Introduction to defect management

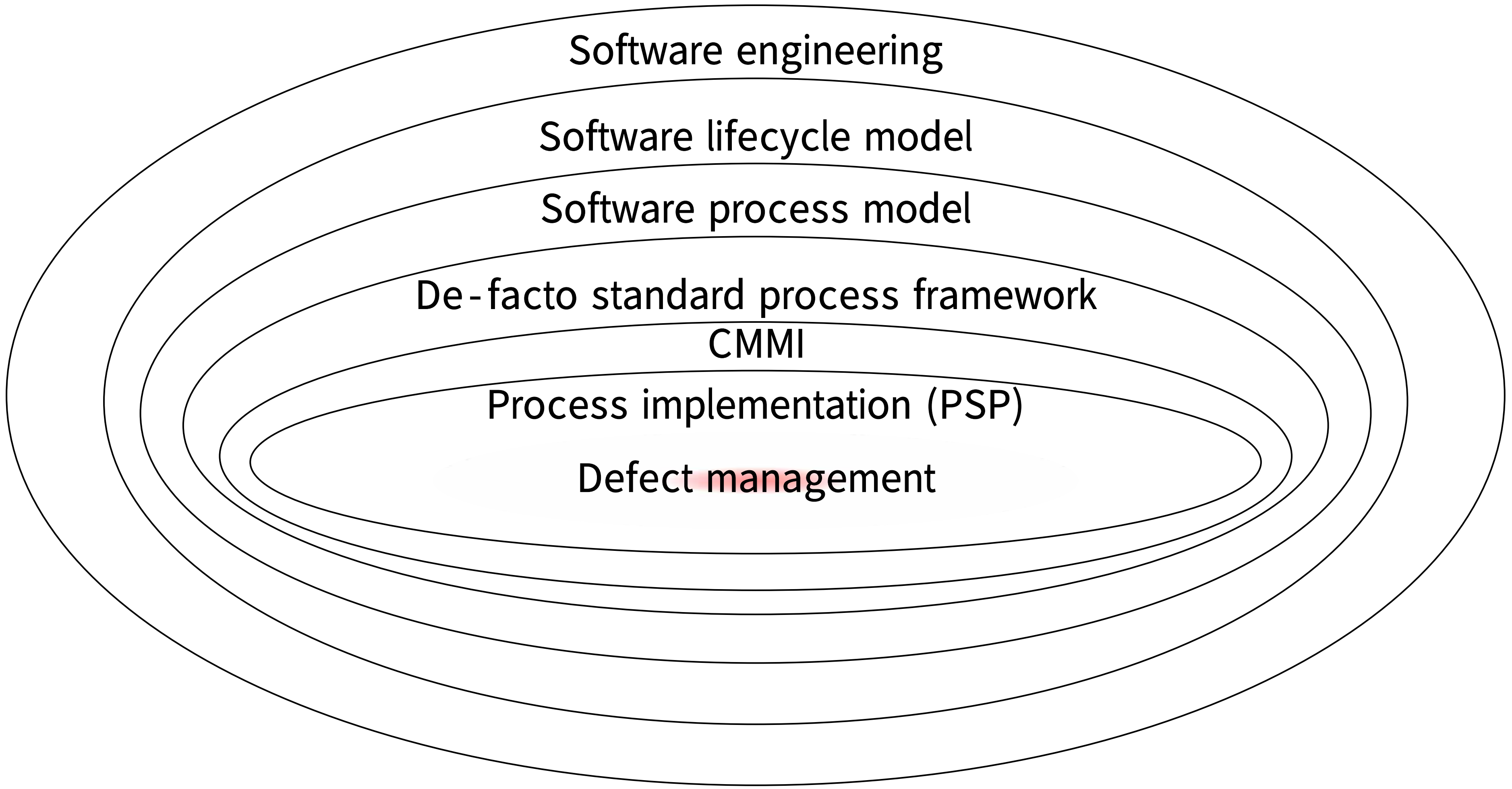
WRG

11 10th, 2005

Presented by J. Park

Table of Contents

Defect management position in this seminar



Part I : Defect Management

많은 소프트트웨어 엔지니어들은 프로그램을 컴파일하고 테스트할 때까지 결함에 대해 걱정할 필요가 없다고 생각하는 경향이 있습니다. 따라서 프로그램을 빨리 테스트하기 위해 설계와 코딩 단계를 서두르게 됩니다.

-Watts. S. Humphrey

Facts about Defect Management

- 대부분의 프로그래머가 흔히 범하는 오류가 있다
  - 편향적 오류(**biased error**)라고 불림
  - **Index**의 밀림(**round-off error**), 정의/참조의 불일치, 설계항목 누락, 변수의 초기화 생략, 조건 누락 등
- 오류는 뭉치는 경향이 있다
  - 오류의 반이 모듈의 15 %에서 발견된다(1975)
  - 오류의 80%가 단지 모듈의 20% 이내에서 발견된다(1995)
  - 대략 80% 의 결함이 모듈의 20%에서 나오고, 모듈의 절반 정도는 오류가 없다(2001)
- 소프트웨어 오류 제거에 단 하나의 최상의 방법은 없다
  - 코드 검사(**review**), 검토(**inspection**), 테스트 등
- 오류는 항상 남아 있다. 심각한 오류를 제거하거나, 최소화하는 것이 목표가 되어야 한다
  - 개인적 수련으로 오류 발생 비율을 75%까지 줄일 수 있다(2001)
  - 모든 버그를 찾아내지는 못할 것이다(2002)
  - 시스템 중단의 90%는 많아봐야 10%의 결함 때문이다(2001)

Facts about Defect Management (Cont'd)

- 오류 제거는 생명 주기(**life cycle**)에서 가장 시간이 많이 소모되는 단계다
  - **20-20-20-40**
  - 오류 제거는 코딩과 유지 보수 사이에 끼어 있는 **일회성 이벤트**가 아니라 소프트웨어 개발 **전 단계에 걸쳐서 반복적으로 수행**되어야 한다.
- 완전히 테스트했다고 믿는 소프트웨어도 보통은 **logic path**의 **55~60%** 정도만 테스트가 된 경우가 많다
  - 실행되지 않은 logic path의 검증을 위해서 **code review** 와 **inspection** 을 사용하는 것이 권장된다.
- 엄격한 검사(**review**)는 첫번째 테스트 케이스를 실행시키기도 전에 소프트웨어 제품에 포함된 오류의 **90%** 까지 제거할 수 있다
  - **동일한 오류를 찾는데 드는 비용 역시 테스트 비용보다 훨씬 적음**
  - 오류 제거는 코딩과 유지 보수 사이에 끼어 있는 **일회성 이벤트**가 아니라 소프트웨어 개발 **전 단계에 걸쳐서 반복적으로 수행**되어야 한다.
- 엄격한 검사(**review**)가 좋기는 하지만, 테스팅을 대체할 수도 없고 대체해서도 안된다
  - Logic path를 100% 모두 검사한다고 해도 테스트가 다 된것은 아니다
- 고객 만족도 측정과 프로세스 개선 측면에서 봤을 때 출시 후 검사(**review**)의 중요성은 인정되지만, 이를 시행하는 조직은 거의 없다
  - 프로젝트가 끝난 후 1~3 주후에 검사(retrospective)를 하기를 권고함

Fallacies about Defect Management

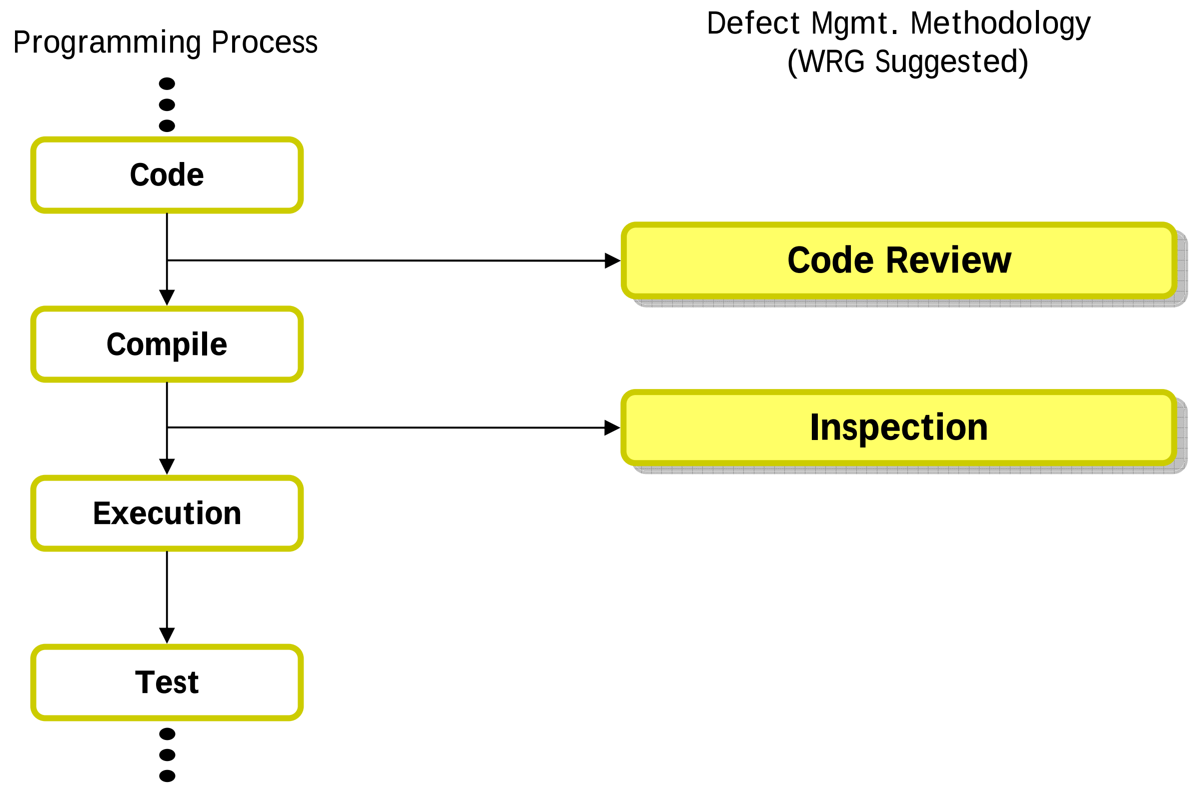
- ❖ 보는 눈이 많으면, 모든 버그는 그 깊이가 얕다
  - ➢ 어떤 오류의 깊고 얕음은 그것을 찾고자 하는 사람의 수와는 관계가 없다
  - ➢ 검사(inspection)에 대한 연구에 따르면, 검사의 참가자가 늘어날 수록 발견되는 버그의 수는 급격히 줄어든다
- ❖ 랜덤 테스트 입력은 테스트를 최적화하는 좋은 방법이다
  - ➢ 예외 처리 코드를 실행시킬지에 대한 믿음이 부족하다
  - ➢ 테스터의 지혜와 직관을 활용하지 못한다
  - ➢ 소프트웨어의 규모가 커질수록, 랜덤 테스트는 그 효율이 줄어든다(2001)
- ❖ 프로그래밍을 가르칠 때, 프로그램을 어떻게 작성하는지 보여주며 가르친다
  - ➢ 외국어를 배우는 것과 동일하다

