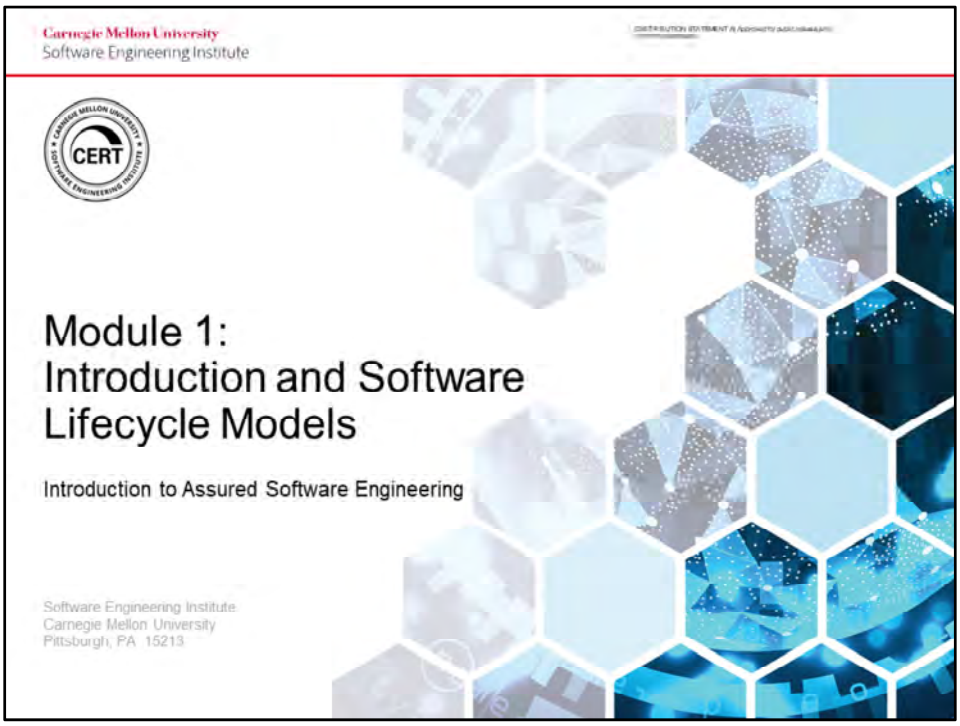
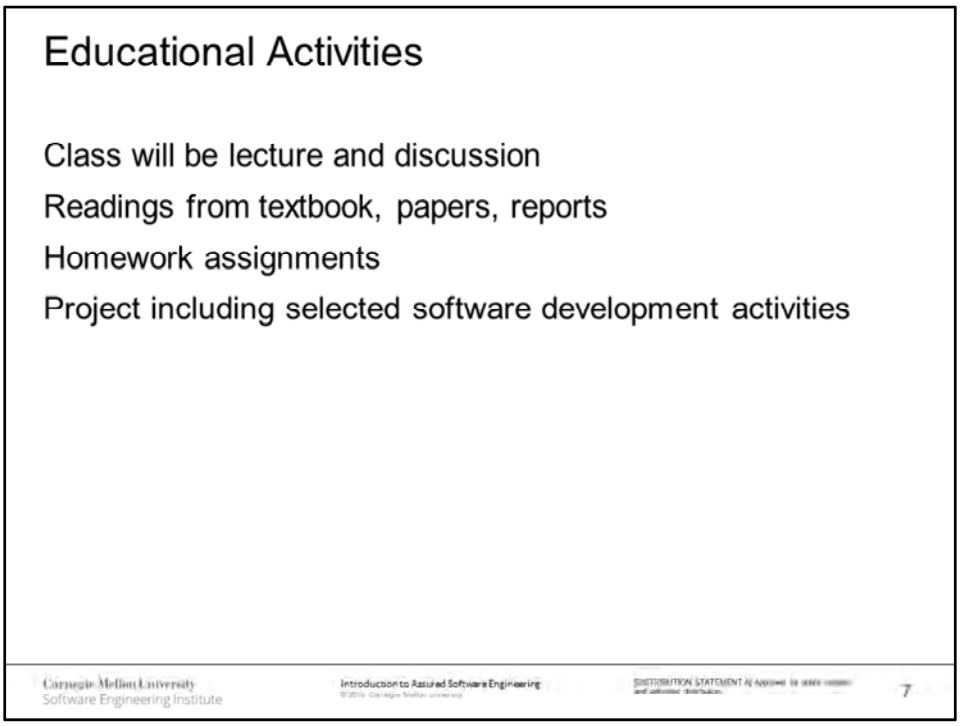
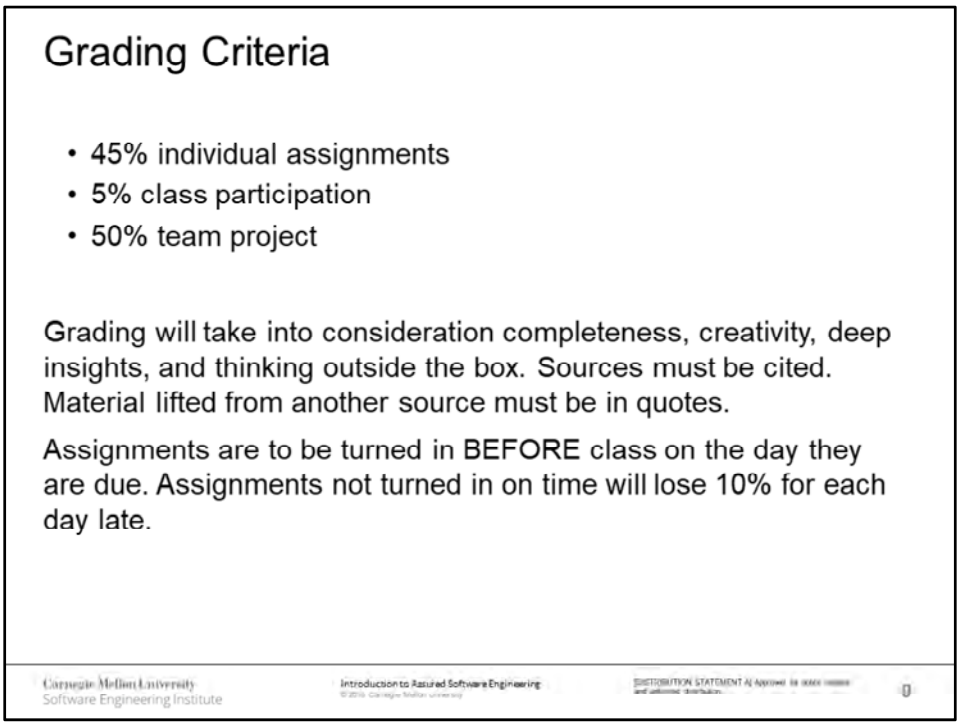


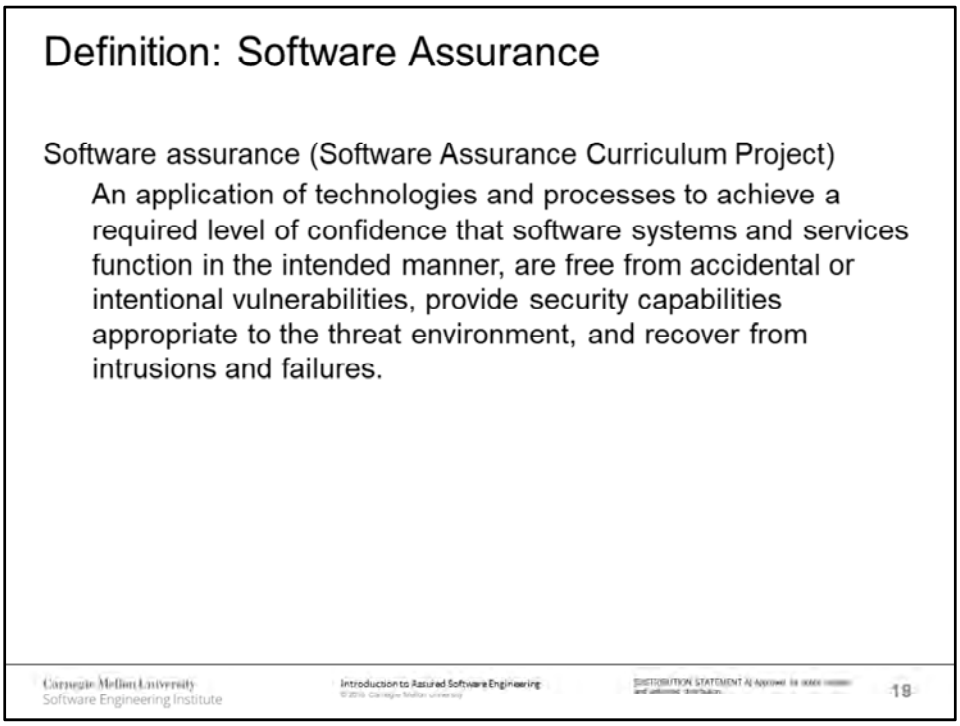
This material is based upon work funded and supported by the Independent Agency under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.



