.

Profilers: a tool for analyzing a methods performance - shows where the method spends its time; in OO terms, traces messages and their action within methods.

Debuggers: control the execution and examination of a running program.

Incremental compilers and linkers: support repetitive compile and link steps and include affected objects/methods.
Resources

Object-Oriented Conferences

The Object-Oriented Systems Conference - Digital Consulting, Inc. (508) 470-3880
Seminars & Conference in Object-Oriented Programming (SCOOP) - Journal of Object-Oriented Programming $895 (212) 274-0640
Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) - ACM
Technology of Object-Oriented Languages & Systems (TOOLS) -

Object-Oriented magazines, newsletters and journals

The C++ Report $ 49 (212) 274-0640 Sigs Publication, Inc.
Also HOTLINE on Object-Oriented Technology $ 249
Journal of Object-Oriented Programming $ 49
Object Magazine $ 40 COOT, Inc. (212) 274-0640
Inside Turbo C++ from The Cobb Group (800) 223-8720 $ 79

Other resources

Object Management Architecture Guide from Object Management Group (508) 820-4300 $ 30
1991 International OOP Directory from COOT, Inc. (212) 274-0640

Object-Oriented Tools

Object-Oriented Analysis & Design tools

Object+ from EasySpec (713) 480-3233
Object-Vision from Object Vision (415) 540-4889
ObjectMaker, CGen and AdaGen from Mark V Systems (818) 995-7671
Power Tools from Iconix Software (213) 458-0092
Software through Pictures from Interactive Development Environments (800) 888-4331
System Architect from Popkin Software & Systems (212) 571-3434
Teamwork/Ada and Teamwork/C++ from Cadre Technologies (401) 351-5950

Object-Oriented Programming tools

Hybrid Object-Oriented languages
C++
  Designer C++ from Oasys (617) 890-7889
  Saber C++ from Saber Software Inc. (617) 976-7636
  Turbo C++ from Borland (800) 331-0877
  Zortec C++ from Zortec (617) 646-6703
Object Pascal
  Turbo Pascal from Borland (408) 438-5300
Object C
  Stepstone Corp. (203) 426-1875

True Object-Oriented languages
Smalltalk from Digital (213) 645-1082
  and from ParcPlace Systems (415) 659-1000
Actor from The Whitewater Group (800) 869-1144
Eiffel from Interactive Software Engineering (805) 685-1006

Object-Oriented Databases

GemStone from Servio Logic Development (503) 644-4242
Genesis from Deductive Systems
Object-Base from Object-Sciences Corp.
Object-Store from Object Design
Objectivity/DB from Objectivity Inc. (415) 688-8000
Ontos from Ontologic (617) 272-7110
Statice from Symbolics
Resources

Object-Oriented Conferences

The Object-Oriented Systems Conference - Digital Consulting, Inc. (508) 470-3880

Seminars & Conference in Object-Oriented Programming (SNOOP) - Journal of Object-Oriented Programming $895 (212) 274-0640

Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) - ACM

Technology of Object-Oriented Languages & Systems (TOOLS) -

Object-Oriented magazines, newsletters and journals

The C++ Report $ 49 (212) 274-0640 Sigs Publication, Inc.
Also HOTLINE on Object-Oriented Technology $ 249
Journal of Object-Oriented Programming $ 49

Object Magazine $ 40 COOT, Inc. (212) 274-0640

Inside Turbo C++ from The Cobb Group (800) 223-8720 $ 79

Other resources

Object Management Architecture Guide from Object Management Group (508) 820-4300
$ 30

1991 International OOP Directory from COOT, Inc. (212) 274-0640

Object-Oriented Tools

Object-Oriented Analysis & Design tools

Object+ from EasySpec (713) 480-3233
Object-Vision from Object Vision (415) 540-4889
ObjectMaker, CGen and AdaGen from Mark V Systems (818) 995-7671
Powertools from Iconix Software (213) 458-0092
Software through Pictures from Interactive Development Environments (800) 888-4331
System Architect from Popkin Software & Systems (212) 571-3434
Teamwork/Ada and Teamwork/C++ from Cadre Technologies (401) 351-5950

Object-Oriented Programming tools

Hybrid Object-Oriented languages
C++

Designer C++ from Oasys (617) 890-7889
Saber C++ from Saber Software Inc. (617) 976-7636
Turbo C++ from Borland (800) 331-0877
Zortec C++ from Zortec (617) 646-6703

Object Pascal
Turbo Pascal from Borland (408) 438-5300
Objective C
Steppstone Corp. (203) 426-1875

True Object-Oriented languages
Smalltalk from Digital (213) 645-1082
and from ParcPlace Systems (415) 859-1000
Actor from The Whitewater Group (800) 869-1144
Eiffel from Interactive Software Engineering (805) 685-1006

Object-Oriented Databases

GemStone from Servio Logic Development (503) 644-4242
Genesis from Deductive Systems
Object-Base from Object-Sciences Corp.
Object-Store from Object Design
Objective/DB from Objective Inc. (415) 688-8000
Ontos from Ontologic (617) 272-7110
Statice from Symbolics

Copyright © 1991 by Foresite Systems
Object-Oriented GUI tools

C_Talk & C_Views from CNS (612) 944-0170
HOOD Toolset from Caset Corp. (714) 496-8670
Objectworks C++/Smalltalk from Parc Place Systems (800) 762-6265
VZ Programmer from VZ Corp. (801) 595-1352

Object-Oriented Environments

Eiffel from ISE
MacApp from Apple Computer (408) 973-4831
NextStep from Next, Inc.
Object Plus from EasySpec
Object-Vision from Object Vision
Prograph from TGS Systems (902) 429-5642
Software through Pictures and Saber (C Environment)

Selling your organization on object-oriented technology

Productivity

While many organizations use the promise of increased productivity to sell object-oriented technology this approach can be difficult to prove in the short term. Studies have shown that object-oriented tools can improve productivity but usually not on the first, second or even third project. After having used the tool and technique successfully a few times, components in the object-oriented life cycle can be reused on other systems. Most estimates are that productivity can improve after two to three years. Many organizations fail to measure their productivity before or after they implement object-oriented technology and cannot show any measurable results.

Quality

Improved software quality is definitely a reasonable choice for a short term goal or benefit to sell object-oriented technology in an organization. Given that the object-oriented tool or technique is used, the resulting software will probably have fewer problems in the later phases of the SDLC. But to show an improvement in software quality, you must first measure before the object-oriented technology is implemented. Most organizations do not have formal quality measurement measures in place prior to implementing object-oriented technology.

Increased predictability

Since object-oriented tools and techniques make extensive use of reusable components and encapsulation, the resulting software will typically act more predictably. Since components can be thoroughly tested and subsequent child components inherit these characteristics, they should also be more predictable.