<Facts and Fallicies of Software Engineering>, Robert L. Glass

그럼 왜 Review 는 주목받지 못하는가

- ❖ Review 로 돈을 버는 major vendor 는 거의 없다
- ❖ Review 에는 새로운 것도 없고 따라서 시장성도 없다
  - ➢ 근본적으로 20~30년 전의 것과 달라진 것이 많지 않다
- ❖ Review 는 소프트웨어 개발 단계의 뒤쪽 단계에 있는 보이지 않는 부분으로 간주된다
- ❖ Review 는 효율적이긴 하지만, 녹초가 될 정도로 정신을 집중해야 하는 고된 작업이다

Defects Management



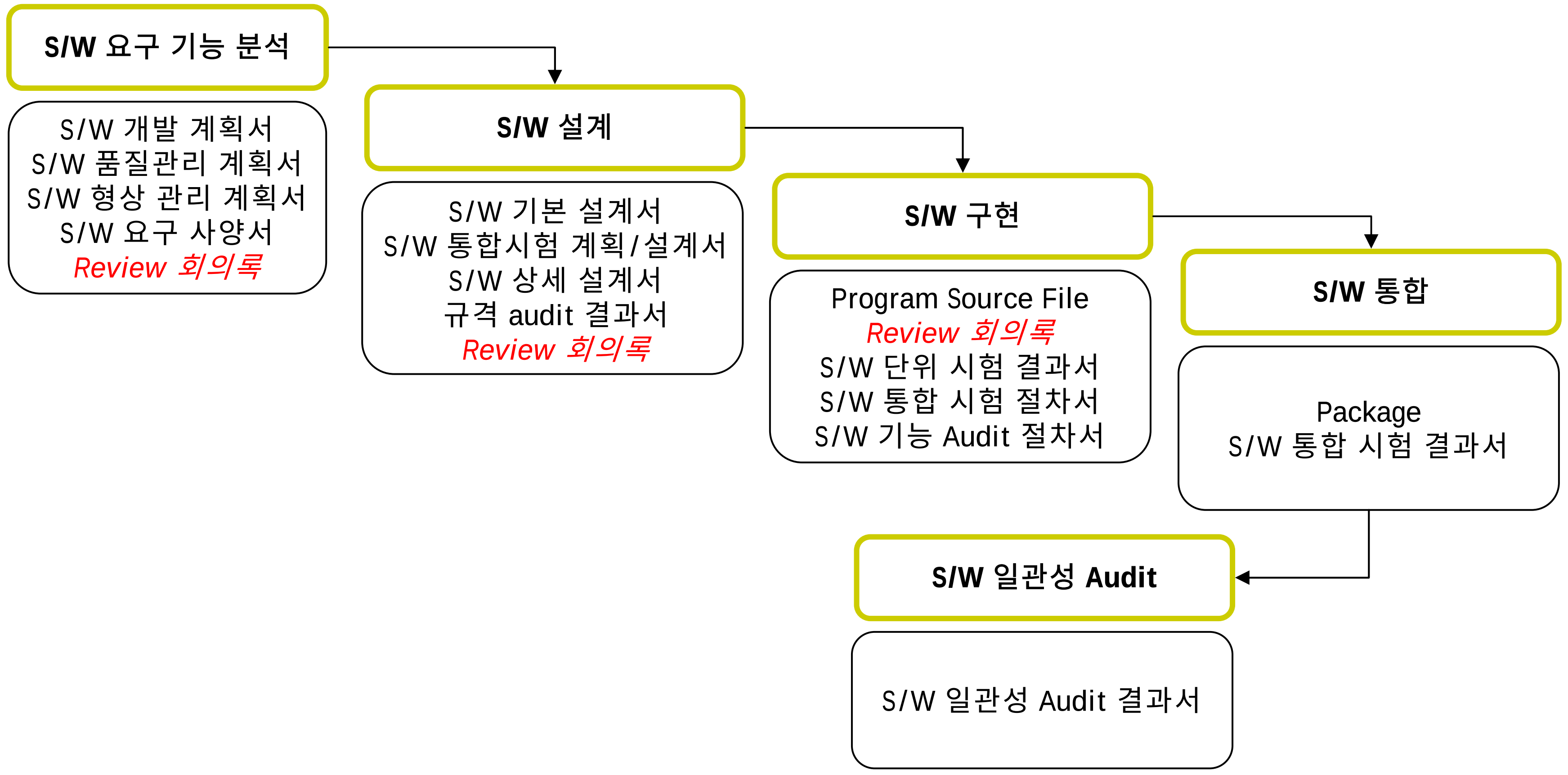
- Inspection은 Code Review보다 Systematic하며 조직적으로 일어나는 QA활동으로서, 6개의 명확한 활동(Planning, Overview, Preparation, Meeting, Rework and Follow-up)으로 구성
- Inspection은 Code Review에 비해서 그 무게가 무거우며, 조직적인 문화, 제도, 프로세스의 뒷받침이 필요한 활동임

Part II : Code Review

프로그램 작성은 경이로운 분야입니다. 이것은 굉장한 퍼즐을 매일 창조하고, 이 퍼즐들이 여러분들이 의도한 대로 실제 작동되는 것을 보는 것과 같습니다….그러나 불행하게도 대부분의 조직에서 프로그램 작성은 전혀 즐거운 일이 아닙니다…왜냐하면 초급 프로그래머들이 보람되고 전문적인 방법으로 프로그램을 작성하는 방법을 배우지 못했기 때문입니다.

-<Personal Software Process> 서문中

S 전자의 S/W 개발 프로세스



❖ 여기서 언급되는 **review**는 모두 **team activity**로서의 **review** 임

❖따라서, **personal review** 에 대한 **guide** 제시가 필요함

Review Methods

- Inspection
  - S/W의 team review를 위한 구조적 절차
  - 정형적 구조 및 참여자들을 위한 명확한 역할이 정의되어 있음
  - 1976년, Michael Fagan에 의해 소개됨
  - Part III 에서 자세하게 논의함
- Walk-through
  - 발표회 형식을 갖는 덜 정형화(less formal)된 검토 회의
  - 각 SW 개발자는 해당 SW가 어떻게 실행되는지를 코드 내부를 이동하며(walk-through) 설명
  - 참여자는 이슈를 제기하고 질문을 함
  - 사용자와 개발자가 함께 참여할 경우, *생략된 기능* 및 *잘못 이해되고 있는 부분*을 파악하는데 더 효율적인 효과를 가져옴
  - Inspection의 준비(preparation) 단계를 위해 사용됨
- *Personal Reviews*
  - 개발자 *스스로, 자신의 산출물*을 검토하는 것
  - Workstation을 공유해야 하는 시절에, compile 시간을 줄이기 위한 노력으로 많은 기법들이 연구됨

이런 다양한 review method 들은 **코드 이외의 산출물**(개발 계획서, 요구 사항 기술서, 시험 계획 및 test case 등)에도 사용되어 질 수 있으며, 모든 산출물들은 **1회 이상**의 review를 반드시 거쳐야 한다

Why Review Programs

"프로그램을 review하는 것은 초벌 원고를 review하는 것과 같다"

- Review를 수행하는 가장 중요한 이유는 그 경제성에 있다

	Relative code review time	Relative Unit Test Fix Time	Relative Post Unit Test Fix Time
38 Pascal Programs	**1**	8	16
25 C++ Programs	**1**	12	60

- Fix time은 defect type 과 무관하다
- PSP 연구 보고서에는, 일반적으로 review 단계에서 해당 fix를 발견하여 고치는 비용(시간)이 testing 단계보다 **3배 ~ 5배** 가량 효율적이라고 보고하였다

Why Review Programs (Cont'd)

