Extending UML to Model and Roundtrip Engineer Scalable System Designs

© Leszek A. Maciaszek, Bruc Lee Liong
Macquarie University, Sydney, Australia

Topics
- What is maintainable and scalable system?
- The BCEMD framework
- Tutorial case study
- The Cumulative Class Dependency (CCD)
- The Cumulative Message Dependency (CMD) and Cumulative Event Dependency (CED)
- Managing Forward and Reverse Engineering
- Conclusions

The theme - Laws of ecology

The first law of ecology: Everything is connected to everything else.

The second law of ecology: Everything must go somewhere.

The third law of ecology: Nature knows best.

Faber's “The book of laws”

The nature of software

"Software is a place where dreams are planted and nightmares harvested, where terrible demons compete with magical panaceas, a world of warware and silver bullets."

- Brad Cox

Software development invariant

- Software production is an art
  - Software is developed, not manufactured
  - … but
    - OT & re-use
      - reuse from libraries
      - reuse from frameworks
    - … but what about core business?
    - Component technology and distributed object systems

Complexity

- that property of a system “which makes it difficult to formulate its overall behaviour even when given almost complete information about its atomic components and their inter-relations”
  - Bruce Edmonds

- Complexity only makes sense when considered as relative to a given observer:
  - System complexity
    - the complexity of the system w.r.t. the observer
  - Observer complexity
    - the complexity of the observer w.r.t. the system

Complexity only makes sense when considered as relative to a given observer:
- System complexity
  - the complexity of the system w.r.t. the observer
- Observer complexity
  - the complexity of the observer w.r.t. the system

© L.A. Maciaszek, B.L. Liong
1-October-2002
Concepts related to complexity

- Size
- Ignorance
- Minimum description size
- Variety
  - Order and disorder

Complexity and evolution

- There is no one appropriate measure of complexity appropriate to evolution in general
- There is a tendency for complexity to increase over evolution
  - Evolution in computer lingo:
    - adaptive maintenance
      - monitoring and auditing of an operational system with the aim of making diagnoses and adapting the system in order to perform better
    - scalability
      - re-designing and re-implementing an operational system in order to satisfy the additional or changing requirements and to respond to changing environment

Guidance comes from natural systems

- Premise 1:
  - We have difficulty handling the complexity of computer systems
- Premise 2:
  - Natural systems are the most complex systems that work
- Conclusion:
  - Why not model computer systems on natural systems

Objects as holons

- The guidance for a new paradigm for software production (object technology) comes from natural systems
- Software object is a holon
  - holon (from the Greek word: holos = whole and with the suffix on suggesting a part, as in neutron or proton)
- Software object has identity, state and behavior
- Software objects collaborate by requesting, inheriting or composing services of other objects

Static and dynamic holocracies

- Hierarchies can be
  - Static
    - structure does not change much over time but behavior can moderately change
    - most often found in nature (biological, physical, chemical classifications)
  - Dynamic
    - common in business applications (company structures, product and employee classifications, plant operation specifications)
- Inheritance model works poorly on dynamic classifications

Where to start our inquiry?

- Let’s start with the seemingly abstract yet fundamental problem of the relations between the whole and its parts - any „whole“, whether it be the universe or human society, and any „part“, whether an atom or a human being.
- „parts” and „wholes” in an absolute sense do not exist
  - a living organism is a hierarchy of sub-wholes or „holons“ (organs, nerves, muscles, cells, etc.)
  - each member of this hierarchy, on whatever level, is a structure with considerable autonomy that performs some service to the organism as a whole
  - a hundred years ago, Alexis Carrel showed that a minute strip of tissue taken from a heart of a chicken embryo and put into a nutrient solution will go on pulsating for years
  - the same tissue within the organism obeys a built-in code of rules (canon)
- environmental effects can affect or disrupt the canon

[Edmonds, 1995]

[Koestler, 1978; 1980]
Hierarchies do not operate in a vacuum

- Holons are hierarchically layered according to complexity
  - e.g. in a biological organism we can distinguish a hierarchy of atoms, molecules, organelles, cells, tissues, organs, and organ systems
  - Such hierarchies of holons are called holocracies
- Hierarchies combine in systems of cooperating objects
  - The trees in a forest are vertical structures
  - The meeting points of branches from neighboring trees form horizontal layers
  - Without the trees there could be no entwining, and no layers
  - Without the layers, each tree would be isolated, and there would be no integration of functions

[Horst, 1978; 1980]

Objects can be modeled on holons

- Objects are software „behavioural structures” similar to holons
- Objects provide services while hiding implementation details
- Objects are combined hierarchically in software systems - holocracies
- Objects are composed into systems via:
  - Specialization/Inheritance of services
  - Composition/delegation to do a service
  - Permanent associations via object Identifiers (OID)
- Object with OID is distributed (can be located anywhere) and “flying” (can be moved any time)

Theory and practice of inheritance

- In theory inheritance allows:
  - incremental specifications,
  - exploitation of common properties between classes,
  - localization of modifications.

