ELEC 875
Design Recovery
and
Automated Evolution

Week 1 Class 1
Introduction

System Evolution

- Real systems evolve over time
  - ◊ not just bug fixes
  - ◊ environment changes over time
  - ◊ new/old features
  - ◊ legacy systems

- Design Recovery
  - ◊ Recover design level facts about software artifacts

- Automated Evolution
  - ◊ semi-automated changes to systems

Course Structure

- 5.5 weeks of lectures
  - ◊ background material (readings)
  - ◊ basis
- Midterm (25%)
  - ◊ based on lectures
- Advanced Readings + TXL
  - ◊ reports (30%) and discussion (15%)
- Project (30%)
  - ◊ Project Presentation
  - ◊ TXL

Legacy

noun A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor

adj associated with something that is outdated or discontinued

Legacy Systems

- Software
  - ◊ *inherited* (more than one generation of developers)
  - ◊ *valuable*
    - – significant resources to replace
    - – significant *risk* to replace
- Problems:
  - ◊ original developers may not be available
  - ◊ older development methods used (outdated?)
  - ◊ extensive modifications
  - ◊ missing or outdated documentation
  - ◊ studies show 50% – 75% of available effort (domain dependent)

Legacy Systems

- Traditionally viewed as old and expensive
  - ◊ *prohibitively expensive*
  - ◊ only a matter of time before they must be replaced
  - ◊ drain on resources
  - ◊ outdated

Legacy Systems

- Alternate View:
  - ◊ crown jewels
  - ◊ organizations that have not let their legacy systems get out of control (i.e. most large financial institutions) have a significant advantage over other organizations
  - ◊ system is working and evolves

Legacy Systems

- Continuous Evolution
  - ◊ You own a wooden ship. You replace each board in the ship each time you sail. At what point in time do you have a new ship?
  - ◊ Ship of Theseus

  - ◊ Space Shuttle
  - ◊ Operating Systems
  - ◊ Compilers
  - ◊ Financial Systems (systems written in 1962 are still running).

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

◊ source code

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

- ◊ source code
- ◊ database definitions

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

◊ source code

◊ database definitions

◊ screen definitions (also web page definitions)

◊ communication definitions

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

- ◊ source code
- ◊ database definitions
- ◊ screen definitions (also web page definitions)
- ◊ communication definitions
- ◊ stored procedures

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

- ◊ source code
- ◊ database definitions
- ◊ screen definitions (also web page definitions)
- ◊ communication definitions
- ◊ stored procedures
- ◊ scripting languages (JCL, TCL, Shell, DOS BAT)

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

- ◊ source code
- ◊ database definitions
- ◊ screen definitions (also web page definitions)
- ◊ communication definitions
- ◊ stored procedures
- ◊ scripting languages (JCL, TCL, Shell, DOS BAT)
- ◊ some forms of documentation

Design Recovery

- Recover Design Information from Source Artifacts.

Source Artifacts:

- ◊ source code
- ◊ database definitions
- ◊ screen definitions (also web page definitions)
- ◊ communication definitions
- ◊ stored procedures
- ◊ scripting languages (JCL, TCL, Shell, DOS BAT)
- ◊ some forms of documentation
- ◊ 4GL languages (application generation)

Resources

- Conferences:
  - ◊ IEEE International Conference on Software Maintenance (ICSM)
  - ◊ IEEE Working Conference On Reverse Engineering (WCRE now SANER))
  - ◊ European Conference On Software Maintenance and Reengineering (CSMR now SANER)
  - ◊ IEEE International Conference on Program Comprehension (ICPC)
  - ◊ IEEE International Conference On Software Engineering (ICSE)
  - ◊ Foundations on Software Engineering

Resources

- Journals

- Web
  - ◊ Authors Web Pages:
    – Dr. Timothy Lethbridge (SITE, U of Ottawa)
    – Dr. Hausi Müller (CS, U of Victoria)
    and many others (check references in articles)

  - ◊ http://citeseerx.ist.psu.edu/
  - ◊ google scholar

Papers for next week

- Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N., "An Examination of Software Engineering Work Practices", *CASCON '97*, Toronto, October, pp. 209-223.

- Lethbridge, T. and Singer, J. (1997), "Understanding Software Maintenance Tools: Some Empirical Research", *Workshop on Empirical Studies of Software Maintenance (WESS 97)*, Bari Italy, October, pp. 157-162.

- R. Ferenc, S. Sim, R. Holt , R. Koschke, T. Gyimóthy, "Towards a Standard Schema for C/C++", *8th Working Conference On Reverse Engineering (WCRE'01)*, Stuttgart, Germany, October, pp. 49-58.

Biggerstaff - Introduction

- “Design Recovery For Maintenance and Reuse", *IEEE Computer*, 22(7), July 1989, pp. 36–99
- Seminal Paper
  - ◊ Discusses the General Goal
  - ◊ Prototype: Desire - first step towards the goal
- Design Recovery Already Happens
  - ◊ “a common, sometimes hidden part of many activities scattered throughout the software life cycle”
- Domain Expertise - Domain Model
  - ◊ Tools need to abstract domain knowledge as well.