- **Review** 단계에서는 defect를 **직접** 접근한다
  - 코드 개발자는 본인이 작성한 Logic이 생성해내야 하는 결과에 대해서 알고 있다
  - 코드를 읽어나가다 보면, 전체 프로그램의 실행 모습(behavior)을 정신적으로 인식하게 된다 (construct for mental-context)
  - 코드 작성자가 코드 상에서 이해 안가는 부분을 만나게 되면, 반드시 개발자가 원하는 모습대로 움직이지 않는다

"Review는 프로그램의 동작에 대한 원리와 이유에 대한 전체적 그림을 인식하게 해준다"

- **Testing** 단계에서는 징후(symptoms)를 보고 defect를 찾아가는 과정(**debugging**)을 거친다
  - Testing은 기대하지 않았던 프로그램의 동작 행위를 기반으로 시작된다
  - 연구 결과에 의하면, 한 OS S/W의 경우, 3명의 숙련된 엔지니어들이, 고객이 클레임을 건 하나의 defect 를 3개월 동안 추적하여 찾아내었다. 똑같은 defect 를 고의로 코드에 주입하여 5명의 엔지니어로 구성된 다른 팀에게 주었더니. 2시간 만에 해당 defect 를 발견하고, 그 외에 다른 71개의 defect도 지적하였다 (by review)

" 만약 당신이 debugging에 능숙하다면, 당신은 이미 해당 S/W의 Logic, Parameter 값들에 익숙하다는 것이다. 그렇다면, 그전에 왜 미리 S/W를 검토하는 것이 이상한가?"

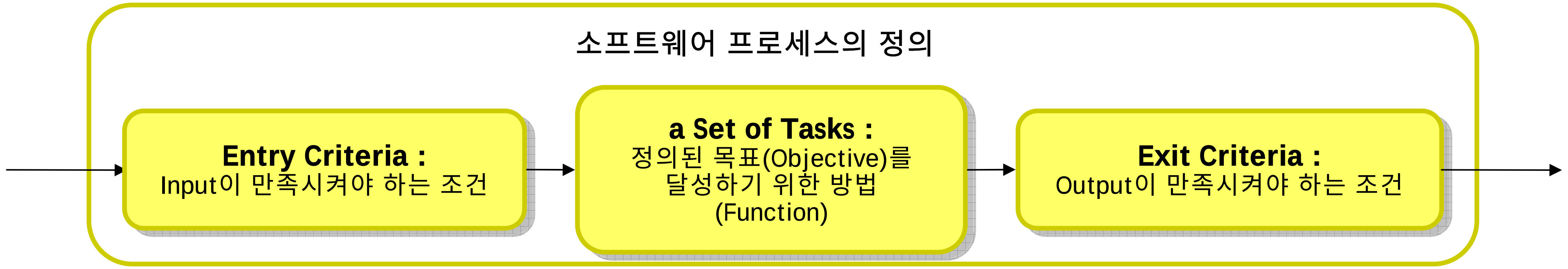
Personal Reviews : Review First, Then Compile

- "If you want a **quality product**, spend the time to **personally** engineer it, **review it**, and rework it until you are **satisfied with its quality**"
  - 그 후에 compile과 testing을 수행함
- 일반적으로 엔지니어들은 프로그램 작성 후에, 즉시 compile을 하려고 시도함
  - Compiler는 엔지니어를 계속 새로운 문제들로 이끌어감
    - Complaints about "How slow the compilers are !"
  - 또한 compiler 자체는 완벽하지 않음
    - C++의 경우, syntax 상에서 발견되지 못한 defect들이 있음(약 9.3 %)
- 일단 프로그램이 compile이 되고 나면, 엔지니어는 코드에 대한 집중도가 떨어진다
  - 프로그램이 무사히 실행되었으면 하는 기대 심리만 올라감
- Compile time defect의 개수와 testing time defect의 개수는 일반적으로 **강한 상관 관계**가 있다고 알려져 있음
  - Compile time때 defect의 개수가 적으면, testing time때의 defect의 개수도 적다
- 따라서 review는 compile 전에 수행하는 것이 매우 중요하며, 반드시 필요하다
  - Compiler는 스스로의 review의 품질을 검증해주는 도구로 사용하자 !!!

Review Principles

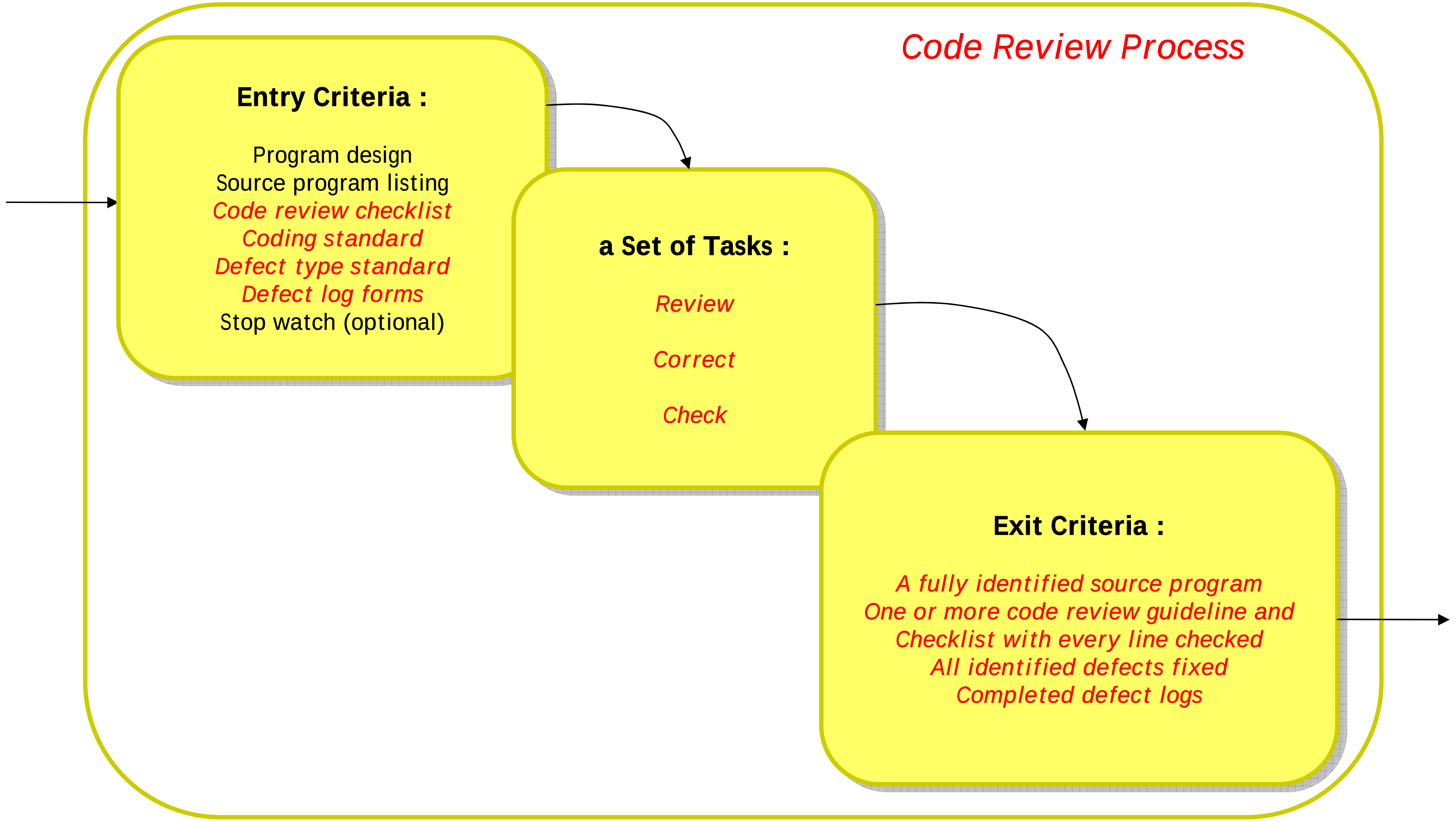
- Review의 목표를 설정한다
  - Review 교육 훈련 보고서에 의하면, 첫번째 compile전에 80% 이상의 defect를 발견할 수 있었음
  - 결함 기록 로그를 참고하여 스스로에게 가장 적절한 목표를 설정하고 지속적으로 향상시켜 나간다
  - 목표는 review measure중에서 review 양품률(review yield)를 이용하여 설정한다
    - 예) 2005년 1월의 W사의 정모 연구원의 목표 review yield는 60% 이다

- 정의된 review 프로세스를 준수한다
  - Review 프로세스도 일반적인 프로세스와 같이, entry criteria, tasks, exit criteria로 구성된다
  - Review를 위한 준비물로는 review할 소스 코드 이외에도, 설계 문서, 코딩 표준화 문서, 결함 표준화 문서, 결함 기록 로그 등이 있다