- In practice inheritance:
  - is frequently overused and abused,
  - leads to systems that are difficult to understand and maintain,
  - a new category of legacy systems, programmed in C++ rather than Cobol, is being created.

Inheritance and reuse

- Inheritance supports re-use but at the same time it opens the proverbial “can of worms”
- Consider the following questions:
  - Should the creator of a class be prevented from changing the definition of the class, once any other user derived a subclass from the class?
  - Should the creator of a class be given access privileges to all direct and indirect subclasses of the class?
  - What naming conventions need to be used and imposed on class developers so that to avoid future name clashes due to subclassing?

Inheritance and databases

- Inheritance conflicts with databases
- Databases are designed to store and manage lots of objects belonging to relatively small number of classes
- Inheritance uses class specialization – many new classes, few objects
- If specialization is used as the main modeling construct, the database intention grows quickly to the size that is unmanageable and unproportionally large to its extension

No dynamic and multiple classification

- In most OO-implementation languages, objects cannot:
  - change class
    - no dynamic classification
  - belong to more than one class
    - no multiple classification
- The repercussions are severe whenever objects persist longer than a single execution of a single program.
Inheritance and query capability

- In practice, it is impossible to declare all data to be private
- Users accessing databases by means of SQL or SQL-like ad-hoc query language raise justified expectations that they want to directly refer to data members in the queries, rather than to be forced to work with some data access methods that make query formulations more difficult and more error-prone
  - This requirement is particularly strong in data warehouse applications dominated by OnLine Analytical Processing (OLAP) queries.

Inheritance and encapsulation

- Encapsulation is orthogonal to inheritance
  - Inheritance compromises encapsulation by requiring protected data members
  - Computations spanning objects belonging to different classes require frequently that different classes be "friends" of each other, thus further breaking encapsulation
  - Occasionally, static data members, global to all objects of the class, have to be used
  - And, one has to realize that inheritance and encapsulation refer to the notion of the class, not the object - an object cannot hide anything from another object of the same class.

Advantages of aggregation/delegation

- Whenever a composite object cannot complete a task by itself it can call on the methods in one of its component objects
- Functionality implemented by including (cloning) the functionality of existing objects in the newly required functionality
- Better updatability in changing environments (sharing and reuse can be determined dynamically at run-time)
- More natural (objects are naturally combined to form larger solutions and can evolve in an unanticipated way)

Programming environments ignore aggregation

- In programming environments (including most object databases), aggregation is implemented in the same way as conventional relationships - by acquiring references between composite and component objects
  - Although compile-time structure of aggregation is the same as for conventional relationship, the run-time behavior is different
  - The semantics of aggregation is stronger and it is the programmer's responsibility that run-time structures obey this semantics

Aggregation deserves more emphasis

- It has the potential of replacing inheritance as a pivotal modeling concept
- Systems structured according to aggregation are simpler, easier to understand, and capable to assimilate future changes while offering the same, if not superior, functionality
- Aggregation can facilitate reuse while at the same time enhances scalability and maintainability of systems
  - These strengths of aggregation have now been recognized in design patterns
  - Additionally, aggregation provides a natural way of structuring multimedia objects

Holocracies and aggregation

- The hierarchical composition in natural systems can be seen as having a large number of extensively interlinked objects, but few classes
  - There are thousands or millions of objects of class "molecule" that combine to form an object of class "organella"
  - The solution to complexity may be in reducing the network structures through the use of multi-layered object hierarchies similar to hierarchies of holons (so called "holocracies")
  - by restricting the number of potential interaction paths between classes
  - by stratifying classes into layers and imposing strict rules on class interactions within a layer and between layers
  - Aggregation can model large quantities of objects by grouping them and establishing part-whole relationships between them and placing them on adjacent stratified layers
  - higher aggregation layers hide (encapsulate) the complexity of object interactions in lower layers