Biggerstaff

- Design recovery whenever a system is maintained
- Several Steps
  - ◊ Program Understanding
    - \- Modules
    - \- Key data items
    - \- Software engineering artifacts
    - \- Informal design abstractions
    - \- Relate SE artifacts and informal abstractions to the code
  - ◊ Population of Reuse and Recovery Libraries
  - ◊ Applying Results of Design Recovery

Identify the Modules

- Not all languages have modules
- software of any size has modules
- variety of ways to implement modules
  - ◊ separate files and compilation units
    - module.h module.c
    - no nested modules
    - smaller modules (one file)
    - may be more than one implementation file
      - e.g. module1.c module2.c
- naming convention for type, procedure or variable names

Key Data Items

- Most programs are organized around one or more specific data items.
  - ◊ Master journal record in transaction systems
  - ◊ Master account database
  - ◊ Ready, wait and device queues in operating systems
- These data items are some abstraction of the problem domain. What are they?
  - ◊ Customer, Sale, Deposit, Process
- How are they related to the modules
  - ◊ SA&D vs ADTs
  - ◊ Functional Decomposition vs OO

SE Artifacts

- The result of Design Recovery (as expressed by Biggerstaff) are design artifacts
  - ◊ dependent on shop
  - ◊ PDLs, Dataflow, Data Dictionary
  - ◊ UML?
- Does not have to match the artifacts originally used to create the system
- Artifacts must be appropriate for system
  - ◊ Consequences of a poor fit?
  - ◊ UML for 40 year old transaction system

Informal Design Abstractions

- Informal descriptions of concepts that occur in the code (automatable?)
- Design Rational
- Original Designers are not available, or it may be so long that they do not remember
  - ◊ People's version of history change over the years
  - ◊ Guess
  - ◊ Source Code Comments
  - ◊ Existing Documentation

Relating Abstractions to Code

- Link the recovered design back to the code
- Which functions are part of which module?
- Which files are part of a UML class?
- Which data structure represents a particular informal concept

- Necessary to answer low level questions that have been abstracted out
  - ◊ needed in order to use the system
  - ◊ not designing systems from scratch, modifying existing systems.
    - modifications to the design imply modifications to particular pieces of code

Reuse and Application

- late 80’s early 90’s - big thing was code reuse
- Identify reusable parts of code
  - ◊ generalize to make more reusable
  - ◊ factoring and decoupling

- Biggerstaff - not just code reuse, but also design recovery reuse
  - ◊ help build similar components
  - ◊ help recover similar components from other systems

Desire

- linguistic patterns - lexical
  - ◊ representation of informal information
  - ◊ naming convention
- Structural Requirements
  - ◊ presence of one component implies another
  - ◊ some structures are aggregations of other structures
- Incomplete Match
  - ◊ not all systems are created equal
  - ◊ manual intervention

Informal Information

```
#include <stdio.h>
#include “h0001.h”
#include “h0002.h”
#include “h0003.h”

f0001(a0001)
unsigned int a0001;
{
    unsigned int i0001;
    f0002(g0005, d0001, d0002);
    f0002(a0001, d0003, d0002);
    f0003(g0001[a0001].so001, g0001[a000l].sOOO2);
    go006 = a0001;
    i0001 = g0001[a0001].s0003;
    if( !f0004(i0001)&&(gOOO2->gOOO3)[iOOOl].sOOO4 == d0004)
        fOO05(iOOO1);
}
```

Informal Information

```
#include <stdio.h>
#include “proc.h”
#include “window.h”
#include “globdefs.h”

change-window(nw)
unsigned int nw;
{
    unsigned int pn;
    border-attribute(cwin,NORM_ATTR,INV_ATTR,INV-ATTR);
    border-attribute(nw,NORMHLIT-ATTR,INV-ATTR);
    move-cursor(wintbl[nw].crow,wintbl[nw].ccol);
    cwin = nw;
    pn = wintbl[nw].pnumb;
    if(!outrange(pn) && (g->proctbl)[pn].procstate == SUSPENDED)
        resume(pn);
}
```

Example Curses Screen (Debian)

Prototype

- lower level
  - ◊ functions, files, global data items
  - ◊ definition locations, use locations
  - ◊ calls uses depends
- Components
  - ◊ parser, analysis, view generation
  - ◊ links comments to artifacts
- Viewer
  - ◊ queries link back to source code