Complexity controlled

Given that there are very few small, simple computer systems being developed, controlling the growing complexity of software development is a real problem. Object-oriented technology can simplify this by providing a single collection point for development information and powerful tools for analyzing this information for its accuracy. Also, though the use of class libraries, much of the complexity in developing systems can be reduced.

Maintenance

Object-oriented technology can help provide a more rigorous environment for software development along with formalized quality assurance, tracking, testing and other tools and techniques. Since much of maintenance involves enhancing or extending existing class hierarchies or object classes, maintenance can be improved by modularizing functions. Organizations that are required to provide documentation in the form of MIL-STD-2167A or similar standards can often benefit from the information available from object-oriented technology.

Copyright © 1991 by Foresite Systems
Object-Oriented GUI tools

- C_Talk & C_Views from CNS (612) 944-0170
- HOOD Toolset from Caset Corp. (714) 496-8670
- Objectworks C++/Smalltalk from Parc Place Systems (800) 762-6265
- VZ Programmer from VZ Corp. (801) 595-1352

Object-Oriented Environments

- Eiffel from ISE
- MacApp from Apple Computer (408) 973-4831
- NextStep from Next, Inc.
- Object Plus from EasySpec
- Object-Vision from EasySpec
- Prograph from TGS Systems (902) 429-5642
- Software through Pictures and Saber (C Environment)

Selling your organization on object-oriented technology

Productivity

While many organizations use the promise of increased productivity to sell object-oriented technology this approach can be difficult to prove in the short term. Studies have shown that object-oriented tools can improve productivity but usually not on the first, second or even third project. After having used the tool and technique successfully a few times, components in the object-oriented life cycle can be reused on other systems. Most estimates are that productivity can improve after two to three years. Many organizations fail to measure their productivity before or after they implement object-oriented technology and cannot show any measurable results.

Quality

Improved software quality is definitely a reasonable choice for a short term goal or benefit to sell object-oriented technology in an organization. Given that the object-oriented tool or technique is used, the resulting software will probably have fewer problems in the later phases of the SDLC. But to show an improvement in software quality, you must first measure before the object-oriented technology is implemented. Most organizations do not have formal quality measurement measures in place prior to implementing object-oriented technology.

Increased predictability

Since object-oriented tools and techniques make extensive use of reusable components and encapsulation, the resulting software will typically act more predictably. Since components can be thoroughly tested and subsequent child components inherit these characteristics, they should also be more predictable.

Complexity controlled

Given that there are very few small, simple computer systems being developed, controlling the growing complexity of software development is a real problem. Object-oriented technology can simplify this by providing a single collection point for development information and powerful tools for analyzing this information for its accuracy. Also, though the use of class libraries, much of the complexity in developing systems can be reduced.

Maintenance

Object-oriented technology can help provide a more rigorous environment for software development along with formalized quality assurance, tracking, testing and other tools and techniques. Since much of maintenance involves enhancing or extending existing class hierarchies or object classes, maintenance can be improved by modularizing functions. Organizations that are required to provide documentation in the form of MIL-STD-2167A or similar standards can often benefit from the information available from object-oriented technology.
CASE transition strategies

Moving from an ad-hoc, unstructured, non-automated software development environment to a CASE environment involves risk. The transition from the 'old way' to the 'new way' can be very difficult and without careful planning, may fail. Many experts have provided us with a wealth of information on this transition to CASE including Wayland Systems, Barb Bouldin, Roger Pressman and Watts Humphrey. In addition, we will be looking at two hardware vendors plans for installing CASE: IBM and DEC.

The famous "J" curve

Vendor Promisses

The actual curve

Management Fears

Copyright © 1991 by Foresite Systems
CASE transition strategies

Moving from an ad-hoc, unstructured, non-automated software development environment to a CASE environment involves risk. The transition from the 'old way' to the 'new way' can be very difficult and without careful planning, may fail. Many experts have provided us with a wealth of information on this transition to CASE including Wayland Systems, Barb Bouldin, Roger Pressman and Watts Humphrey. In addition, we will be looking at two hardware vendors plans for installing CASE: IBM and DEC.

The famous "J" curve

Productivity

Vendor Promisses

The actual curve

Management Fears

Time

Copyright © 1991 by Foresite Systems
Page 560

Copyright © 1991 by Foresite Systems
Page 561
The J curve and the seven (7) stages of software engineering expertise

Stage 1 - Innocent

These people have never heard of software engineering methods and have no knowledge of their applicability. Stage 1 people may be blissfully unaware of any problems or a "software crisis".

Stage 2 - Exposed

Stage 2 people have heard of software engineering methods through colleagues, magazine articles or whatever, and believe that these methods have some relevance to them and their jobs.

Stage 3 - Apprentice

The apprentice or stage 3 person has been through a seminar or tutorial on software engineering methods and have a broad, superficial understanding of the methods themselves. But stage 3's have never had a chance to use the methods on actual projects.

Stage 4 - Practitioner

Practitioners have actually used software engineering methods seriously at least one time. They know about the tough parts and how to make the methods work, but these methods are not always second nature. Additional guidance is often required by stage 4's to ensure productivity.

Stage 5 - Journeyman (or Journeywoman)

Stage 5's use software engineering methods regularly and naturally in the day to day work. They are typically more productive and rarely need any advice or guidance on methods.

Stage 6 - Expert

Stage 6's are thoroughly conversant with software engineering and know the rules so well, they even know when to break them in order to achieve better results. They are often found training others in software engineering methods.

Stage 7 - Researcher

Stage 7's are at the leading edge of software engineering practices and are often called up to write books, give papers and speak on software engineering methods. They discover new ideas and advance the state of the art in software engineering.

* From Wayland Systems, (206) 957-0124

Barbara Bouldin has written extensively on managing the change required to transition from non-automated development techniques to CASE. Ms. Bouldin spent many years at AT&T as a change agent on CASE and structured methods. Her book, "Agents of Change," is an excellent resource for the issues related to transitioning to a CASE environment. Ms. Bouldin has identified a life cycle for implementing change and several key factors to successfully managing this change:

- Provide information in familiar terms
- Listen effectively, brainstorm and develop the basic game plan
- Plan intermediate deliverables
- Don't use edicts - involve users
- Build on what is already effective
- Avoid excessive ambition
- Manage the expectations
- Implement without disruption
- Keep everyone involved
- Perform periodic review & sign off

Unforeseen obstacles to change

- Noncooperation from users
- Inexperienced staff
- Management intervention
- Unsolicited reorganization

Critical Success Factors for CASE Tool Acceptance

- There must be a change agent
- There must be management support
- You must use the appropriate approach
- You must have a good product
- You must have a good vendor
The J curve and the seven (7) stages of software engineering expertise

Stage 1 - Innocent

These people have never heard of software engineering methods and have no knowledge of their applicability. Stage 1 people may be blissfully unaware of any problems or a "software crisis".

Stage 2 - Exposed

Stage 2 people have heard of software engineering methods through colleagues, magazine articles or whatever, and believe that these methods have some relevance to them and their jobs.