[Maciaszek et al., 1996]
MVC - Model, View, Controller

View

“in the head”

“on the glass”

“under the hood”

Model

Controller

“in the head”

“on the glass”

“under the hood”

BCEMD approach

Boundary Package

Control Package

Entity Package

DbInterface Package

Mediator Package

Prefixes for class names: B, C, E, M, D

Meaning of B-C-E-M-D

- The Boundary package
  - classes that define GUI objects.
- The Control package
  - classes responsible for the program’s logic, algorithmic solutions, and main computations.
- The Entity package
  - classes representing “business objects”
- The DBInterface package
  - responsible for all communications with the persistent data store
- The Mediator package
  - creates a level of independence between entity and dbinterface objects and relieves the program’s logic (control objects) from necessity to directly communicate with the dbinterface objects if entity objects are not available in program’s memory

Consider non-BCEMD design

Primary BCEMD principles

- Class Naming Principle (CNP)
  - class name prefixed with a single letter identifying its package
    - E_Invoice instead of Invoice
- Package Communication Principle (PCP)
  - package can only communicate directly with its neighbor package
- Implementation Inheritance Principle (IIP)
  - cross-package implementation inheritance is disallowed
Secondary BCEMD principles

- **Explicit Association Principle (EAP)**
  - Inter-package communication can only be conducted via explicit association/aggregation links between two classes or a class and an interface

- **Dominant Class Principle (DCP)**
  - Inter-package associations (as per EAP) link dominant classes or a dominant class and a dominant interface

- **Acquaintance Relationship Principle (ARP)**
  - Communication between classes acquainted at run-time must adhere to other BCEMD principles

BCEMD with no circular dependencies

Is this possible?

Circular dependency between packages

Breaking the cycle with interface

That's why "interfaces are often included in the package that uses them, rather than in the package that implements them"

Boundary is now independent from control

BCEMD layers and partitions

- A complex system may be divided into manageable modules by using both horizontal layers and vertical partitions
- Each partition (package) may be layered and each layer may be partitioned
- A layer at the top of the hierarchical composition contains all five packages but the layers underneath may contain any subset of the five packages
- Most of this tutorial assumes a single layer of five BCEMD packages

Multi-layer BCEMD holocracy
**Example of component layer**

- **Boundary**: B_LoginForm, B_ConfirmationForm, B_QueryForm
- **Control**: C_InputValidator, C_NextWindowGenerator
- **Data Interface**: D_PreferenceReader, D_ScreenDumper
- **Entity**: E_PasswordBean, E_MessageList, E_Preferences
- **Mediator**: M_Session

**Tutorial case study**

**EMS – Email Management System**
- part of an Electronic Document Management System
- capture, indexing, classification and retrieval of all email business correspondence

**Iteration 1 - composing and emailing of query messages**
- stored in the database and scheduled to be sent to contacts (customers)
- a simple console-based Java application that accesses an Oracle database via JDBC
- upon employee’s login to the database, the application can retrieve from the database and display the list of queries (messages) to be emailed to contacts
- the employee can then request an automatic emailing of chosen messages
- after a successful email, the database is updated accordingly

**Execution screen shot 1**

**Execution screen shot 2**

**Email received**

**Scaling up**
### Class model for business objects

- **Employee**
  - employee_id : String
  - first_name : String
  - last_name : String
  - login_name : String
  - employee_email : String

- **Contact**
  - contact_id : String
  - organization : String
  - first_name : String
  - last_name : String
  - contact_email : String

- **QueryMessage**
  - message_id : String
  - message_text : String
  - date_created : Date
  - date_emailed : Date

- **Employee**
  - employee_id : String
  - first_name : String
  - family_name : String
  - login_name : String
  - employee_email : String

- **Contact**
  - contact_id : String
  - organization : String
  - first_name : String
  - family_name : String
  - contact_email : String

### Class model for BCEMD

- **C_Init**
  - C_Init
    - M_Broker
    - E_Employee
    - E_QueryMessage
    - E_Contact
    - D_Reader
    - D_Updater