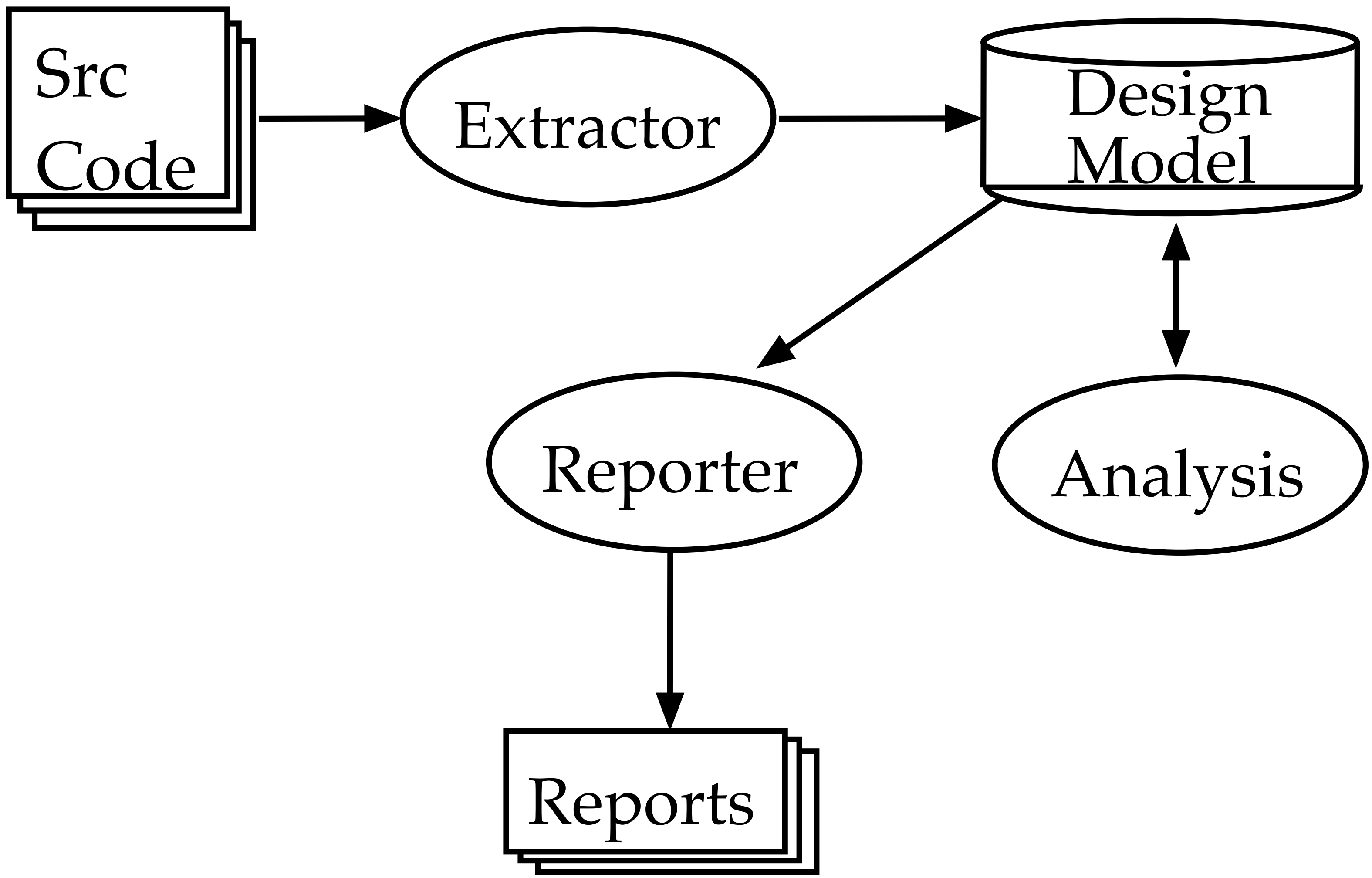
Analysis

- Prototype is lower level
  - ◊ starting point is the code
  - ◊ may also include comments
- Link Back to Code
  - ◊ always important
  - ◊ use to modify existing code
  - ◊ knowledge of design is important, but only useful if it helps you in the maintenance task
- Manual Intervention
  - ◊ Design recovery includes abstract concepts. Until real AI is created, human mind is still king.