Stage 3 - Apprentice

The apprentice or stage 3 person has been through a seminar or tutorial on software engineering methods and have a broad, superficial understanding of the methods themselves. But stage 3's have never had a chance to use the methods on actual projects.

Stage 4 - Practitioner

Practitioners have actually used software engineering methods seriously at least one time. They know about the tough parts and how to make the methods work, but these methods are not always second nature. Additional guidance is often required by stage 4's to ensure productivity.

Stage 5 - Journeyman (or Journeywoman)

Stage 5's use software engineering methods regularly and naturally in the day to day work. They are typically more productive and rarely need any advice or guidance on methods.

Stage 6 - Expert

Stage 6's are thoroughly conversant with software engineering and know the rules so well, they even know when to break them in order to achieve better results. They are often found training others in software engineering methods.

Stage 7 - Researcher

Stage 7's are at the leading edge of software engineering practices and are often called up to write books, give papers and speak on software engineering methods. They discover new ideas and advance the state of the art in software engineering.

* From Wayland Systems, (206) 957-0124

Barbara Bouldin has written extensively on managing the change required to transition from non-automated development techniques to CASE. Ms. Bouldin spent many years at AT&T as a change agent on CASE and structured methods. Her book, "Agents of Change," is an excellent resource for the issues related to transitioning to a CASE environment. Ms. Bouldin has identified a life cycle for implementing change and several key factors to successfully managing this change:

- Provide information in familiar terms
- Listen effectively, brainstorm and develop the basic game plan
- Plan intermediate deliverables
- Don't use edicts - involve users
- Build on what is already effective
- Avoid excessive ambition
- Manage the expectations
- Implement without disruption
- Keep everyone involved
- Perform periodic review & sign off

Unforeseen obstacles to change

- Noncooperation from users
- Inexperienced staff
- Management intervention
- Unsolicited reorganization

Critical Success Factors for CASE Tool Acceptance

- There must be a change agent
- There must be management support
- You must use the appropriate approach
- You must have a good product
- You must have a good vendor
Recognizing Success
(You will recognize success when...)
- Will not always be dramatic but will be fundamental
- When the first deliverable is completed within budget and on-time
- Recognize and accepted by management & users
- Measured improvement utilizing techniques specified in the measurement plan
- Substantial work remains but your effort is secure

Barb Bouldin's Life Cycle of Implementing Change

Roger Pressman: A Software Engineering Life Cycle

Roger Pressman has written on the process of selecting and implementing structured methods and CASE tools. In his books, Dr. Pressman introduces a Software Engineering Life Cycle (SELC) for the transition to a CASE environment.

This SELC involves five basic steps:
- **Assessment**: Assess where the organization is in its software development environment and identify strong and weak points in the process. Pressman recommends organizations perform a Software Engineering Audit (SEA) prior to moving forward with CASE.
- **Education**: Educate everyone involved in the methods, techniques and tools to be used or already available.
- **Selection**: Select any new techniques and/or tools that can help improve the software development process.
- **Installation**: Install techniques and tools as selected.
- **Evaluation**: Evaluate whether the software process has been improved by the tools and techniques installed and how the process can be further improved with other tools and techniques.

It is important to recognize that Pressman strongly suggests this SELC is a repetitive process and doesn’t stop. Once methods and tools have been successfully implemented, the need still exists to evaluate and continue education.
Recognizing Success
(You will recognize success when...)
- Will not always be dramatic but will be fundamental
- When the first deliverable is completed within budget and on-time
- Recognize and accepted by management & users
- Measured improvement utilizing techniques specified in the measurement plan
- Substantial work remains but your effort is secure

Barb Bouldin's Life Cycle of Implementing Change

Roger Pressman: A Software Engineering Life Cycle

Roger Pressman has written on the process of selecting and implementing structured methods and CASE tools. In his books, Dr. Pressman introduces a Software Engineering Life Cycle (SELC) for the transition to a CASE environment.

This SELC involves five basic steps:

**Assessment:** Assess where the organization is in its software development environment and identify strong and weak points in the process. Pressman recommends organizations perform a Software Engineering Audit (SEA) prior to moving forward with CASE.

**Education:** Educate everyone involved in the methods, techniques and tools to be used or already available.

**Selection:** Select any new techniques and/or tools that can help improve the software development process.

**Installation:** Install techniques and tools as selected.

**Evaluation:** Evaluate whether the software process has been improved by the tools and techniques installed and how the process can be further improved with other tools and techniques.

It is important to recognize that Pressman strongly suggests this SELC is a repetitive process and doesn't stop. Once methods and tools have been successfully implemented, the need still exists to evaluate and continue education.
Watts Humphrey: Software process maturity & process control

Watts Humphrey spent some time at IBM and then at the Software Engineering Institute at Carnegie Mellon University. While at SEI, Humphrey pioneered the Software Process Maturity model and wrote about it in his book, “Managing the Software Process.”

The Software Process Maturity (SPM) model has been widely discussed and used by organizations to determine whether they are ready for CASE. Humphrey has also identified a life cycle for improving and controlling the software process itself.

The software process must be defined and the product or outcome of the process is the software itself. Quality standards should be established to define the product. Evaluating the process for quality improvement can be done with software process models like the SPM. Measuring the process can help identify ways of improving the process.

"If you don’t know where you are going, any road will do."

---

**Watts Humphrey Software Process Control**

- **Process Models**
  - Evaluate the Process
- **Quality Standards**
  - Set Quality Standards
- **Process**
  - Define the Process
- **Product (Software)**
  - Improve the Process

**Watts Humphrey Process Maturity Model**

- **Initial**
  - 1 - 2 Years
  - Little use of:
    - Project Planning
    - Project Estimation
    - Irregular usage of CASE tools
- **Repeatable**
  - 2 - 4 Years
  - Software development process is:
    - Single CASE tool
    - Regular usage of CASE tools
    - No configuration management
- **Defined**
  - 2 - 4 Years
  - Software development process is:
    - Formally defined
    - Accepted & used for its merits
    - Quality but not quantified
- **Managed**
  - 1 - 3 Years
  - Software development process is:
    - In general, used
    - Backed by management
    - Sufficient for the status quo
- **Optimized**
  - Software development process is:
    - Formally defined
    - Understood by all involved
    - Accepted & used for its own merits
    - Quality and quantified

**Organizations at this level can:**
- Introduce new technology
- Begin to study their process
- Assess the impact of new technology
- Improve their own process

**Weak Points in Software Process are:**
- Identified
- Modified
- Software process is under measured control

**Estimated status of USA companies**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>86 %</td>
</tr>
<tr>
<td>Repeatable</td>
<td>13 %</td>
</tr>
<tr>
<td>Defined</td>
<td>1 %</td>
</tr>
<tr>
<td>Managed</td>
<td>0 %</td>
</tr>
<tr>
<td>Optimized</td>
<td>0 %</td>
</tr>
</tbody>
</table>

Copyright © 1991 by Foresite Systems

Page 566

Page 567
Watts Humphrey spent some time at IBM and then at the Software Engineering Institute at Carnegie Mellon University. While at SEI, Humphrey pioneered the Software Process Maturity model and wrote about it in his book, "Managing the Software Process."