### Instantiation dependencies between classes

- **C_Init**
  - B_Console
  - C_Actioner

- **M_Broker**
  - C_Init

- **E_Employee**
  - E_QueryMessage
  - E_Contact

- **D_Reader**
  - D_Updater

### Behavioral collaboration (excerpt)

- **B_Console**
  - B_Console
    - B_Console
      - M_Broker
        - E_Employee
        - E_QueryMessage
        - E_Contact
        - D_Reader
        - D_Updater

### Structural collaboration (excerpt)

- **C_Init**
  - C_Init
    - B_Console
    - E_Employee
    - E_QueryMessage
    - E_Contact

### Cumulative class dependency

- **numerical value that characterizes the relative difficulty of maintaining and evolving an object-oriented system**
- **represents the complexity of all explicitly defined relationships between classes**
  - associations and aggregations (represented by instance variables in class definitions) and inheritance relationships
  - an extension of CCD includes also the cost of acquaintance relationships
    - when a client object C passes the handle of another object O to a supplier S as one of the arguments of the message

- **CCD calculation is tightly integrated with the BCEMD framework, which forces explicit definition of relationship link**

[Maciaszek and Liong, 2002]
**CCD definition**

DEFINITION: Cumulative class dependency (CCD) is the total maintenance cost over all classes $C_i$ in a system of the number of classes $C_j$ to be potentially changed – according to statically-defined relationships between classes in the immediate neighborhood – in order to modify each class $C_i$. The CCD is not increased by a modification to a class $C_i$ if the nature of a relationship link to another class $C_j$ ($C_i \neq C_j$) does not introduce a maintenance dependency. The immediate neighborhood condition eliminates double calculation.

[Maciaszek and Liong, 2002]

**CCD computation**

- Can be done in few of stages
  - class dependency (CD) – “how many classes this class depends on?”
  - class dependency for package (CDP) – the sum of CDs for all classes in a package
  - CCD is the sum of CDPs for all packages in the system
- The BCEMD framework reduces CCD by disallowing relationships between “non-neighboring” packages

[Maciaszek and Liong, 2002]

**Minimizing CCD with interfaces**

- For a non-BCEMD design, CCD would be 36
- Maintainability Growth Factor (MGF) would be 36/20 = 1.8
- What is CCD for our earlier case study model with 9 classes?

[Maciaszek and Liong, 2002]

**Minimizing CCD with composition**

- Mediator acts as a dominant class and encapsulates subset classes
- Changes to subset classes do not affect superset class (except for changes to operation signatures)

[Maciaszek and Liong, 2002]

**Mixing composition and inheritance**

- Composition allows composing behaviors at run-time
- $D_{DBInterface}$ object can be retrieving from a database via JDBC calls or SQLJ statements by replacing its $D_{Selector}$ instance with $D_{JDBCSelector}$ or $D_{SQLSelector}$

[Maciaszek and Liong, 2002]
Composition can increase CCD

Composition can increase CCD

The price for the possibility of composing behavior at run-time is a significant increase in CCD.

Acquaintance inside package

Acquaintance between package

Acquaintance between package

Cumulative message dependency

Like CCD, CMD provides a numerical value that characterizes the relative difficulty of maintaining and evolving an object-oriented system.

Unlike CCD, CMD relates to the dynamic run-time behavior of programs.

Message originator (client object) usually does not know the specific receiver of the message (supplier object) until runtime:
- Our task is not to trace any specific scenario of program’s execution.
- Our aim is to discover all possible dependencies between clients and suppliers of services so that we can conduct an impact analysis caused by a service (method) change.

BCEMD framework lowers the CMD cost to a manageable size.

Delegation in CMD

Delegation in CMD

Responsive delegator:
- An object can delegate the authority to perform the work but it is not relieved from a contractual responsibility for the work.
- If a responsible delegator object delegates the work to an object in another package then the cost of inter-package dependency is carried by the responsible delegator.
- Further delegation sequence does not result in an additional cost (non-responsive delegators do not carry a maintainability cost).

Therefore, we can concentrate in CMD calculations on methods in the immediate neighborhood of the client object.
- Corresponds loosely to the degree of separation equal to one in Small Worlds.
CMD definition

DEFINITION: Cumulative Message Dependency (CMD) is the total maintenance cost over all Synchronous Messages SMi within client objects of the costs associated with changes to methods Mj in supplier objects or responsible delegator objects that are responsible for servicing SMi. To avoid double calculation, only “first-call” methods Mj (i.e. immediate neighbours) are considered. If any SMi in the system invalidates the BCEMD framework, the CMD for all inter-package messages SMi is increased by the Maintainability Growth Factor (MGF).