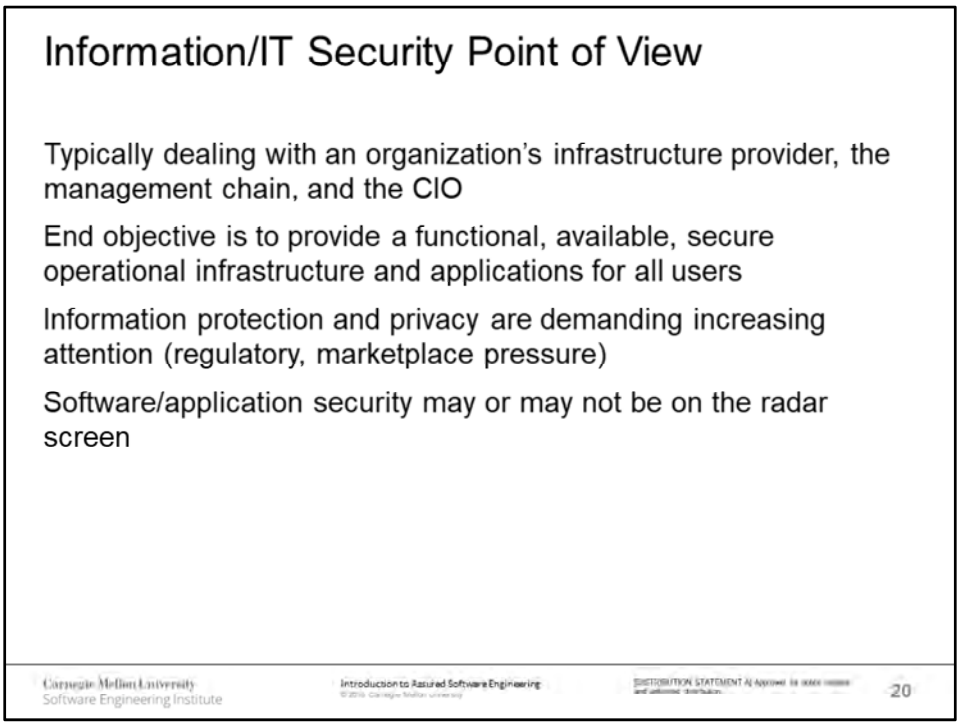
Note to instructor: This course material does not necessarily include assignments, exercises, or proposed projects.
These would need to be provided by the instructor or the slides would need to be modified accordingly.