The Software Process Maturity (SPM) model has been widely discussed and used by organizations to determine whether they are ready for CASE. Humphrey has also identified a life cycle for improving and controlling the software process itself.

The software process must be defined and the product or outcome of the process is the software itself. Quality standards should be established to define the product. Evaluating the process for quality improvement can be done with software process models like the SPM. Measuring the process can help identify ways of improving the process.

---

**Watts Humphrey Software Process Control**

1. **Process**
   - Define the Process
   - Improve the Process
   - Evaluate the Process

2. **Product (Software)**
   - Set Quality Standards

- Measure the Process
- Set Quality Standards

---

"If you don’t know where you are going, any road will do."

---

**Watts Humphrey Process Maturity Model**

- **Initial**
  - 1 - 2 Years
  - Little use of:
    - Project Planning
    - Project Estimation
    - A defined development process
    - Irregular use of tools
    - No configuration management

- **Repeatable**
  - 2 - 4 Years
  - Software development process is:
    - Understood
    - In general (? ) use
    - Backed by Management
    - Sufficient for the status quo
    - Irregular use of tools, methods, etc.

- **Defined**
  - 2 - 4 Years
  - Software Development process is:
    - Formally defined
    - Understood by all involved
    - Accepted & used for its own merits
    - Qualified but not quantified
    - Organizations at this level can:
      - Introduce new technology
      - Begin to study their process

- **Managed**
  - 1 - 3 Years
  - Software Process Data Is:
    - Collected
    - Used to modify the process
    - Related to product quality
    - Organizations at this level can:
      - Assess the process impact of new technology
      - Improve their own process

- **Optimized**
  - Software process is under measured control

---

**Estimated status of USA companies**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>86%</td>
</tr>
<tr>
<td>Repeatable</td>
<td>13%</td>
</tr>
<tr>
<td>Defined</td>
<td>1%</td>
</tr>
<tr>
<td>Managed</td>
<td>0%</td>
</tr>
<tr>
<td>Optimized</td>
<td>0%</td>
</tr>
</tbody>
</table>

Copyright © 1991 by Foresite Systems

Page 566
Three levels of OO development

Peter Wegner of Brown University has identified three (3) levels of object-orientation that an organization may be using:

1) Object-based development supports the concepts of objects and the use of messages to communicate between objects.
2) Class-based development supports the concepts of objects, messaging and classes.
3) Object-oriented development supports the notion of objects, messages, classes and inheritance.

Profile of successful CASE users

They are ...

Using formal structured methods
Using formal life cycle methods
Emphasizing Quality over Productivity
Providing extensive training to all personnel involved
Addressing front-end and back-end CASE issues
Planning to expand use of CASE and methods

Average time to reach proficiency for tools = 3 to 6 months
Average time to reach proficiency for methods = 18 + months

Most promising approach to software improvement

Front-End CASE 35 %
Life Cycle methods 18 %
Back-End CASE 15 %
JAD/Facilitated Sessions 14 %
Structured Methods 10 %
Rapid Prototyping 10 %
4GLs 10 %
Object-Oriented Methods 5 %
Expert Systems 3 %
Three levels of OO development

Peter Wegner of Brown University has identified three (3) levels of object-orientation that an organization may be using:

1) Object-based development supports the concepts of objects and the use of messages to communicate between objects.
2) Class-based development supports the concepts of objects, messaging and classes.
3) Object-oriented development supports the notion of objects, messages, classes and inheritance.

Profile of successful CASE users

They are ...

Using formal structured methods
Using formal life cycle methods
Emphasizing Quality over Productivity
Providing extensive training to all personnel involved
Addressing front-end and back-end CASE issues
Planning to expand use of CASE and methods

Average time to reach proficiency for tools = 3 to 6 months
Average time to reach proficiency for methods = 18 + months

Most promising approach to software improvement

Front-End CASE 35 %
Life Cycle methods 18 %
Back-End CASE 15 %
JAD/Facilitated Sessions 14 %
Structured Methods 10 %
Rapid Prototyping 10 %
4GLs 10 %
Object-Oriented Methods 5 %
Expert Systems 3 %
Selecting an appropriate OO technology

When considering which (or if any) OO technology to select and implement there are many issues that should be taken into consideration.

First and foremost, should a software engineering audit or software assessment be performed prior to evaluating OO technology? If the answer is yes, the works of Roger Pressman, while the relate primarily to software engineering and CASE, can be used as guidelines for assessment of the software development environment. Also refer to the SEI assessment and HP's software improvement approach.

Beyond these considerations, organizations should evaluate where the potential for OO technology is greatest or the transition to OO technology is easiest.

Where is the best potential for OO technology?

Remember that OO technology can be used to support all or part of the software development life cycle (SDLC), from Analysis through Programming and Maintenance.

If an organization is involved in developing for a GUI, an event-driven environment or something similar, an OOPL may be the best place to initiate OO technology. Development towards a GUI can be greatly enhanced with the use of OOPL's, class libraries and an OO Environment.

If an organization is already using structured techniques and CASE, perhaps analysis and/or design would be a better place to start. Remember to make use of whatever skills and expertise your staff already has with existing techniques and tools. Combining OOA with structured design or structured analysis with the Booch OOD technique are good for moving slowly towards OO technology.

Likewise, if an organization already has a significant investment in C or Pascal, they may want to look at moving towards C++ and Object Pascal so they can make use of their existing expertise and slowly move towards OO technology.
Selecting an appropriate OO technology

When considering which (or if any) OO technology to select and implement there are many issues that should be taken into consideration.

First and foremost, should a software engineering audit or software assessment be performed prior to evaluating OO technology? If the answer is yes, the works of Roger Pressman, while the relate primarily to software engineering and CASE, can be used as guidelines for assessment of the software development environment. Also refer to the SEI assessment and HP's software improvement approach.

Beyond these considerations, organizations should evaluate where the potential for OO technology is greatest or the transition to OO technology is easiest.

Where is the best potential for OO technology?

Remember that OO technology can be used to support all or part of the software development life cycle (SDLC), from Analysis through Programming and Maintenance.

If an organization is involved in developing for a GUI, an event-driven environment or something similar, an OOPL may be the best place to initiate OO technology. Development towards a GUI can be greatly enhanced with the use of OOPL's, class libraries and an OO Environment.

If an organization is already using structured techniques and CASE, perhaps analysis and/ or design would be a better place to start. Remember to make use of whatever skills and expertise your staff already has with existing techniques and tools. Combining OOA with structured design or structured analysis with the Booch OOD technique are good for moving slowly towards OO technology.