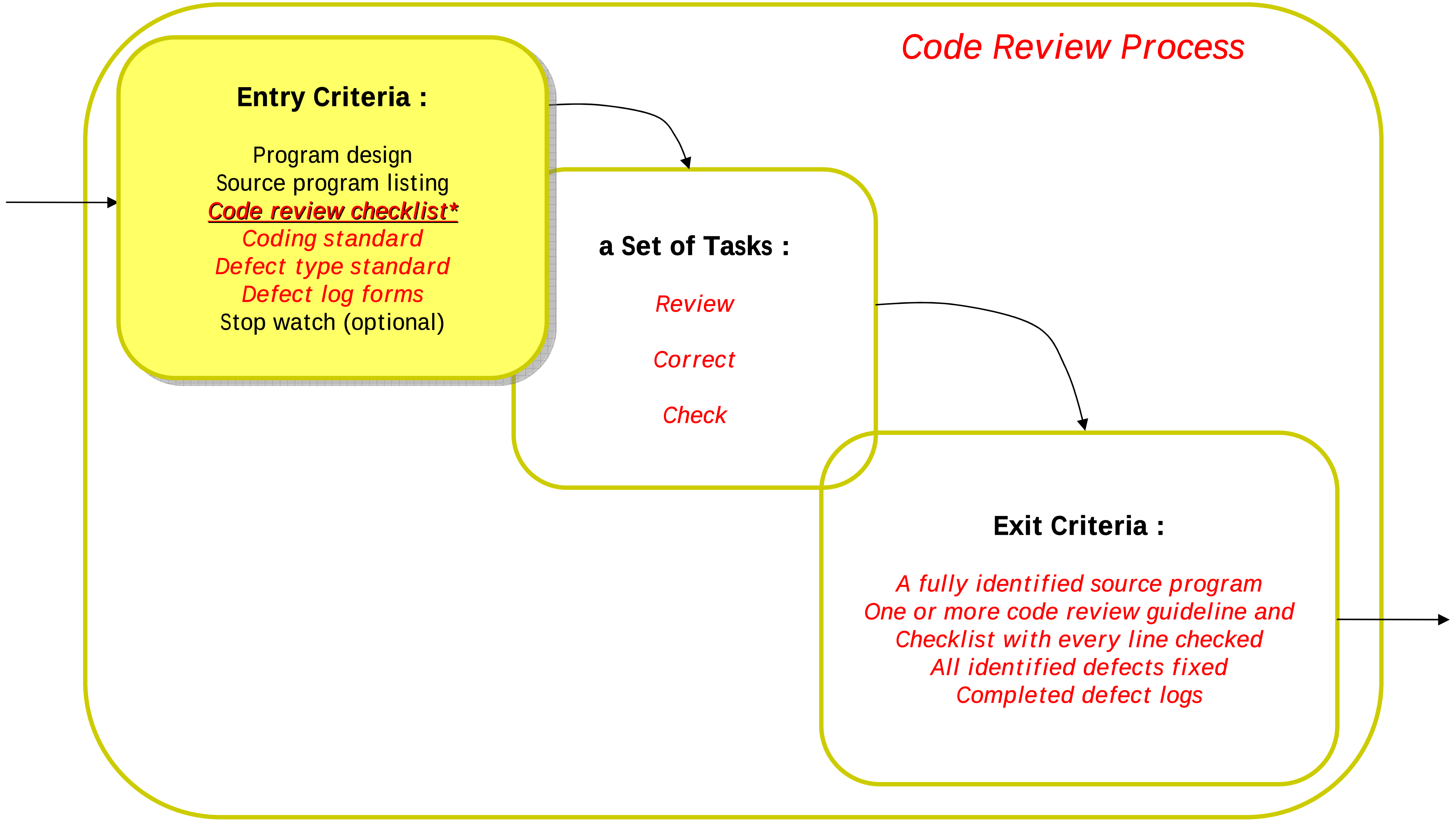

소프트웨어 프로세스의 정의
Entry Criteria : Input이 만족시켜야 하는 조건
a Set of Tasks : 정의된 목표(Objective)를 달성하기 위한 방법 (Function)
Exit Criteria : Output이 만족시켜야 하는 조건

- Review 결과를 측정하여, review 프로세스를 개선시킨다
  - "Learn from the facts. Let the data talk, and use your best judgment"
  - Review measure들을 이용한다

Code Review Process



Code Review Process



Entry Criteria : Checklist

- C++ 코드 리뷰를 위한 checklist (1/2)

목적	효과적인 코드검토 수행	#	#	#	#	누적	누적 %
개요	▪ 각각의 검토 단계를 마치면 오른쪽 항목에 발견된 결함의 수를 기재하고 만약 결함이 없으면 오른쪽 항목에 체크 표시 ▪ 다음 프로그램을 검토하기 전에 해당 프로그램, 클래스, 객체 또는 Methods에 관한 Checklist 작성을 끝낼 것						
완료	▪ 설계된 모든 기능이 코딩 되었는지 점검						
Includes	▪ Includes 문에 결함이 없는지 검증						
초기화	▪ 변수와 인수의 초기화를 점검 ➢프로그램 초기화 단계에서 ➢각 루프의 시작 지점에서 ➢함수/procedure 입력 단계에서						
호출	▪ 함수 호출 형태 점검 **(Pointer, 인수들, '&'의 사용)**						
명칭	▪ 명칭의 철자를 체크하여 사용 ➢명칭이 일관성이 있는가? ➢명칭이 관련 범위 내에 있는가? ➢모든 구조/Class가 '.' 참조를 사용하는가?						
문자열	▪ 모든 문자열 점검 ➢Pointer에 의해 인식되는가? ➢Null 값으로 끝나는가?						

Entry Criteria : Checklist (Cont'd)

- C++ 코드 리뷰를 위한 checklist (2/2)

목적	효과적인 코드검토 수행	#	#	#	#	누적	누적 %
Pointer	▪**모든 Pointer**를 점검 ➢**NULL** 값으로 초기화 되었는가? ➢삭제하기 전에 생성되어 있는가? ➢신규생성 전에 사용되던 부분은 삭제하고 있는가?						
출력 Format	▪출력 Format을 점검 ➢각 출력 행의 진행이 적절한가? ➢각각의 간격이 적절한가?						
{ } 쌍	▪{ }를 적절하게 사용했는지 확인						
논리 연산자	▪**==, =, ||** 등의 사용이 적절한지 확인 ▪각 함수에 **( )**가 적절히 되어 있는지 확인						
행 단위 체크	▪코드의 모든 행을 점검 (명령구문, 적절한 구두점)						
표준	▪코드가 코딩 표준을 준수하고 있는지 확인						
파일 열기/닫기	▪모드파일이 다음과 같은지 점검 ➢적절하게 선언 되었는가? ➢열렸는가? 닫혔는가?						
전반적인 체크	▪시스템 이상 여부와 예기치 못한 문제들을 체크하기 위해 프로그램을 전반적으로 검토						
합계							

Checklist 사용 tip

- Checklist를 적용하는 소스코드 파일, 각 함수 별로 checklist 의 상단에 review한 파일, 함수 이름을 적고 각각 별도로 기록한다
- 두 가지 이상의 topic을 동시에 검토하는 것에 어려움을 느낄 때는 **checklist를 section별로 분리하라**
  - ➢ 각 섹션 별로 code review를 하는 것이 더 효과적이다
    - 예) condition 내의 조건들만 보는 checklist를 별도로 제작한다
  - ➢ 각 섹션은 비슷한 특성을 갖는 검토 사항들로 그룹화한다

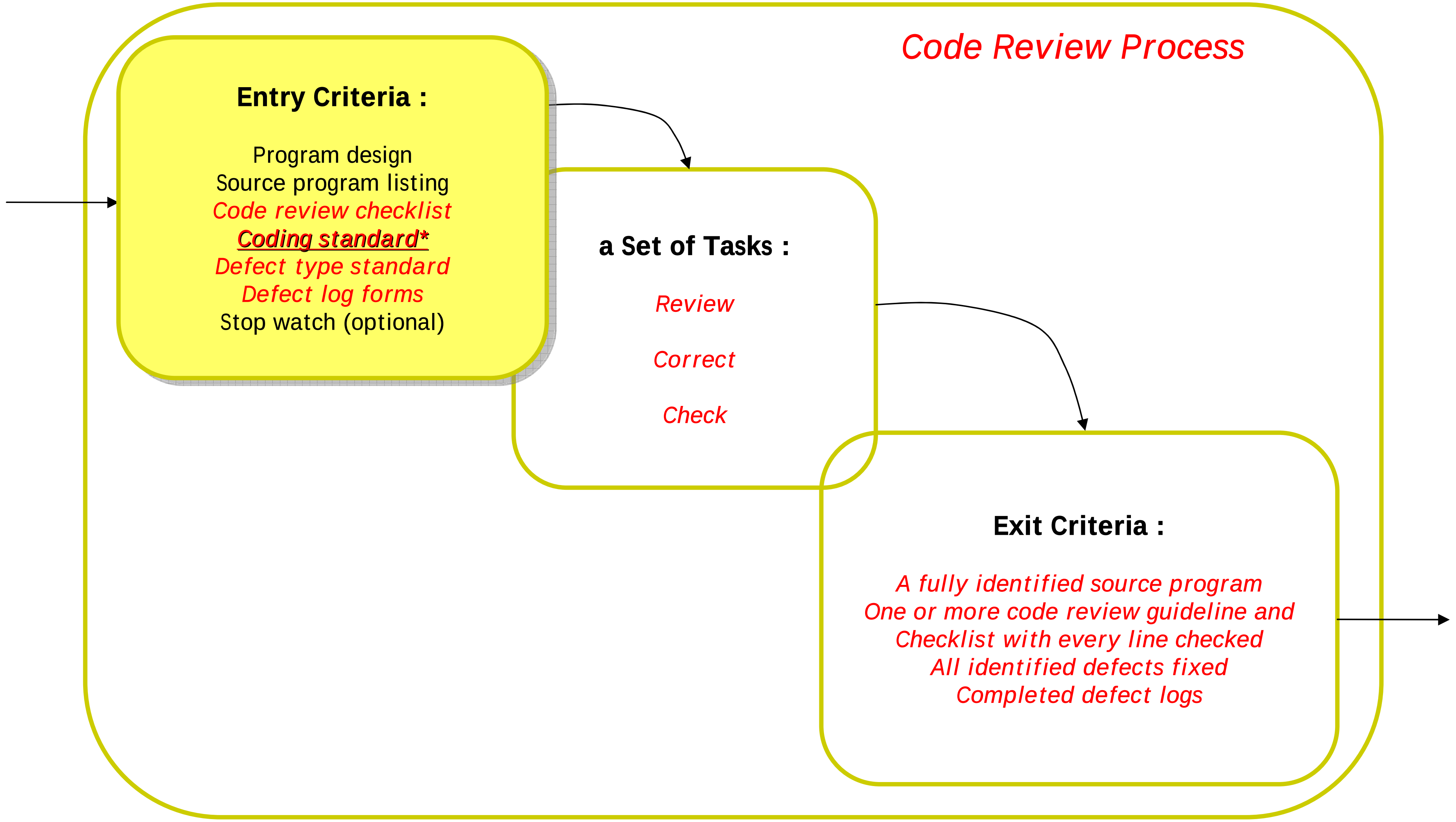
- 메인 함수 내에 여러 개의 함수 호출이 있을 경우, bottom-up 형식으로 review를 진행
  - ➢ 가장 함수 호출을 적게 하는 함수부터 review를 시작한다
  - ➢ Cf. 설계 검토는 top-down 식으로 진행한다