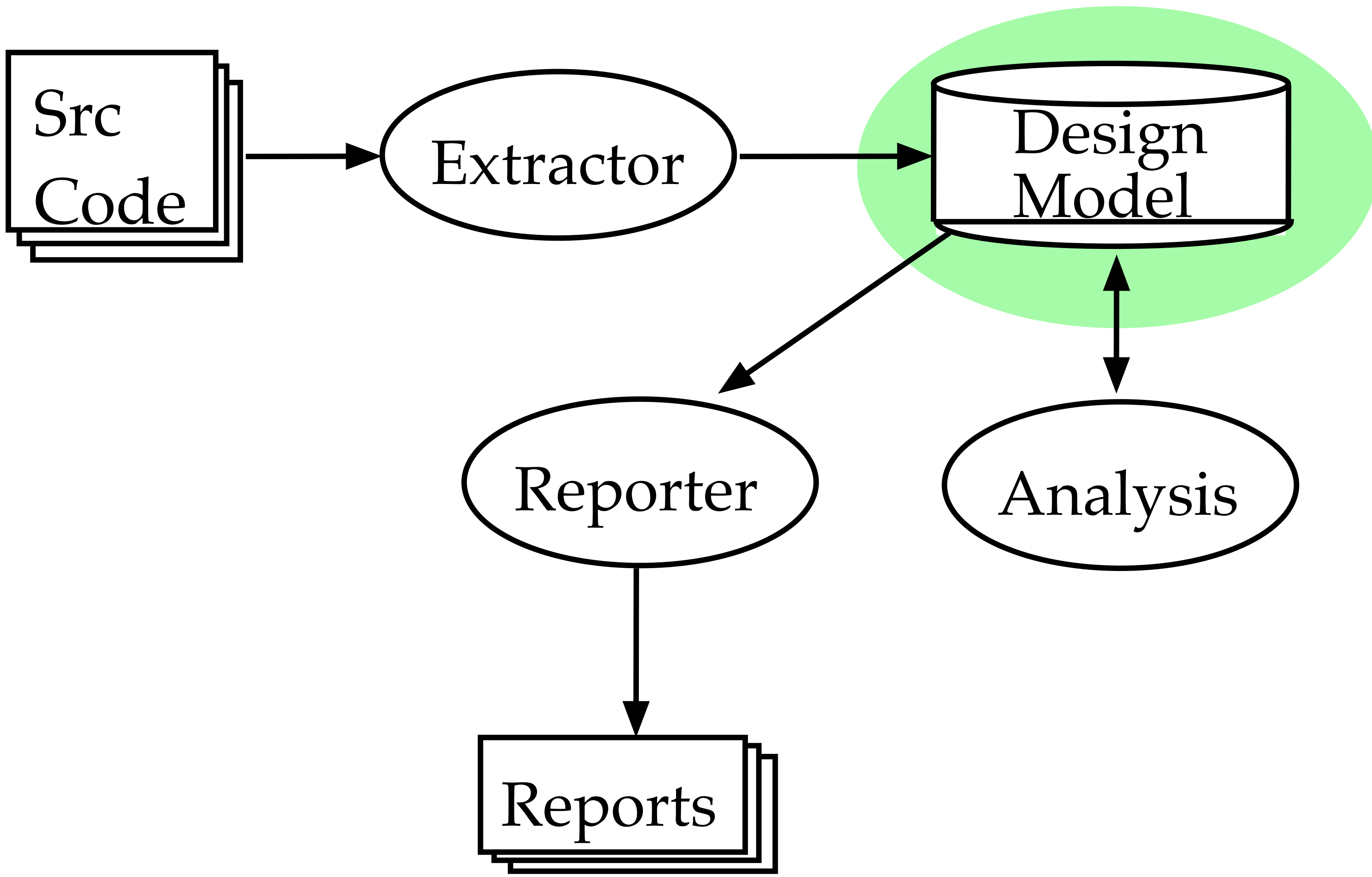
Analysis

- Informal Information
  - ◊ semantics is not the only thing
    - turing computable argument
  - ◊ real systems do *not* contain random code
    - they have to understand it and have some confidence that it actually works
  - ◊ naming conventions
  - ◊ structural conventions
- One main goal is to help humans
  - ◊ don't underestimate humans