Likewise, if an organization already has a significant investment in C or Pascal, they may want to look at moving towards C++ and Object Pascal so they can make use of their existing expertise and slowly move towards OO technology.
Choosing the right pilot project for OO

- Pick a project of reasonable size
- Choose a project that is (a) useful, (b) visible and (c) low-risk
- Pick a project that can be measured
- Pick a target environment that is familiar
- Pick normal people for the project - not superstars

Choosing the right pilot project for object-oriented technology

Pick a project of reasonable size

Choosing a very small system for the pilot project may not convince anyone that the tools are effective. Choosing a large project can often drag on for months and the team members can become overwhelmed with the new techniques in the process. Picking a medium sized system to develop using object-oriented technology helps to get everyone familiar with the tools and techniques, gives the staff involved a feeling that the tool actually is a benefit and does not drag on too long.

Choose a project that is useful, visible and low-risk

Trying to implement object-oriented technology with a development effort that no one plans to use can reduce the interest of everyone involved and thus doom the project before it even gets started. Choosing a project that is critical to the success of the business, on the other hand, means that the staff involved may revert to their old ways to ensure the project is completed on time.

Since using object-oriented technology for the first time involves a lot of learning and often mistakes, don't expect the process to be quick or painless. Pick a project that is visible, but offers little risk of failure. Never choose a development effort that requires immediate success and does not allow team members to experiment with the new techniques and tools since this is an important part of the process.

Pick a project that can be measured

Since you will be trying to show that using object-oriented technology is better than the traditional methods, be sure to have measurable aspects of the system when finished. Collecting items like the number of lines of debugged code per programmer per day, the number of bugs found after the system is in production and the amount of time spent in each phase helps to prove the benefits of object-oriented technology to management.

Pick a target environment that is familiar

Don't pick a project that requires learning techniques that are over and above the methodology and the object-oriented tool. Project members may feel overwhelmed if they are expected to learn these techniques along with a new DBMS, TP monitor or programming language. If you expect too much of the team members, they will probably fail and the object-oriented tool and methodology will be a likely scapegoat.

Pick normal people for the project and not the superstars

If you do pick the best and the brightest, the resulting success will probably be attributed to the team and not the methodology and object-oriented tool. Make sure the people chosen are committed to learning the new techniques and using the tools.
Choosing the right pilot project for OO

- Pick a project of reasonable size
- Choose a project that is (a) useful, (b) visible and (c) low-risk
- Pick a project that can be measured
- Pick a target environment that is familiar
- Pick normal people for the project - not superstars

Choosing the right pilot project for object-oriented technology

Pick a project of reasonable size

Choosing a very small system for the pilot project may not convince anyone that the tools are effective. Choosing a large project can often drag on for months and the team members can become overwhelmed with the new techniques in the process. Picking a medium sized system to develop using object-oriented technology helps to get everyone familiar with the tools and techniques, gives the staff involved a feeling that the tool actually is a benefit and does not drag on too long.

Choose a project that is useful, visible and low-risk

Trying to implement object-oriented technology with a development effort that no one plans to use can reduce the interest of everyone involved and thus doom the project before it even gets started. Choosing a project that is critical to the success of the business, on the other hand, means that the staff involved may revert to their old ways to ensure the project is completed on time.

Since using object-oriented technology for the first time involves a lot of learning and often mistakes, don't expect the process to be quick or painless. Pick a project that is visible, but offers little risk of failure. Never choose a development effort that requires immediate success and does not allow team members to experiment with the new techniques and tools since this is an important part of the process.

Pick a project that can be measured

Since you will be trying to show that using object-oriented technology is better than the traditional methods, be sure to have measurable aspects of the system when finished. Collecting items like the number of lines of debugged code per programmer per day, the number of bugs found after the system is in production and the amount of time spent in each phase helps to prove the benefits of object-oriented technology to management.

Pick a target environment that is familiar

Don't pick a project that requires learning techniques that are over and above the methodology and the object-oriented tool. Project members may feel overwhelmed if they are expected to learn these techniques along with a new DBMS, TP monitor or programming language. If you expect too much of the team members, they will probably fail and the object-oriented tool and methodology will be a likely scapegoat.

Pick normal people for the project and not the superstars

If you do pick the best and the brightest, the resulting success will probably be attributed to the team and not the methodology and object-oriented tool. Make sure the people chosen are committed to learning the new techniques and using the tools.
Software Metrics: Management and Estimation

To the extent that bugs or design flaws can be tracked and measured, the resulting reduction in the cost of fixing them can also be measured. Also, by tracking how well software systems meet the functional requirements they were designed for, companies can measure the increase in overall quality of the systems they deliver to the user community.

Companies can begin by tracking the number of bugs or design problems occurring throughout a software project along with the cost to fix these problems. By assigning a dollar amount to the time required to fix these problems, the costs can be added to the overall cost to determine the "real cost" of developing the system. If this type of data is collected for projects that are not using object-oriented tools and techniques, these figures can be compared with projects that are using these tools and techniques, resulting in an increase in software quality that can be quantified in monetary terms.

Since development costs are directly related to the productivity of the development staff, these two potential benefits can be grouped together. Given that increased staff productivity leads to reduced development costs, the productivity of the development staff directly determines the cost of the system.

One problem inherent in measuring the productivity of software development professionals is that different people given the same identical requirements will deliver very different implementations of a program to satisfy these requirements. While this is not necessarily bad, it does make measuring productivity somewhat difficult and often very subjective.

Another problem companies face when beginning to think about measuring productivity is that they probably have no existing basis for performing this measurement. How can you measure and compare productivity using a structured technique and an object-oriented tool, if you haven't been measuring it all along? Therefore, prior to estimating and measuring the productivity of their staff and how quickly they can develop programs that satisfy these requirements.

While many companies use the lines of code metric, there are inherent problems with this form of productivity measurement. First and foremost, this metric is very dependent on the programming language used (3GL vs COBOL vs Assembler) and often disregards the maintainability and performance of the finished code. A program written in COBOL that allows records to be added to a VSAM file will be significantly smaller than one written in Assembler. Also, more lines of code may not mean a better performing program. If documentation within a program is included in the lines of code metric, it is often difficult to include this in the productivity metric. All of these factors lead to the limited applicability of the lines of code metric to accurately measuring productivity.

An alternative to the lines of code metric is the function point measurement, developed by A.J. Albrecht of IBM. This metric defines a function point as a measurable, external aspect of a software system. The five basic types of function points measured are inputs to a program, outputs from a program, user inquiries to a program, file or database updates and interfaces to other programs or systems.