Checklist 만들기

- 스스로의 경험은 가장 좋은 checklist가 된다
  - ➢ 각 개발자 별로 실수하는 유형과 특성이 있다
    - W. Humphrey의 경우, "syntax, interface, function과 assignment"에서 97%의 defect를 발견
  - ➢ 따라서 review하면서 결함 기록 log를 만드는 것은, 품질 좋은 checklist의 작성을 위한 중요한 기록이 된다
  - ➢ 최초의 review시에는 PSP에서 제공하는 결함 유형 표준(defect type standard)를 기초로 시작할 수 있다
  - ➢ 예) WRG Z130 팀의 Checklist 의 예(일부)

- **Pareto distribution** 을 작성해본다
  - ➢ 개발자 스스로 발견한 defect들을 type별로 sorting한 분포를 말한다
  - ➢ 많이 범하는 defect의 경우, 1개라도 발견하면 Code 내에 더 있을 것이라고 믿고, 더욱 더 유심하게 살펴보아야 한다

Code Review Process



Coding Standard

- Checklist는 프로그램을 검토하기 위한 하나의 표준을 제공할 수 있음
- 표준(standard)이란 공식적 비교를 위한 기준
    - Coding standard란 작업할 때, 하나의 모델로서 도움이 될 수 있도록 승인된 코딩 관행을 의미
- Defect 발생을 억제하는데 도움을 준다
- Code의 가독성(readability)를 향상시켜서, 향후 유지 보수 시에도 큰 도움을 준다
- 조직의 특성, 프로젝트의 특성 등에 의하여 개별적으로 정의되는 경향이 많다
    - GNU Coding Standard
    - Atacama Milimeter Array C Coding Standard
    - SUN Java Coding Standard
    - C++ Coding Standard

Coding Standard (Code Standard C++)

Purpose	C++ Program을 위한 지침
Program Headers	모든 프로그램을 시작할 때 부연설명이 잘된 Header를 작성
Header 양식	/***/ /* Program Assignment: the program 번호 */ /* Name: 개발자 명 */ /* Date: 개발 시작일 */ /* Description: 개발 내용(Program)과 Function에 대한 */ /* 간략한 설명 */ /***/
Listing Contents	Listing contents 에 대한 개요 제공
Contents Example	/***/ /* Listing Contents: */ /* Reuse instructions */ /* Modification instructions */ /* Compilation instructions */ /* Includes */ /* Class declarations: */ /* CData */ /* ASet */ /* Source code in c:\classes\CData.cpp */ /* CData */ /* CData( ) */ /* Empty( ) */ /***/
Reuse instructions	● Program의 활용에 대한 설명 및 declaration format, parameter 값과 type, limit등에 대한 정보 제공 ● Illegal 값에 대한 경고, overflow condition 및 오작동을 유발할 수 있는 각종 condition에 대한 경고 제공

Coding Standard (Code Standard C++ cont'd)

Reuse Example	/**/ /*Reuse Instructions: */ /* int Printline (char *line_of_character) */ /* Purpose: string을 프린트 함, 'line_of_character', on one print */ /* line */ /* Limitations: line 길이의 최대값은 LINE_LENGTH 값으로 제한 */ /* Return: 프린터가 준비되지 않았을 경우 0 , else 1 */ /**/
Identifiers	모든 변수와 함수명, constants 및 identifier의 목적에 맞는 직관적인 명칭 사용. 일반적이지 않는 줄임말 사용을 피하고 한 철자의(single-letter) 변수 사용 금지
Identifier Example	Int number_of_students; /* 적절한 사용의 예 */ Float x4, j, ftave; /* 적절치 못한 사용의 예 */
Comments	● 코드의 동작 내용을 쉽게 알 수 있게 코드의 내용을 기록 함 ● 기록되는 Comment는 코드의 목적과 동작에 대한 설명을 담고 있어야 함 ● 변수 선언 시 그에 대한 이유를 설명해야 함
Good Comment	if(record_count > limit) /* 모든 record 가 빠짐없이 실행 (count)되었습니까? */
Bad Comment	if(record_count > limit) /* record_count 가 limit의 값보다 */ /* 큰지 확인 */
Major Sections	큰 블록 단위의 프로그램 section은 그 section의 프로세싱에 대한 단위 설명 (Block Comments)을 시작으로 함

Coding Standard (Code Standard C++ cont'd)

Example	`/*******************************************************************************/`
	`/* 본 section은 'grade' array의 내용을 검증하고 */`
	`/* 학급의 평균값을 산출 함 */`
	`/*******************************************************************************/`

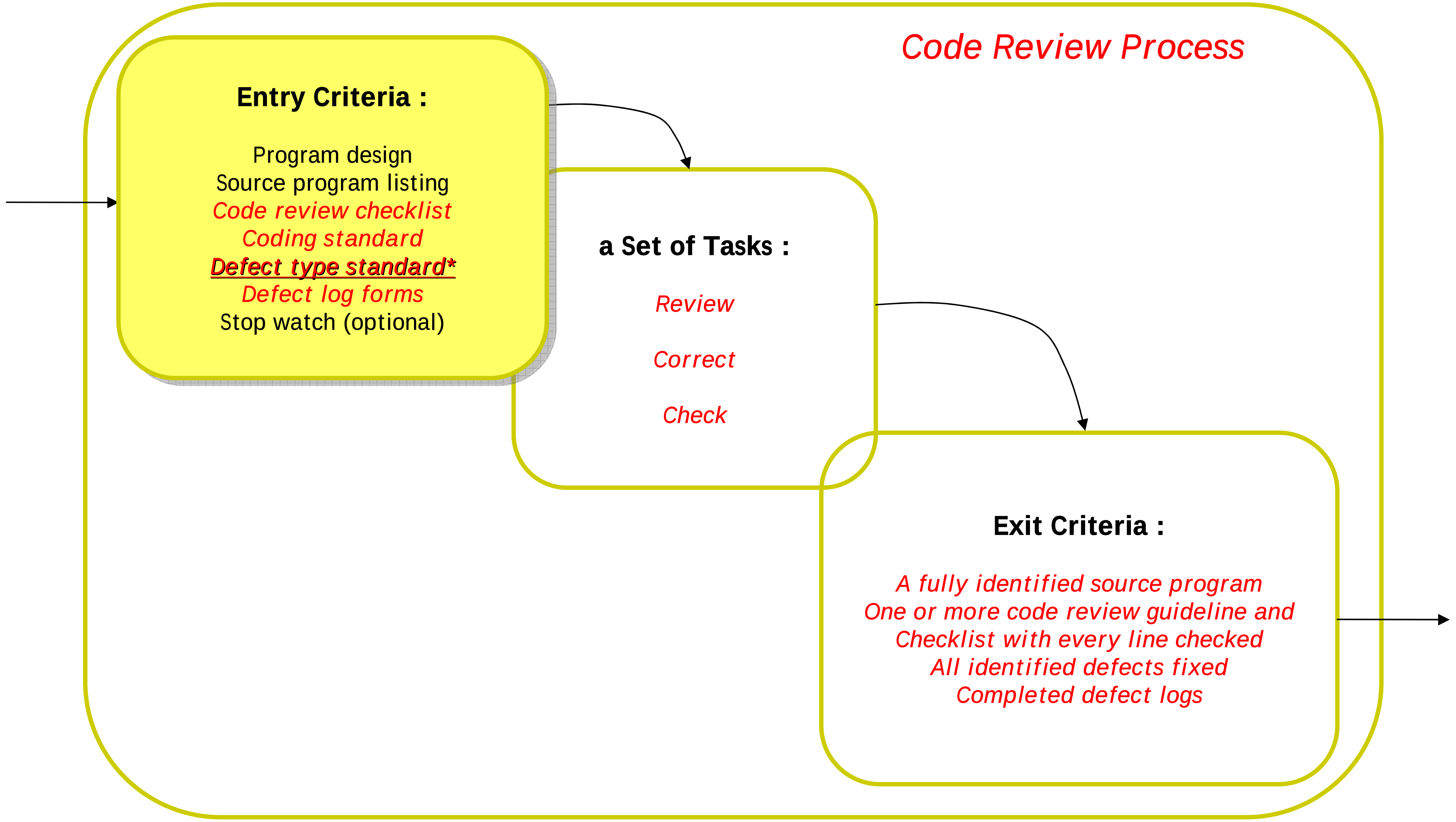
Blank Spaces	● 프로그램 작성 시 충분한 space를 (여유공간) 확보 ● 모든 프로그램은 반드시 한 줄 이상씩 띄어 써라
Indenting	● 괄호의 레벨마다 들여쓰기를 해라 ● 괄호의 시작과 끝의 위치를 일정하게 유지하라