CMD computation

Five steps:
- Message Dependency (MD)
  - for each message of a method of a class
- Message Dependency for Method (MDM)
  - for each method of a class; the sum of all MDs of the method
- Message Dependency for Class (MDC)
  - the sum of all MDMs of the class
- Message Dependency for Package (MDP)
  - the sum of all MDCs within the package
- CMD
  - the sum of all MDPs in the system

CMD computation

Message Dependency (MD) is calculated by considering the method’s body:
- one (1) for each message to a method within the same class,
- one (1) for each message to a supplier or responsible delegator within the same package (or an API class),
- one (1) for each message to a static final method (a method in a utility class),
- two (2) for each message to a supplier or responsible delegator in another package,
- zero (0) for each message to a non-responsible delegator within the same or another package.

CMD = 5 (the code)

public class C_Actioner { // client of E_Employee
E_Employee emp;
public void do1() {// MD = 2
emp.do3();
}
public class E_QueryMessage { // client of E_Employee
E_Employee emp;
public void do2() {// MD = 1
emp.do3();
}
public class E_Employee { // responsible delegator
M_Broker brk;
public void do3() { // MD = 0
brk.do3();
}
public class M_Broker { // delegator
D_Updater upd;
public void do3() {// MD = 0
upd.do3();
}
public class D_Updater { // supplier
public void do3() { // MD is null
// code for do3
}
}

CMD = 5 (not BCEMD; MGF = 2.4)

CMD = 5.8 (not BCEMD; MGF = 2.4)
DEFINITION: Cumulative Event Dependency (CED) is the total maintenance cost over all methods containing “fire event” messages PEi plus over all methods containing “process event” messages PE, within publisher objects plus over all methods servicing these “process events” SE, within subscriber objects. The PE, maintenance cost is associated with changes to signatures of SE, methods. The SE, maintenance cost is associated with changes to messages in the bodies of PE, methods. Messages within registrator objects as well messages contained in bodies of SE, methods are excluded as they are computed as part of the CMD calculation. If any of the two rules for CED are found to be broken, the system is deemed to invalidate the BCEMD framework and the CED for all inter-package event dependencies is increased by the Maintainability Growth Factor (MGF).

CED vs CMD

In CMD, if object A sends a message to object B, then A depends on B because A expects some results out of the execution of B
In CED this translates to a publisher being dependent on the subscriber
Because publisher has no knowledge how the subscriber processes the event, the dependency is weaker but it exists nevertheless
Publisher depends on the signature of the subscriber’s method that processes the event
Capturing this dependency is additionally motivated by the publisher and subscriber executing in separate threads and, therefore, introducing additional maintenance risk.
- As opposed to CMD, where – for each message – the client object depends on the supplier object but not vice versa, in CED the dependency is both-directional - the subscriber object depends on the publisher object
- Additionally, in CED, the method containing the “fire event” message depends on the method that services it

CED naming conventions

Methods performing registration of subscribers to a publisher are named beginning with the phrase add and ending with the phrase Listener
ame of the event object is placed in-between these two phrases, e.g. addXXXListener, where XXX is the name of the event object
because usually subscribers are passed to the registration methods in arguments typed as interfaces, further program analysis is required to determine the class of a subscriber
- Each event method begins with the phrase fire, such as fireCommandButtonEvent
- fire method goes through the list of subscribers and, for each subscriber, calls its associated process method
- as a matter of convenience (rather than a rule), a name of the process method may begin with the phrase process, such as processCommandButtonEvent()

CED computation

Four steps:
- Event Dependency (ED)
  - for methods containing fire messages and for fire and process methods of a class
- Event Dependency for Class (EDC)
  - the sum of all EDs of the class
- Event Dependency for Package (EDP)
  - the sum of all EDC within the package
- CED
  - the sum of all EDPs in the system
CED computation

- one (1) for each fire message to a publisher fire method when the message and the method are in the same class or in the same package
- two (2) for each fire message to a publisher method when the message and the method are in neighboring packages
- one (1) for each publisher’s process message to a subscriber method when the publisher and subscriber are in the same package
- two (2) for each publisher’s process message to a subscriber method when the publisher and subscriber are in neighboring packages
- one (1) for each subscriber method’s dependency on the publisher when the publisher and subscriber are in the same package,
- two (2) for each subscriber method’s dependency on the publisher when the publisher and subscriber are in neighboring packages,
- if the subscriber is an interface then ED costs in points 5 and 6 are replaced by the costs of interface inheritance (• a subscriber implementing the interface depends on it; this cost is added to CED but excluded from the calculation of another metric called Cumulative Inheritance Dependency (CID))