Albrecht assigned weights to each of the five types of function points as follows:

- inputs to a program or system = 4 points
- outputs from a program or system = 5 points
- user inquiries to a program or system = 4 points
- file or database updates = 10 points
- interfaces to other programs or systems = 7 points

By applying these weights to specific programs under development, companies can track the productivity of their staff and how quickly they can develop programs that satisfy these functions. Since this metric is independent of the language used and disregards program performance and documentation, it does not suffer from the inherent limitations of the lines of code metric described above.

Companies such as Software Productivity Research, Corp. (SPR) of Cambridge, MA have collected empirical data over the years on productivity using function points. SPR defines the following national averages for productivity as defined by function points (FPs) reached per person month:

<table>
<thead>
<tr>
<th>Average Function Points per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of system</td>
</tr>
<tr>
<td>MIS/Commercial</td>
</tr>
<tr>
<td>Systems/Embedded</td>
</tr>
<tr>
<td>Military</td>
</tr>
</tbody>
</table>

If a company is already measuring the cost of software maintenance without an object-oriented tool, this information can be compared with the cost of maintenance with an object-oriented tool. In this fashion, a monetary figure can be assigned to the difference and the resulting decrease in maintenance cost can be measured. These cross life-cycle activities are also listed in table 17.
Software Metrics: Management and Estimation

To the extent that bugs or design flaws can be tracked and measured, the resulting reduction in the cost of fixing them can also be measured. Also, by tracking how well software systems meet the functional requirements they were designed for, companies can measure the increase in overall quality of the systems they deliver to the user community.

Companies can begin by tracking the number of bugs or design problems occurring throughout a software project along with the cost to fix these problems. By assigning a dollar amount to the time required to fix these problems, the costs can be added to the overall cost to determine the "real cost" of developing the system. If this type of data is collected for projects that are not using object-oriented tools and techniques, these figures can be compared with projects that are using these types of tools and the resulting increase in software quality can be quantified in monetary terms.

Since development costs are directly related to the productivity of the development staff, these two potential benefits can be grouped together. Given that increased staff productivity leads to reduced development costs, the productivity of the development staff directly determines the cost of the system.

One problem inherent in measuring the productivity of software development professionals is that different people given the same identical requirements will deliver very different implementations of a program to satisfy these requirements. While this is not necessarily bad, it does make measuring productivity somewhat difficult and often very subjective.

Another problem companies face when beginning to think about measuring productivity is that they probably have no existing basis for performing this measurement. How can you measure and compare productivity using a structured technique and an object-oriented tool if you haven't been measuring it all along? Therefore, prior to estimating and measuring the staff productivity using a structured technique and an object-oriented tool, companies should consider implementing measurement tools for the cost and productivity of existing systems and ongoing projects not using object-oriented tools.

Those companies already measuring and tracking their development costs and staff productivity typically use one of two metrics to do so:

1) the number of lines of code produced per person month
2) the number of function points achieved per person month

While many companies use the lines of code metric, there are inherent problems with this form of productivity measurement. First and foremost, this metric is very dependent on the programming language used (3GL vs COBOL vs Assembler) and often disregards the maintainability and performance of the finished code. A program written in COBOL that allows records to be added to a VSAM file will be significantly smaller than one written in Assembler. Also, more lines of code may not mean a better performing program. If documentation within a program is included in the lines of code metric, it is often difficult to include this in the productivity metric. All of these factors lead to the limited applicability of this metric to accurately measuring productivity.

An alternative to the lines of code metric is the function point measurement, developed by A. J. Albrecht of IBM. This metric defines a function point as a measurable, external aspect of a software system. The five basic types of function points measured are inputs to a program, outputs from a program, user inquiries to a program, file or database updates and interfaces to other programs or systems.

Albrecht assigned weights to each of the five types of function points as follows:

- inputs to a program or system = 4 points
- outputs from a program or system = 5 points
- user inquiries to a program or system = 4 points
- file or database updates = 10 points
- interfaces to other programs or systems = 7 points

By applying these weights to specific programs under development, companies can track the productivity of their staff and how quickly they can develop programs that satisfy these functions. Since this metric is independent of the language used and disregards program performance and documentation, it does not suffer from the inherent limitations of the lines of code metric described above.

Companies such as Software Productivity Research, Corp. (SPR) of Cambridge, MA have collected empirical data over the years on productivity using function points. SPR defines the following national averages for productivity as defined by function points (FPs) reached per person month:

<table>
<thead>
<tr>
<th>Average Function Points per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of system</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MIS/Commercial</td>
</tr>
<tr>
<td>Systems/Embedded</td>
</tr>
<tr>
<td>Military</td>
</tr>
</tbody>
</table>

If a company is already measuring the cost of software maintenance without an object-oriented tool, this information can be compared with the cost of maintenance with an object-oriented tool. In this fashion, a monetary figure can be assigned to the difference and the resulting decrease in maintenance cost can be measured. These cross life-cycle activities are also listed in table 17.

Copyright © 1991 by Foresite Systems Page 574

H. lines of code metric to accurately measuring productivity.

A. An alternative to the lines of code metric is the function point measurement, developed by A. J. Albrecht of IBM. This metric defines a function point as a measurable, external aspect of a software system. The five basic types of function points measured are inputs to a program, outputs from a program, user inquiries to a program, file or database updates and interfaces with other programs or systems.

Albrecht assigned weights to each of the five types of function points as follows:

- inputs to a program or system = 4 points
- outputs from a program or system = 5 points
- user inquiries to a program or system = 4 points
- file or database updates = 10 points
- interfaces to other programs or systems = 7 points

By applying these weights to specific programs under development, companies can track the productivity of their staff and how quickly they can develop programs that satisfy these functions. Since this metric is independent of the language used and disregards program performance and documentation, it does not suffer from the inherent limitations of the lines of code metric described above.

Companies such as Software Productivity Research, Corp. (SPR) of Cambridge, MA have collected empirical data over the years on productivity using function points. SPR defines the following national averages for productivity as defined by functions points (FPs) reached per person month:

<table>
<thead>
<tr>
<th>Average Function Points per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of system</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MIS/Commercial</td>
</tr>
<tr>
<td>Systems/Embedded</td>
</tr>
<tr>
<td>Military</td>
</tr>
</tbody>
</table>

If a company is already measuring the cost of software maintenance without an object-oriented tool, this information can be compared with the cost of maintenance with an object-oriented tool. In this fashion, a monetary figure can be assigned to the difference and the resulting decrease in maintenance cost can be measured. These cross life-cycle activities are also listed in table 17.

Copyright © 1991 by Foresite Systems Page 575
Being Successful with OO Technology

- Management must be involved
- Train the project team in the methodology and the tool(s)
- The project team leader must be a change agent & OO champion
- Get end users involved early
- Keep expectations realistic
- Move slowly and take small steps
- Make quality improvement a strategic goal

Being successful with object-oriented technology

Management must be involved

As with any other new technology or technique, to be fully successful, MIS management must be committed and actively involved in the process. Since structured methodologies represent an entirely new way of developing systems, it is critical that upper and middle-level management recognize the need for this commitment. Just selling management on the potential benefits is not enough. If management are informed of the practical expectations and benefits of these tools and techniques, they should also be willing to put some time and energy into successfully implementing them.