Indenting Example	`While (miss_distance > threshold)`
	`{`
	` Success_code = move_robot (target_location) ;`
	` If (success_code == MOVE_FAILED)`
	` {`
	`  Printf ("The robot move has failed. \n") ;`
	` }`
	`}`

Capitalization	● 모든 defines는 대문자를 사용하라 ● identifiers and reserved words와 같은 다른 경우는 소문자를 사용하라 ● 사용자에게 보여지는 문자는 식별성을 최대한 살릴 수 있게 대소문자를 적절하게 섞어 사용

Capitalization Example	`#define DEFAULT-NUMBER-OF-STUDENT 15`
	`Int class-size = DEFAULT-NUMBER-OF-STUDENT ;`

Code Review Process



Defect Type Standard from PSP0

유형번호	유형 이름	설명
10	문서 (Documentation)	주석, Message
20	구문 (Syntax)	철자법, 구두법, 오타, 명령어 형식
30	통합 (Build, Package)	관리변경, Library, Version Control
40	대입 (Assignment)	선언, 이름중복, 범위, 한계
50	인터페이스 (Interface)	Procedure 호출과 참조, 입출력, 사용자 형식
60	확인 (Checking)	결함표시, 부적절한 확인
70	데이터 (Data)	구조, 내용
80	함수 (Function)	논리, Pointer, 반복, 재귀, 계산, 기능 결함
90	시스템 (System)	구성, 시간, Memory
100	환경 (Environment)	설계, Compile, Test, 다른 보조 시스템 문제

(*R. Chillarege*, IEEE TSE 1992)

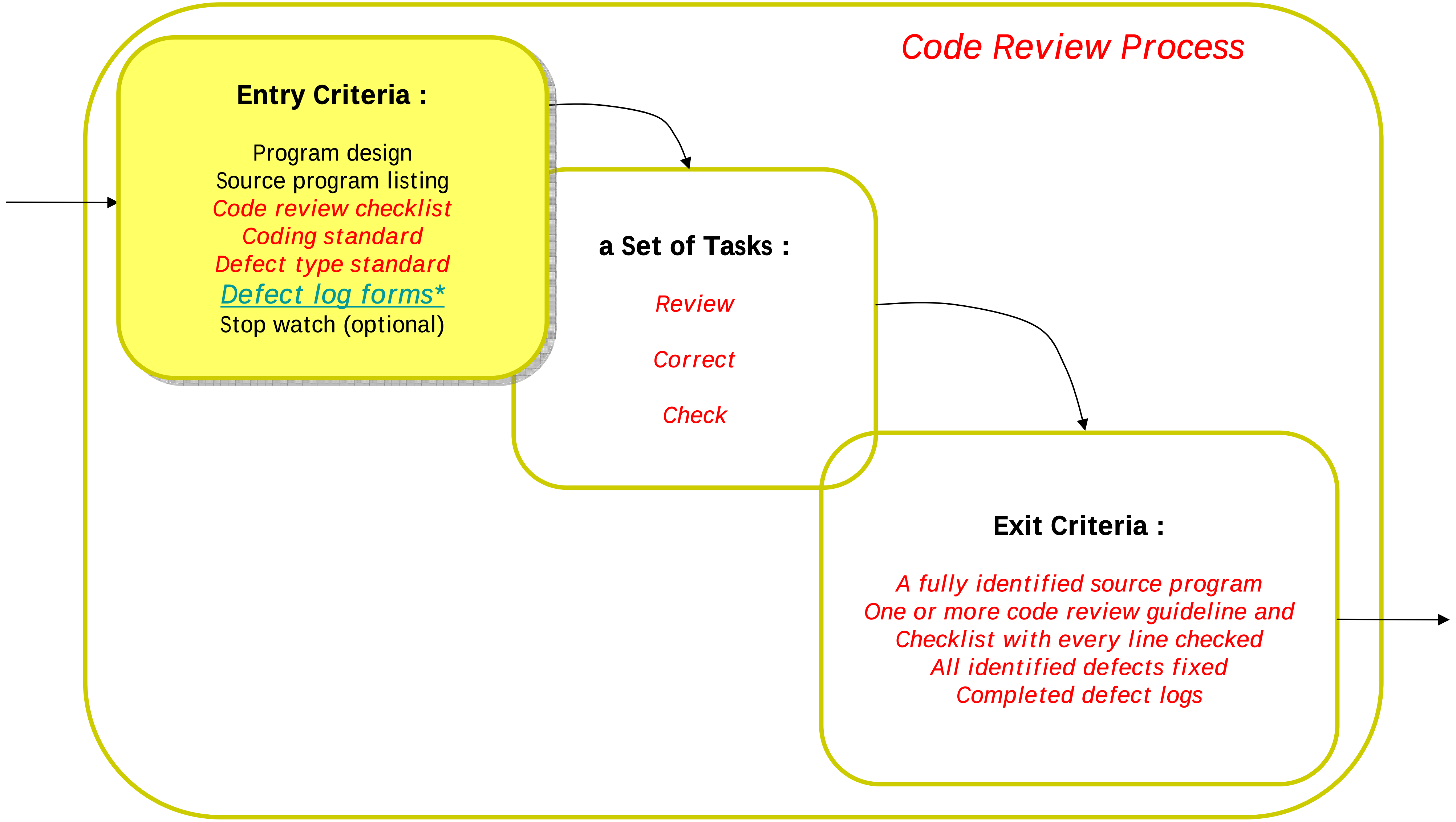
Extended Defect Type Standard

유형번호	유형 이름	설명	%
10	문서 (Documentation)	Comment, messages, manuals	1.1
20	구문 (Syntax)	일반적 syntax problem	0.8
21	**Typos**	**Spelling, punctuation**	**32.1**
22	Instruction formats	General format problem	5.0
23	Begin-end	Did not properly delimit operation	0
30	통합 (Build, Package)	변경 관리, system build, Version Control	1.6
40	대입 (Assignment)	일반적 assignment problem	0
41	Naming	Declaration, duplicates	12.6
42	Scope		1.3
43	Initialization and scope	Variables, objects, and so on	4.0
44	**Range**	**Variable limits, array range**	**0.3**

Extended Defect Type Standard (Cont'd)

유형번호	유형 이름	설명	%
50	인터페이스 (Interface)	일반적 interface 문제	1.3
51	internal	Procedure 호출과 참조(reference)	9.5
52	I/O	File, display, printer, communication	2.6
53	User	Formats. Contents	8.9
60	확인 (Checking)	결함표시, 부적절한 확인	0
70	데이터 (Data)	구조, 내용	0.5
80	함수 (Function)	일반적 logic	1.8
81	Pointers	Pointers, strings	8.7
82	Loops	Off-by-one, incrementing, recursion	5.5
83	Application	Computation, algorithmic	2.1
90	시스템 (System)	구성, 시간, Memory	0.3
100	환경 (Environment)	설계, Compile, Test, 다른 보조 시스템 문제	0