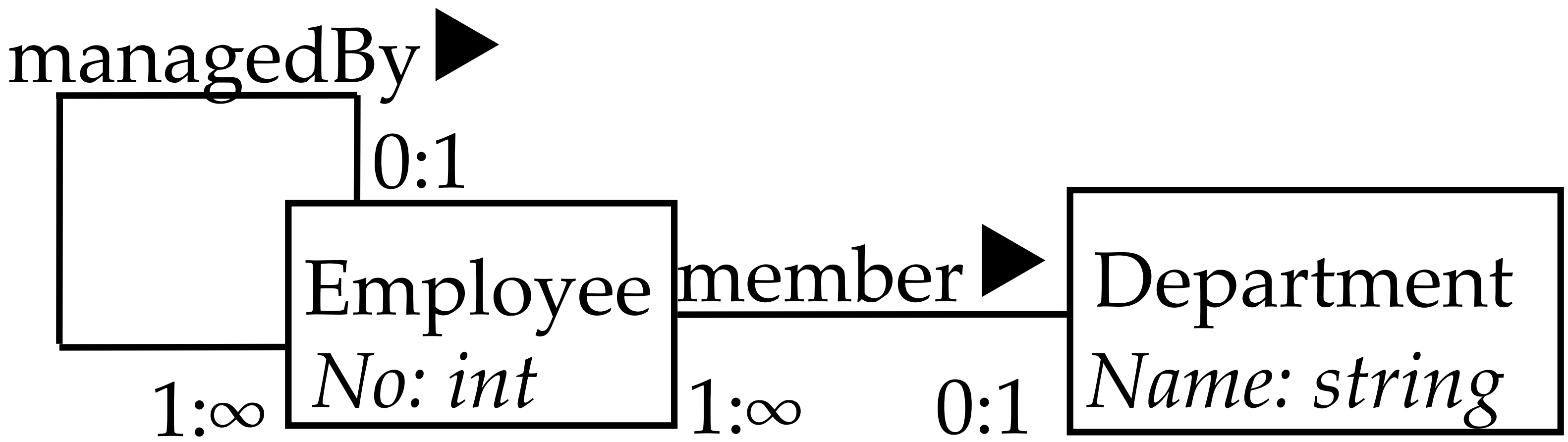
Design Recovery Architecture



Design Recovery Architecture

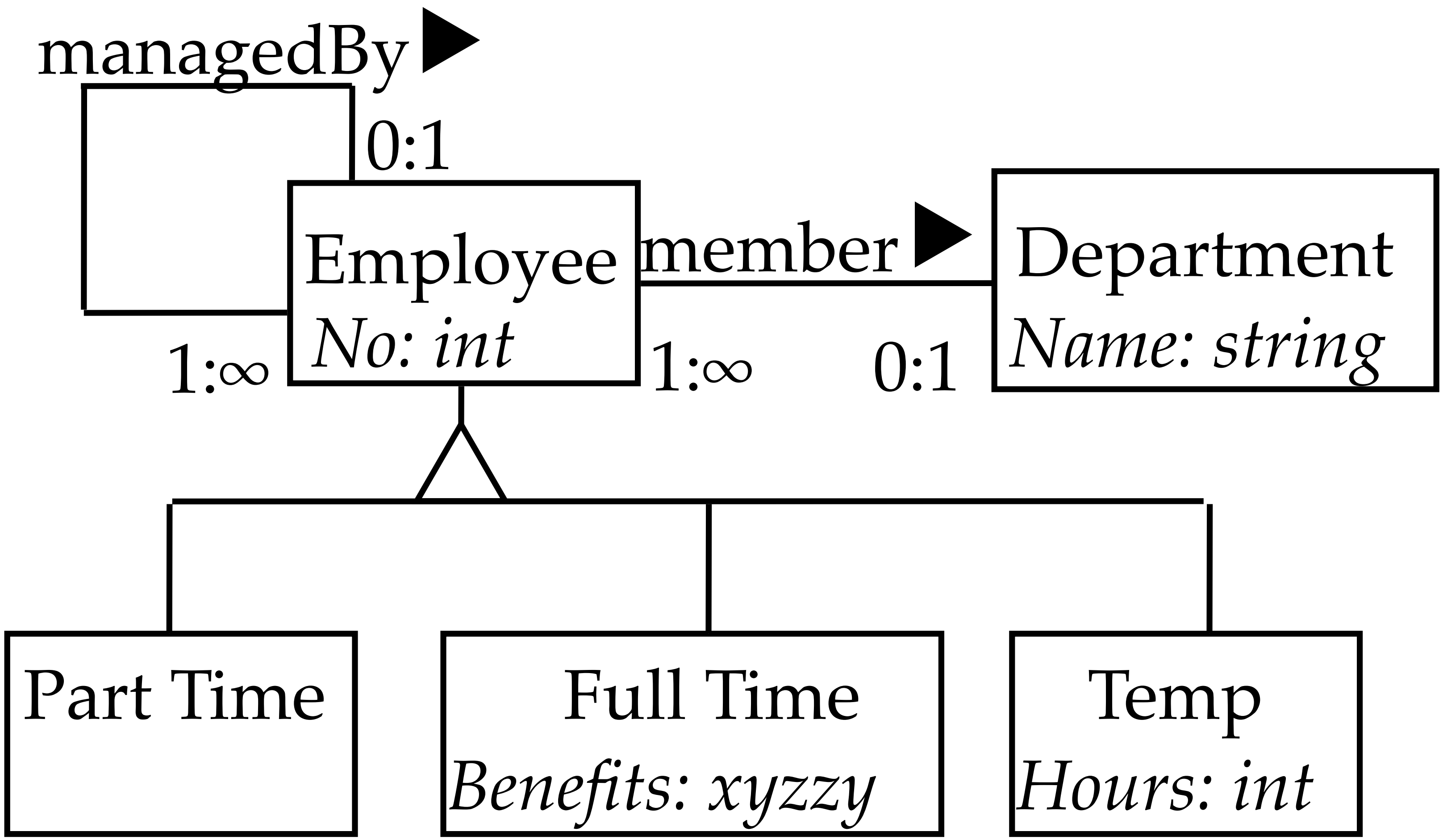


Modeling -ER



managedBy ▶
0:1
Employee
No: int
1:∞
member ▶
1:∞
0:1
Department
Name: string

Modeling - Extended ER



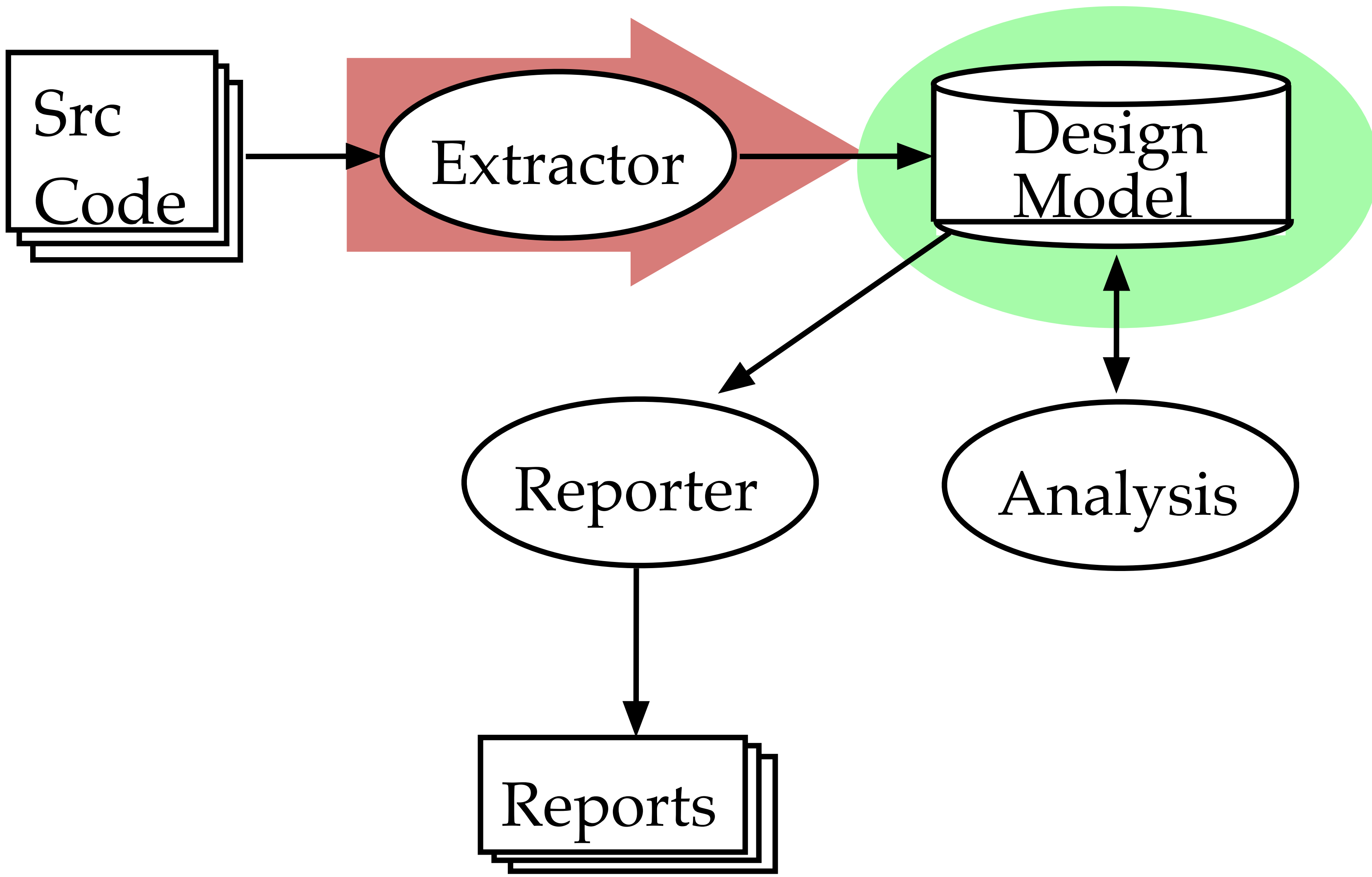
Modeling

- In traditional design (forward engineering), we model the problem domain and incorporate that model into the software in some manner.
  - ◊ OOAD
  - ◊ SA&D

- In design recovery, the problem domain is software. Our model will consist of entities that represent software artifacts (data is a program)

- Long Term Goal: to tie the model extracted from the code to a traditional problem model

Design Recovery Architecture



Base Model

- Entities and Relations in the Base Model directly represent software artifacts
  - ◊ source code elements

- Example Entities
  - ◊ variables
  - ◊ procedures
  - ◊ types
  - ◊ statements

Base Model

- Example Relations
  - ◊ calls (procedure calls a procedure)
  - ◊ references (procedure references a variable)
  - ◊ isFieldOf (field to structure or class)
  - ◊ hasType (type of variable or function)
  - ◊ ifPart (if statement $\Rightarrow$ statement)