Train the project team in the methodology and the tool

Expect to have team members spend from 6 to 10 days learning to use the methodologies and the object-oriented tool. They will only become proficient when they can use the tools and techniques on a project. If team members are not trained and comfortable with the new techniques, they will feel frustrated and often revert to their old methods of development. Training should include technical and managerial staff and should cover the structured techniques, the object-oriented tool and managing the development process while using these tools.

The project team leader must be a change agent and the object-oriented technology champion

Since the success of any project is dependent on the individuals involved, it is essential to get the leader excited and who can put their ego on the line. This person should be respected and have the authority to implement the project in the time given. This person should also be actively involved in the evaluation and implementation and should have experience as a project manager. If this person is not convinced that SMs and object-oriented tools can be beneficial, the project is doomed to fail.

Get the end-users to cooperate and be actively involved

If possible, getting end-users involved in the process of developing systems and in using object-oriented tools effectively is also critical to being successful with object-oriented technology. End-users have a vested interest in the development process, and since most (if not all) object-oriented technology relies heavily on end-user input for prototyping and functional requirements.
Being Successful with OO Technology

- Management must be involved
- Train the project team in the methodology and the tool(s)
- The project team leader must be a change agent & OO champion
- Get end users involved early
- Keep expectations realistic
- Move slowly and take small steps
- Make quality improvement a strategic goal

Being successful with object-oriented technology

Management must be involved

As with any other new technology or technique, to be fully successful, MIS management must be committed and actively involved in the process. Since structured methodologies represent an entirely new way of developing systems, it is critical that upper and middle-level management recognize the need for this commitment. Just selling management on the potential benefits is not enough. If management are informed of the practical expectations and benefits of these tools and techniques, they should also be willing to put some time and energy into successfully implementing them.

Train the project team in the methodology and the tool

Expect to have team members spend from 6 to 10 days learning to use the methodologies and the object-oriented tool. They will only become proficient when they can use the tools and techniques on a project. If team members are not trained and comfortable with the new techniques, they will feel frustrated and often revert to their old methods of development. Training should include technical and managerial staff and should cover the structured techniques, the object-oriented tool and managing the development process while using these tools.

The project team leader must be a change agent and the object-oriented technology champion

Since the success of any project is dependent on the individuals involved, it is essential to get the leader excited and who can put their ego on the line. This person should be respected and have the authority to implement the project in the time given. This person should also be actively involved in the evaluation and implementation and should have experience as a project manager. If this person is not convinced that SMs and object-oriented tools can be beneficial, the project is doomed to fail.

Get the end-users to cooperate and be actively involved

If possible, getting end-users involved in the process of developing systems and in using object-oriented tools effectively is also critical to being successful with object-oriented technology. End-users have a vested interest in the development process, and since most (if not all) object-oriented technology relies heavily on end-user input for prototyping and functional requirements.
Keep expectations realistic

Don't expect object-oriented technology to solve all the problems of software development and don't expect to get something for nothing. To realize the benefits of object-oriented technology, you must put forth the effort and give the technology a chance to succeed. Realistic benefits that other companies have found from using object-oriented technology include better documentation, higher quality systems, better realization of business goals, better communication between team members & some overall improvement in productivity.

Move slowly and take small steps

Don't expect to be able to adopt a structured methodology that supports all phases of the SDLC along with an object-oriented technology (or tool) in a year or even two years. The process of finding the right technique and the supporting object-oriented tool can take several years. If you move slowly and use small steps as you become better, you will find your success will continue. If you attempt to do too much too soon, you will find that the tools and techniques will not support your efforts and your staff will become frustrated and burned out.

Make improvement a strategic goal

Along with the potential benefits of reduce development and maintenance costs, are overall improvements in the systems developed. This deliverable should not be overlooked, and can raise the visibility and status of the MIS department throughout the company. Delivering complete, quality systems to end-users on time is in itself a benefit.

Should you use OO technology?

While many organizations are talking about OO technology, very few are actually using this technology in a production environment.

Experiences with OO

From “OOA in the real world,” SDF 12, Michael Lee and Leon Starr

Projects : Size = 12 to 22 people, duration = 2 1/2 to 5 years (some ongoing)

Getting management support

Assess the match between objectives and capabilities
Demonstrate benefits early
Establish realistic expectations
Use the concept of “recursive planning” for analysis

Essential to the success of OOA

Start a document library with a single view of the model
Introduce technical documentation standards
Establish a standard format for analysis meetings (Whiteboard, brainstorming, etc.)
Put a software technician and CASE tool into place

When is OOA done?

When all the models are completed and integrated
When the models-to-date are sufficient to support a ‘good’ design (core of models in place)
When there is no more time
Keep expectations realistic

Don't expect object-oriented technology to solve all the problems of software development and don't expect to get something for nothing. To realize the benefits of object-oriented technology, you must put forth the effort and give the technology a chance to succeed. Realistic benefits that other companies have found from using object-oriented technology include better documentation, higher quality systems, better realization of business goals, better communication between team members & some overall improvement in productivity.

Move slowly and take small steps

Don't expect to be able to adopt a structured methodology that supports all phases of the SDLC along with a object-oriented technology (or tool) in a year or even two years. The process of finding the right technique and the supporting object-oriented tool can take several years. If you more slowly and use small steps as you become better, you will find your success will continue. If you attempt to do too much too soon, you will find that the tools and techniques will not support your efforts and your staff will become frustrated and burned out.

Make improvement a strategic goal

Along with the potential benefits of reduce development and maintenance costs, are over­all improvements in the systems developed. This deliverable should not be overlooked, and can raise the visibility and status of the MIS department throughout the company. Delivering complete, quality systems to end-users on time is in itself a benefit.

Should you use OO technology?

While many organizations are talking about OO technology, very few are actually using this technology in a production environment.

Experiences with OO

From “OOA in the real world,” SDF 12, Michael Lee and Leon Starr

Projects: Size = 12 to 22 people, duration = 2 1/2 to 5 years (some ongoing)

Getting management support

Assess the match between objectives and capabilities
Demonstrate benefits early
Establish realistic expectations
Use the concept of “recursive planning” for analysis

Essential to the success of OOA

Start a document library with a single view of the model
Introduce technical documentation standards
Establish a standard format for analysis meetings
(Whiteboard, brainstorming, etc.)
Put a software technician and CASE tool into place

When is OOA done?