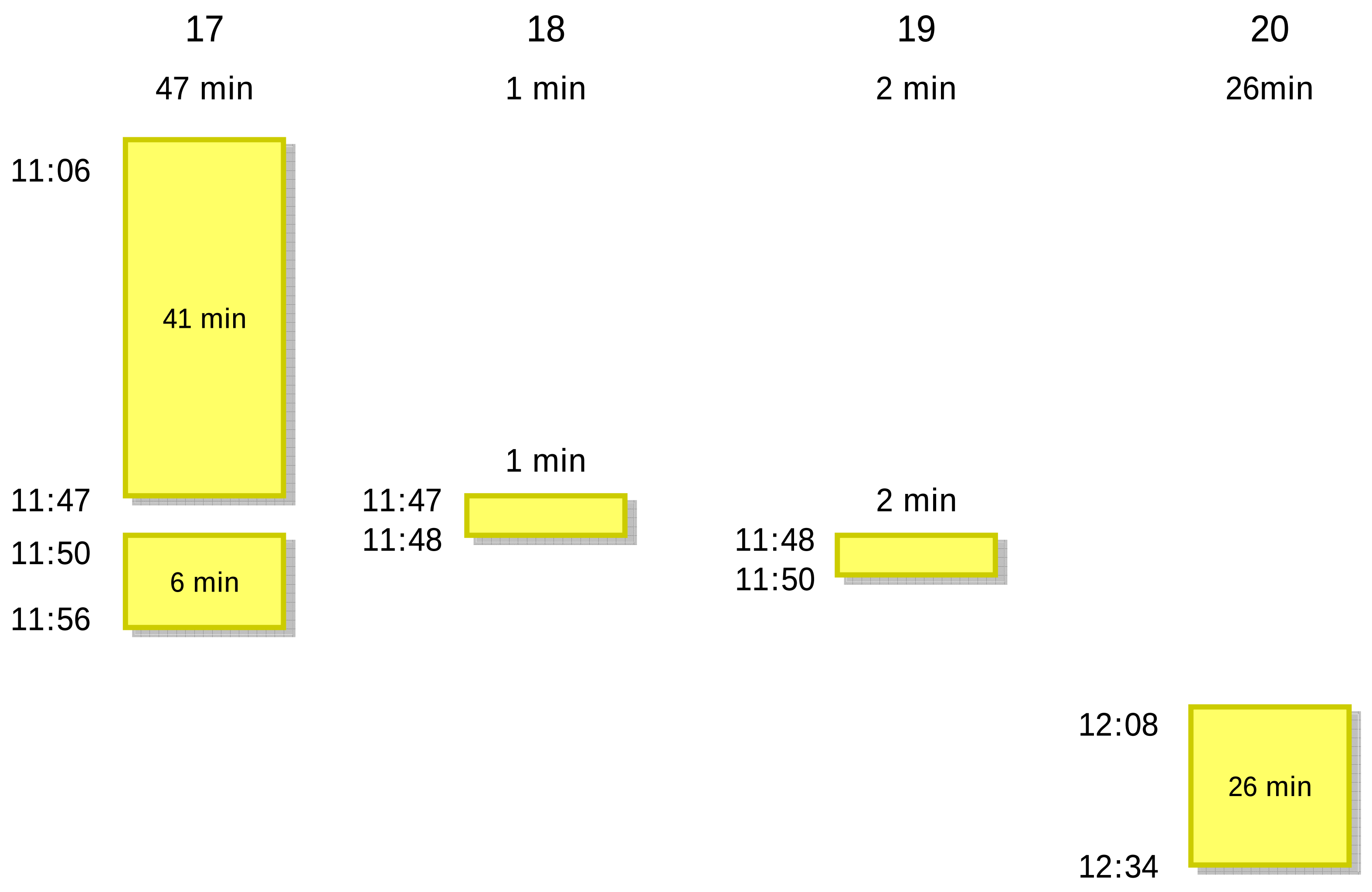
Code Review Process



Defect Recording log 를 작성해야 하는 이유

- 프로그래밍을 향상시키기 위해서
  - Defect data는 일반적으로 프로그래밍 능력을 향상시키는 데 큰 도움을 준다고 알려져 있다
  - Defect를 이해하지 못한다면 관리할 수 없다
- 프로그램의 **Defect를 줄이기 위해서**
  - 관심을 갖고 적합한 방법을 이용한다면 defect를 줄일 수 있다
- 시간을 절약하기 위해서
  - Mistake는 더 큰 disaster로 다가온다
  - Defect가 프로그램에 머무는 시간이 길어질 수록 발견하는 시간도 길어지고 수정도 어려워짐
- 비용 절감을 위해서
  - Unit test 후나 test 단계가 바뀔 때 마다, defect를 찾고 수정하는 비용은 약 10배씩 증가
- 책임감 있게 작업하기 위해서
  - Defect는 개발자들이 만들었으므로 그것을 찾아 수정하는 책임도 있다

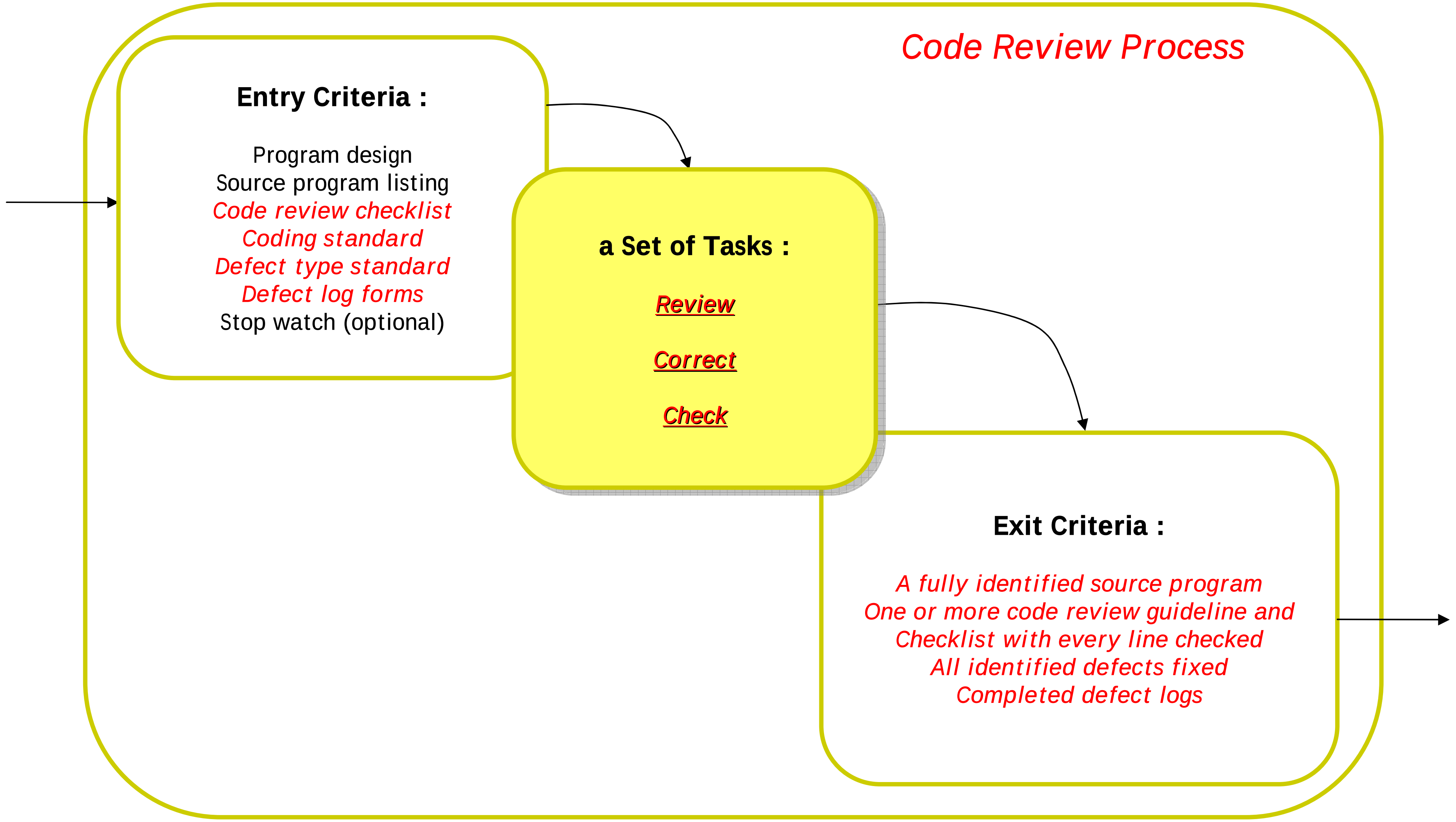
Multiple Defect Problem



Defect Recording log 에 대한 Tip

- 프로그램을 시작할 때, defect recording log를 여러 장(여러 파일) 준비한 후, 첫 장의 첫 부분부터 채워나가라
- 첫 장의 모든 빈칸을 채운 다음에는 둘째 장을 시작하기 전에 머리글 부분을 완성해라
- 처음으로 defect를 발견했을 때는, 일련 번호를 일지에 기입하고, 해당 defect를 완전히 fix하기 전까지는 나머지 데이터를 기입하지 마라
- 각각의 defect 마다 분리된 라인을 사용하라
  - 만약, 하나의 defect으로 Compile 시 여러 개의 결함 메시지가 났다면, 하나의 defect이다
  - 즉, 프로그램을 한번 고칠 때마다, defect recording log에는 단 한 개의 defect만 기입된다
- Defect을 발견한 날짜를 기입해라. 하루에 여러 개의 defect를 발견했다면 날짜를 생략할 수 있다
- Defect type의 판단이 혼란스러워도 자신의 판단에 의지하라. 정확한 type을 알아내기 위하여 본 단계에서 너무 고민하지 마라
- Defect가 형성된 단계(injected) 역시 언제나 명확한 것은 아니지만, 소규모의 프로그램에서는 문제가 되지 않는다
  - 너무 고민하느라 시간을 허비하지 마라
- Removed 단계는 일반적으로 defect을 발견한 단계와 동일하다
- Fix defect는 다른 defect을 고치는 동안 발생된 defect을 의미한다
- Description 란은 나중에 보더라도 이해할 수 있는 수준에서 최대한 간략하게 기술한다

Code Review Process



Review, Correct, Check

1	Review 절차	- 먼저 프로그램의 최종 소스 코드를 완성한다 - 프로그램의 컴파일과 테스트전에 소스 코드를 출력한다(optional) - Code review를 진행한다 - Code를 review할 때는 가능한 많은 수의 defect를 찾아서 fix 할 수 있도록 한 라인 한 라인, 신중하게 review 한다
2	Defect fixing	- 발견된 모든 defect을 fix한다 - 제대로 fix되었는지 확인한다 - **Defect Record log**에 기록한다
3	검토 범위	- 프로그램 설계 과정이 요구 사항에 기술된 모든 기능을 수행하는 지 확인한다 - 소스 코드에 모든 설계 사항을 구현했는지 확인한다
4	프로그램 논리 검토	- 설계 논리가 맞는 지 확인한다 - 프로그램에서 설계 논리를 제대로 구현했는지 확인한다
5	Name/Type 확인	- 모든 name과 type을 제대로 선언한 후에 사용하는지 확인한다 - Integer, signed integer, float point type을 적절하게 선언했는지 확인한다
6	모든 변수 확인	- 모든 변수가 초기화 되었는지 확인한다 - Overflow, underflow, out-of-bound 문제를 확인한다
7	프로그램 구문 확인	- 소스 코드가 언어 특성을 제대로 따르고 있는지 확인한다