Base Model - Notes

- some entities have natural names
  - ◊ variables
  - ◊ procedures
  - ◊ types

  - names may be predefined or user defined

- some entities do not have natural names
  - ◊ statements
  - ◊ blocks
  - ◊ constants

Base Model - Example

```
file main.c
  void printf(char *, …);
  char * foo(int);
  int main(int argc, char **argv){
    printf(“hello world%s”,foo(3)
  }

file foo.c
  char * foo(int x){
     return (“!\n”);
  }
```

Base Model - Example

Entities:

Files: main.c foo.c

Functions: foo, main

Variables: argc, argv, x

Prototypes: foo, printf

Constants: “hello world%s”, “!\n”

Types: void, int, char*, char**, char

Relations:

Contains: (main.c,printf), (main.c, main), (main.c foo)

Calls: (main,foo)

Parameter: (main, argc), (main, argv), (foo,x)

Argument: (foo,3)

HasType: (main, int), (foo, char*), (argc, int), (argv, char**), (x,int), (printf,void),(foo, char*)

Base Model - Issues

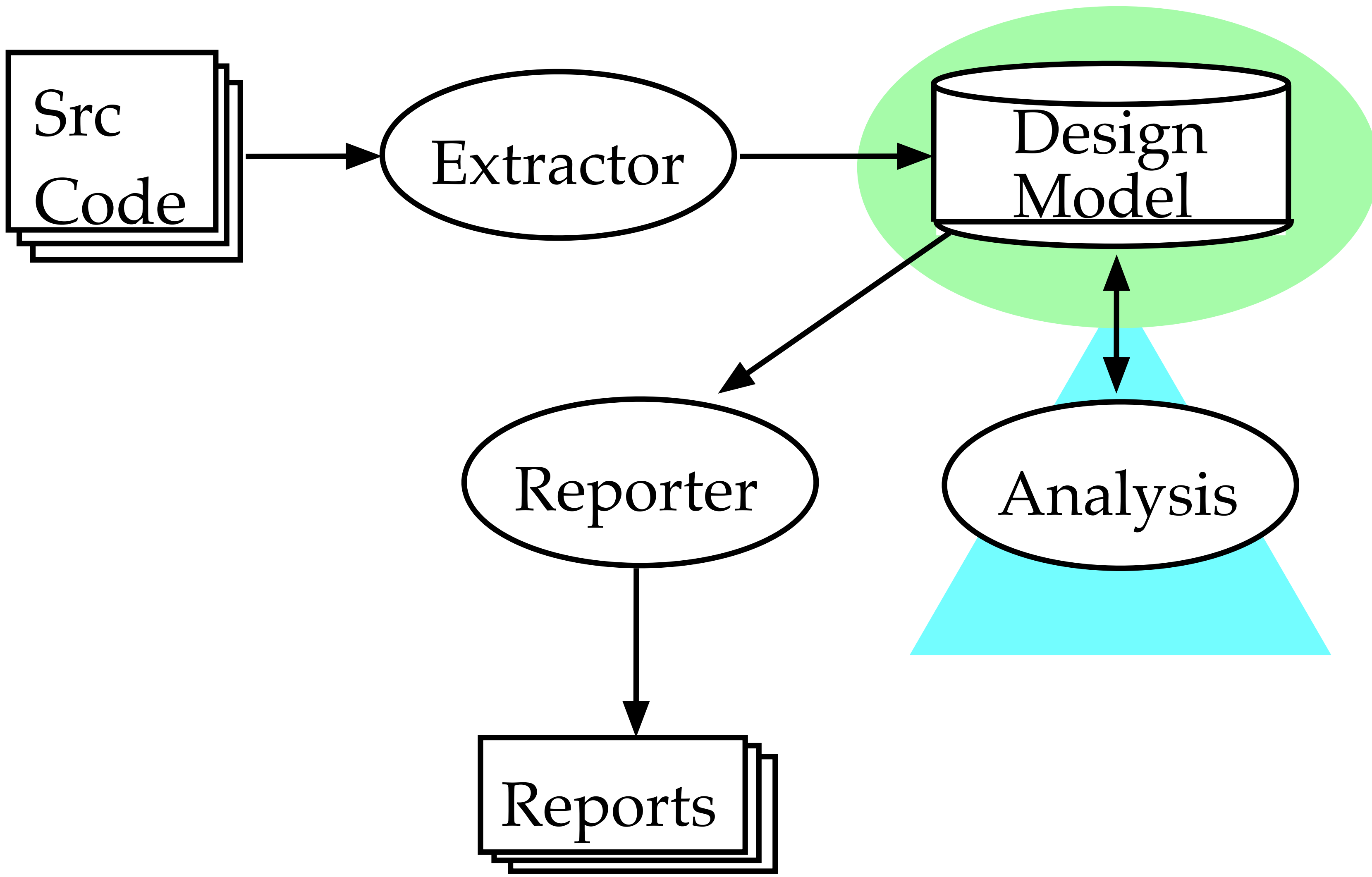
- Unique Naming
  - ◊ some entities have the same name
  - ◊ scoping
  - ◊ name spaces (Java, C, C++)
  - ◊ Model is a database, need a key for each entity
  - ◊ different entity sets - keys needed only for same entity sets and for entity sets that share relations
  - ◊ solutions:
    - unique id for each entity (CPPX, Columbus)
    - name derived from scope (LS/2000)

Base Model - Issues

- Resolution
  - ◊ sample model cannot connect arguments to parameters (more than one call? more than one argument?)
  - ◊ Return value of foo?