When all the models are completed and integrated
When the models-to-date are sufficient to support a 'good' design
(core of models in place)
When there is no more time
Benefits of OOA

Creating OOA models tends to be a self-correcting process
OOD models deal well with dynamic behavior at different levels of abstraction
OOA supports versatility from an analysis and a design point of view
Scenarios executed using an OOA model can form the basis of a test plan
Composition is used vs. Decomposition

Results of the OOA pilot project

Fewer requirements changes
Fewer bugs
Phased delivery of the software
Software completed on schedule

Motivation for choosing OO

Adding functionality to Teamwork
Ease maintenance of 25K - 67K of C code
Reduce propagation of bugs in the code
Gain consistency of new features/functions across the editors
Get reusable, shared objects/code and reduce product size
(went from 25K - 67K of C --> 3K - 10K of C++)

Chose C++ - why? Pragmatic - workstation platform support
(only OOPL available was C++ from AT&T)

Unforeseen problems (primarily with C++)

Platform support was different across platforms and versions
C++ translators are not integrated into an OOE
Mapping C to C++ in debugging is difficult
Purging virtual tables is a manual process
Compile times almost doubled

Reuse unit is not the same - C to C++
In C++ the whole class library is reused, not just the object
Class hierarchies are not modular - takes time to get used to this fact
Visual navigation is more difficult with C++ than C or Ada
(leading to maintenance of C++ is more difficult)
"uses" relationship in configuration management are complicated and
must support "inherits" relationship
More up-front work is required in setting up data hierarchies and
methods to determine if something works
Benefits of OOA

Creating OOA models tends to be a self-correcting process
OOA models deal well with dynamic behavior at different levels of abstraction
OOA supports versatility from an analysis and a design point of view
Scenarios executed using an OOA model can form the basis of a test plan
Composition is used vs. Decomposition

Results of the OOA pilot project

Fewer requirements changes
Fewer bugs
Phased delivery of the software
Software completed on schedule

Motivation for choosing OO

Adding functionality to Teamwork
Ease maintenance of 25K - 67K of C code
Reduce propagation of bugs in the code
Gain consistency of new features/functions across the editors
Get reusable, shared objects/code and reduce product size
(went from 25K - 67K of C -> 3K - 10K of C++)

Chose C++ - why? Pragmatic - workstation platform support
(only OOPL available was C++ from AT&T)

Unforeseen problems (primarily with C++)

Platform support was different across platforms and versions
C++ translators are not integrated into an OOE
Mapping C to C++ in debugging is difficult
Purging virtual tables is a manual process
Compile times almost doubled

Reuse unit is not the same - C to C++
In C++ the whole class library is reused, not just the object
Class hierarchies are not modular - takes time to get used to this fact
Visual navigation is more difficult with C++ than C or Ada
(leading to maintenance of C++ is more difficult)
"uses" relationship in configuration management are complicated and
must support "inherits" relationship
More up-front work is required in setting up data hierarchies and
methods to determine if something works
Opinions

Does class inheritance support and preserve the benefits of encapsulation? Cohesion is sacrificed, functions become scattered among parent/child classes and this creates dependencies between a class' method, its super-class' method, and its sub-class' methods. Performing some level of OOA is critical. C++ requires a more robust environment than C - class hierarchies and libraries require browsing tools. Editing methods is complicated since the method may reside in many places in the class hierarchy. Debugging is difficult. Toolsets are still immature.

Conclusions

Training is essential in the OO programming paradigm. Tool selection requires scrutiny since the current set of products is immature (look very closely at multi-platform support). C++ is still evolving and you will be on the bleeding edge. No (current) accepted OOD technique. OO & C++ on small projects first then evolve the process.

Guidelines for identifying and defining classes and methods

Model with classes the entities that naturally occur in the problem domain.
Design methods with a single purpose.
Design a new method when faced with a choice of extending an existing one.
Avoid lengthy methods.
Store as instance variables the data that are needed by more than one method, or by a sub-class.
Design for the class library, not for yourself alone or your current application.

Common mistakes

Creating unneeded classes.
Declaring classes that are not classes at all.
Opinions

Does class inheritance support and preserve the benefits of encapsulation?
Cohesion is sacrificed, functions become scattered among parent/child classes
and this creates dependencies between a class' method, its super-class' method,
and its sub-class' methods
Performing some level of OOA is critical
C++ requires a more robust environment than C -
class hierarchies and libraries require browsing tools
Editing methods is complicated since the method may reside in
many places in the class hierarchy
Debugging is difficult
Toolsets are still immature

Conclusions

Training is essential in the OO programming paradigm
Tool selection requires scrutiny since the current set of products is
immature (look very closely at multi-platform support)
C++ is still evolving and you will be on the bleeding edge
No (current) accepted OOD technique
OO & C++ on small projects first then evolve the process

Guidelines for identifying and defining classes and methods

Model with classes the entities that naturally occur in the problem domain
Design methods with a single purpose
Design a new method when faced with a choice of extending an existing one
Avoid lengthy methods
Store as instance variables the data that are needed by more than one method,
or by a sub-class
Design for the class library, not for yourself alone or your current application

Common mistakes

Creating unneeded classes
Declaring classes that are not classes at all
Object-Oriented Technology Reading List

Object-Oriented Analysis & Design


Booch, Grady, Software Engineering with Ada, Benjamin/Cummings, 1983.


Coad, Peter and Yourdon, Ed, Object-Oriented Analysis, Prentice Hall, 1990.


Object-Oriented Programming


Goldbert, A and Robson, D, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.


Copyright © 1991 by Foresite Systems

Page 584

Object-Oriented DBMS


Related Topics


DeMarco, Tom, Controlling Software Projects, Yourdon Press, 1982.


Object-Oriented Technology Reading List

Object-Oriented Analysis & Design


Booch, Grady, Software Engineering with Ada, Benjamin/Cummings, 1983.


Coad, Peter and Yourdon, Ed, Object-Oriented Analysis, Prentice Hall, 1990.


Object-Oriented Programming


Goldberg, A and Robson, D, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.


Copyright © 1991 by Foresite Systems

Object-Oriented DBMS


Related Topics


DeMarco, Tom, Controlling Software Projects, Yourdon Press, 1982.


Copyright © 1991 by Foresite Systems
Evaluating & Implementing Software Engineering


**CASE Tools and Software Engineering**


---

**MODULA-2 IN EMBEDDED SYSTEMS**

Christian Vetterli
Hiware AG
Basel, Switzerland

Claude Vonlanthen
Hiware AG
Basel, Switzerland

Christian Vetterli studied under Niklaus Wirth, writing his thesis on the object-oriented expandable document editor OPUS (Object-Oriented Publishing System) written in Modula 2. Dr. Vetterli is responsible for the Modula-2 tool MacMETH (loader and libraries for Apple Macintosh), libraries for Modula-2 development systems on IBM-RT (RISC), and the code generator for the Modula-2 compiler. He is responsible for developing embedded systems software for Hiware.

Claude Vonlanthen is responsible for applications software at Hiware. A graduate engineer from Eigenossische Technische Ilochschule, Zurich, Switzerland, Claude was responsible for developing a Modula-2 embedded system at a large machine factory before joining Hiware.