Review의 효율성을 측정하는 measures

Review 할 프로그램 size (LOC)
Review에 걸리는 시간(minute)
발견한 defect의 개수
발견하지 못한 defect의 개수(escape)

derive

Review 양품률(review yield)
결함률(defects/KLOC)
defects/hour of review time
LOC reviewed/hour
DRL (defect removal leverage)

Review Yield

- 해당 phase (design or code)의 review를 통하여 해당 phase에 inject된 defect를 발견해내는 비율
  - Review yield = (defects found in the review/total defects injected in the phase) * 100
  - 정확한 계산은 불가능하다(전체 결함 수를 알기 어려우므로)

Phase	Defects Found	Defects Injected * 하기의 단계는 defect가 어느 단계에서 발생했는지 판단하는 시점을 의미					
		설계 검토	코딩	코드 검토	컴파일	테스트	Post Development
Planning	0	1	1	1	1	1	1
상세 설계	0	2	3	4	4	5	6
설계 검토	3						
코딩	1			7	13	15	17
코드 검토	8						
컴파일	6						
테스트	3						
Post Development	3						
합계	24						
Yield							
설계 검토		3/3=100%	3/4=75%	3/5=60%	3/5=60%	3/6=50%	3/7=42.9%
코드 검토				8/8=100%	8/14=57.1%	8/17=47.1%	8/20=40%
Total process	12	3/3=100%	4/4=100%	12/12=100%	12/18=66.7%	12/21=57.1%	12/24=50.0%

Review Yield (Cont'd)

Phase	Injected	Removed	Injected 누계	Removed 누계	Net Escapes
Planning	1	0	1	0	1
상세 설계	5	0	6	0	6
설계 검토	0	3	6	3	3
코딩	15	1	21	4	17
코드 검토	0	8	21	12	9
컴파일	0	6	21	18	3
테스트	0	3	21	21	0
Total	21	21			

* Phase yield = 100*{Removed_in_phase/(Removed_in_phase + Net Escapes)}

* Design_review_ yield = 100*{3/(3 + 3)} = 50 %

* Code_review_ yield = 100*{8/(8 + 9)} = 47.1 %

* Compile_ yield = 100*{6/(6 + 3)} = 66.7 %

* Process_ yield = 100*(Removed before compile)/(Removed before compile + escapes into compile and test)

* Process_ yield = 100*{(3+1+8)/(3+1+8+9)} = 100 * (12/21) = 57.1 %

Yield Calculation (Summary)

	# of Defects Found	# of Defects Injected	Yield
Code Review	a	a	a/a * 100 = 100 %
Compile	b	b + a	a/(b + a) * 100
Test	c	c + b +a	a/(c + b + a) * 100

❖ 좋은 review 프로세스는 yield 가 점점 높아져야 함 !

❖ 매번 review 시마다, review yield를 계산해 놓는 습관이 중요함 !

Review measures 에 관한 실험의 결론

- For **review yield**
  - ➢ 실험에 의하면, review yield가 높은 코드가 반드시 defect가 많은 코드라는 뜻은 아니다
- For **defect/hour**
  - ➢ Defect/hour가 줄어들 때, yield가 증가하지 않으면 효율성이 떨어지는 review 작업이다
- For **결함률 (defects/KLOC)**
  - ➢ High-yield code review는 시간당 200 LOC 정도가 적당하였다
  - ➢ Code inspection의 실험값과 유사한 결과임
  - ➢ Code inspection에서는 300 LOC/hour이 한계값 이라고 알려져 있음)
  - ➢ 실험에 의하면, 경험 있는 SW 엔지니어의 결함률(defects/KLOC)은 50 ~ 250 사이
  - ➢ 소프트웨어 그룹들의 평균 결함률 역시 100 정도임
  - ➢ 결함 관리 교육을 받은 엔지니어들은 평균 50 정도 형성
  - ➢ 1,000 KLOC의 코드의 경우, 평균적으로 50,000 개의 defect가 있다는 결론임

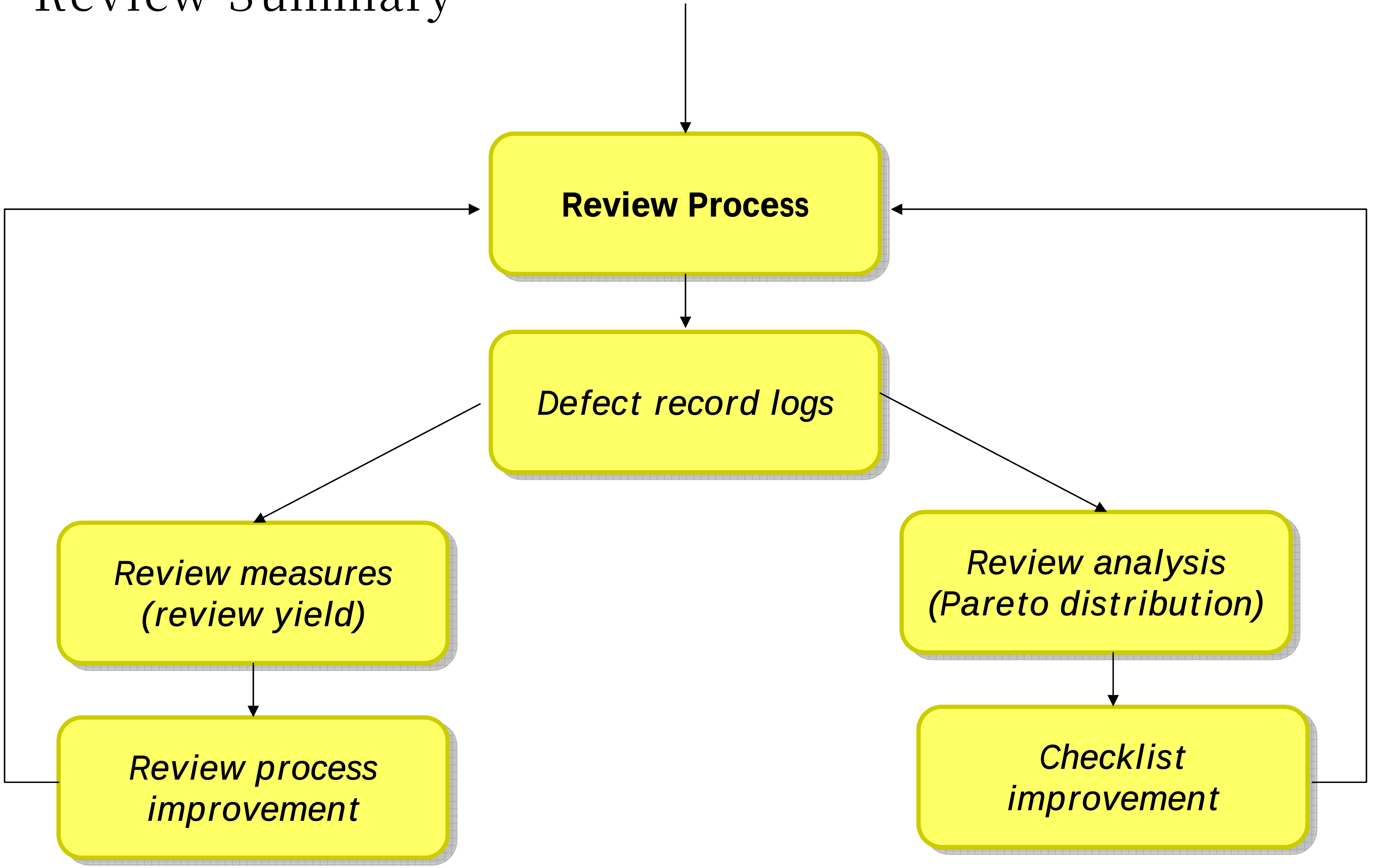
Review measures 에 관한 실험의 결론 (Cont'd)

- For DRL (Defect Removal Leverage)

$$DRL = \frac{Defects/Hour(Phase)}{Defects/Hour(UnitTest)}$$

Phase	Defects Removed per Hour	DRL
25 C++ Programs		
Design Reviews	3.91	3.91/1.39 = 2.8
Code Reviews	**5.01**	**5.01/1.39 = 3.6**
Compile	9.43	9.43/1.39 = 6.8
Unit Test	1.39	1.39/1.39 = 1.0
36 Pascal Programs		
Design Reviews	3.12	3.12/1.31 = 2.4
Code Reviews	**3.15**	**3.15/1.31 = 2.4**
Compile	7.99	7.99/1.31 = 6.1
Unit Test	1.31	1.31/1.31 = 1.0

Review Summary


Review Process
Defect record logs
Review measures (review yield)
Review process improvement
Review analysis (Pareto distribution)
Checklist improvement

Reference

- KAIST SPIC, APSEC 2004 보고서
- KAIST SE LAB (배두환 교수), “Introduction to Software Engineering”, course material
- CMU SEI, “Models of Software Evolution : Life Cycle and Process”, SEI-CM-100-1.0, 1987
- Watts S. Humphrey, *Introduction to Personal Software Process*, Pearson Addison-Wesley, 2003
- Watts S. Humphrey, *A Discipline for Software Engineering, Addison-Wesley*, 1995
- 송태국, 이병헌, *CMM 역량 성숙도 모델*, Pearson Addison-Wesley, 1994
- Dennis M. Ahern et al, *CMMI Distilled*, Addison-Wesley, 2003
- R.L. Glass, *Facts and Fallacies of Software Engineering*, 인사이트, 2004

Thank you!