- Organization
  - ◊ Database practice - organize database to answer common queries
  - ◊ any given organization makes some queries hard, other queries easy

Design Recovery Architecture



Derived Model

- built on top of the base model
  - ◊ derived from information in the base model
  - ◊ new relations between entities
  - ◊ new entities for existing entity types
  - ◊ new entity types
  - ◊ new attributes

- Two types of derived information
  - ◊ deterministic computed information
    - implementation semantics, storage semantics
  - ◊ inferred information (heuristics)

Derived Model - Computed

- storage semantics
  - ◊ programmers can and do play storage games

```
struct xyzzy{
    int x;
    float y;
};
```

- x is at offset 0 and is 4 bytes long
- y is at offset 4 and is 4 bytes long

- Big Endian/Little Endian

Derived Model - Computed

- storage semantics
  - ◊ programmers can and do play storage games

```
union xyzzy{
    int x;
    float y;
};
```

- x is at offset 0 and is 4 bytes long
- y is at offset 0 and is 4 bytes long

- x and y occupy the same memory

Derived Model - Computed

```
struct xyzzy{
  int type;
  union {
    struct {
      …
      int x;
      …
    } option1;
    struct {
      …
      int y;
      …
    } option2;
  } detail;
};
```

what if fields X and Y have the same offset???

what if the programmer intends them to have the same offset??

Derived Model - Computed

- BCD - binary coded decimal
- COMP-3 - BCD + Sign Nibble

8 1	0 1	5 2	9 7	3 2

8 1	0 1	5 2	9 7	3 ±

Derived Model - Computed

- Cobol Data structures

```
01 A.
   05 B                  PIX XX.
   05 C.
      10 D               PIC X.
      10 FILLER          PIC X(3).
   05 F                  PIC 9(4).
   05 G REDEFINES F PIC XXXX.
```

Derived Model - Computed

- BCD - binary coded decimal
- COMP-3 - BCD + Sign Nibble

```
01 CONV-REC.
   05 NUM-VAL                 PIC 99 COMP-3.
   05 ALPHA REDEFINES NUM-VAL.
      10 ALPHA-VAL            PIC X.
      10 FILLER               PIC X

MOVE INBYTE to ALPHA-VAL.
DIVIDE NUM-VAL BY 10.
```

Base Model - Resolution Issue

Relations:

- Contains: (main.c,printf), (main.c, main), (main.c foo)
- Calls: (main,foo)
- Parameter: (main, argc,1), (main, argv,2), (foo,x,1)
- Argument: (foo,3,1)
- HasType: (main, int), (foo, char*), (argc, int), (argv, char**), (x,int), (printf,void),(foo, char*)

x = 3

Base Model - Resolution Issue

```
file main.c
  void printf(char *, …);
  void bar(int,int);
  int main(int argc, char **argv){
    foo(2,3);
    foo(atoi(argv[1]),atoi(argv[2]));
  }

file foo.c
  char * foo(int x, int y){
     …;
  }
```

Base Model - Resolution Issue

```
file main.c
  void printf(char *, …);
  void bar(int,int);
  int main(int argc, char **argv){
    foo(2,atoi(argv[2]));
    foo(atoi(argv[1]),3);
  }

file foo.c
  char * foo(int x, int y){
     …;
  }
```

Derived Model - Inferred

- Use other information to infer information about entities.
- Y2K - Dates
  - ◊ Names of Variables and Functions
  - ◊ Storage Types of Fields
  - ◊ Interaction with OS or with known API
  - ◊ Domain Dependent Patterns

```
01 MTGSTD                    PIC 9(6).
```

Derived Model - Inferred

- Use other information to infer information about entities.
- Y2K - Dates
  - ◊ Names of Variables and Functions
  - ◊ Storage Types of Fields
  - ◊ Interaction with OS or with known API
  - ◊ Domain Dependent Patterns

```
01 CURRENT-DATE-YYMMDD PIC 9(6).
01 MTGSTD              PIC 9(6).

IF MTGSTD > CURRENT-DATE-YYMMDD
```

Derived Model - Inferred

- Move to higher level of abstraction
- Business Rules, Business Types
- Goal:
  - ◊ Link to problem model for program
  - ◊ Employee Number, Customer Name, Customer Address
  - ◊ Where are they used?
  - ◊ How are they related?