CED = 5

CED = 5 (the code)

```
public class C_DisplayEventRegistrator {
    B_Console console; //the publisher
    C_Actioner actioner; //the one and only subscriber
    public void init() {
        console.addDisplayEventListener(actioner);
    }

    public class C_Actioner{
        public void processDisplayEvent(B_DisplayEvent evt)
        {
            System.out.println("Display event received");
        }
    }
```

CED = 7

CED = 8
Acquaintance relationships

- When a client object C passes the handle of another object O to a supplier S as one of the arguments of the message signature
- CMD vs acquaintance relationships
  - C depends on S in the computation of CMD
- CCD vs acquaintance relationships
  - S depends on O in the computation of CCD
- BCEMD vs acquaintance relationships
  - If S and O are not from neighboring packages BCEMD is broken
  - This can be avoided by implementing acquaintance via interfaces

Acquaintance in BCEMD

- \texttt{C\_ActionableItem}
- \texttt{void initializeDisplay()}
- \texttt{ArrayList items = retrieveEmployees();}
- \texttt{browser.displayItems(items);}
- \texttt{void displayItems(ArrayList items)}
- \texttt{for(int i=0;i<items.size();i++)}
- \texttt{C\_ActionableItem item = (C\_ActionableItem) items.get(i);}
- \texttt{performDisplay(item);}

Integrating metrics with UML

- Issues:
  - What metrics
  - Which UML elements will have metrics
  - How to extend UML and how to visualize the metrics
  - How to obtain metrics
    - From the design
    - From the parsed source code
    - From the byte/executable code
  - How to visualize the quality and maintainability of the design

What metrics

- Class dependency metrics
  - Including acquaintance
- Message dependency metrics
- Event dependency metrics
- Inheritance dependency metrics
- Summary, statistics and impact metrics akin of SmallWorlds

Which UML elements

- Message
  - Synchronous
  - Asynchronous (event)
- Method
- Class
- Interface
- Package
- System

How to extend UML

- Stereotypes not useful
- Apply to existing UML elements, but cannot take different values for the same element
- A new UML element called “metric” needed
- Constraints could be used but they are not available for all UML elements and not well-supported by visual modeling tools
- Notes can be used for methods, classes, and packages, but visually awkward
- 4th compartment in classes specifically for metrics (akin of TogetherControl Centre)
- Metric property (field) in repository dialog boxes
- SmallWorlds-like interface
How to obtain metrics

- From UML design
  - CCD if enforced by compulsory associations as in BCEMD
  - CID for interface and implementation inheritance
- From the code (all metrics, incl. CMD & CED)
  - From parsed source code (by implementing a compiler capability) or
  - From byte code (reflection)

What if analysis of the design

Global dependencies

Round-trip engineering

- The combination of
  - forward engineering
    - generation of code from the design
  - reverse engineering
    - recovery of design from the code
- Required for
  - client application programs
  - server database programs

Round-trip engineering with Client

Round-trip engineering with DB
Reverse engineering of metrics

Summary
- Successful development of scalable solutions requires:
  - the enforcement of an architectural design (BCEMD or similar),
  - modeling software on structure and behavior of natural systems, as much as possible
  - metrics to measure complexity, maintainability and scalability of software solutions
  - mature development process and strong management
  - skillful use of roundtrip-engineering tools

Conclusion
A good education should leave much to be desired.
- Alan Gregg

References
- MARTIN, R.C. (2000): Design Patterns and Design Patterns,
  - Design Patterns and Design Patterns: (accessed August 2002), 34p.
- POWERDESIGN (2002): PowerDesigner 9.0,
- RATIONAL (2002): Rational Rose 2002,
- SMALLWORLDS (2002): SmallWorlds 2.0,
  - SmallWorlds 2.0: (accessed August 2002), 34p.
- TOGETHER (2002): Together ControlCenter 6.0,
  - Together ControlCenter 6.0: (accessed August 2002), 64p.
- TOGETHER (2002): Together ControlCenter 6.0,
  - Together ControlCenter 6.0: (accessed August 2002), 64p.

References (cont.)