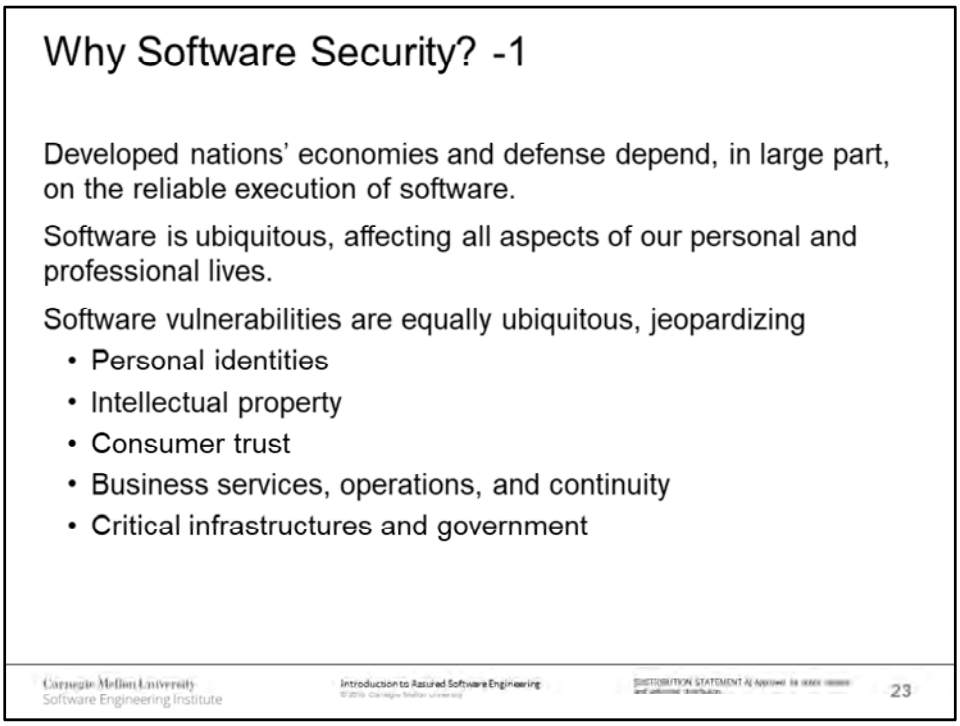
Instructor note: These are just suggestions based on our experience. Feel free to modify as desired for your classes.



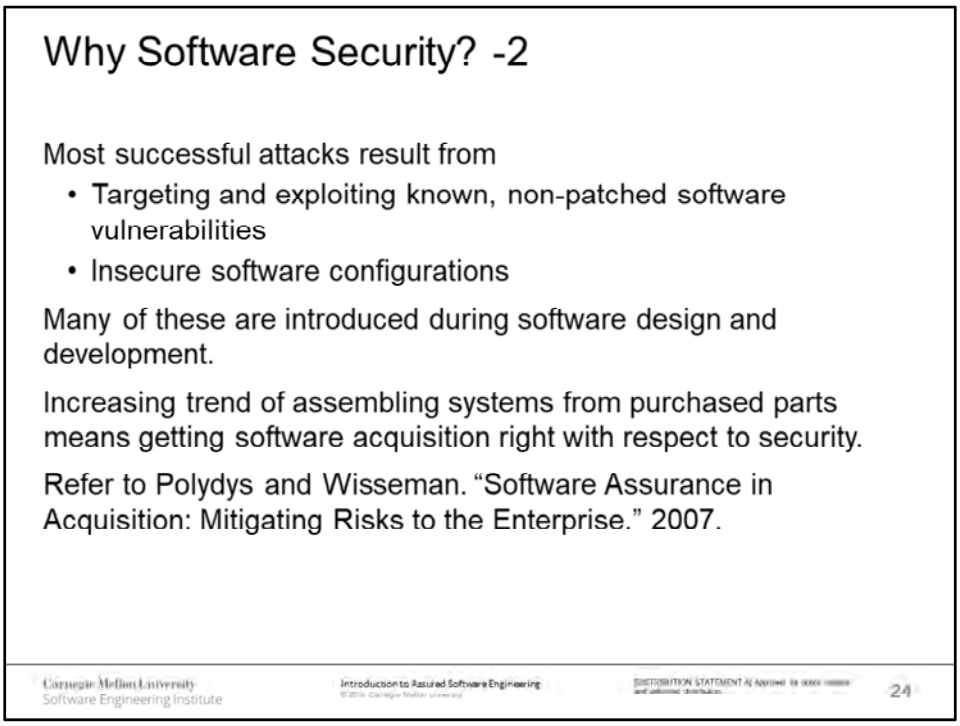
First appearance - comes up a little later in the lecture too.



Typically constrained by more requirements, needs, and wants from the business than resources, cost, and schedule permit. Portfolio management driven by business objectives is a common approach for prioritizing.



Where is software present? Your car? Your apartment? On airplanes? In critical infrastructure?

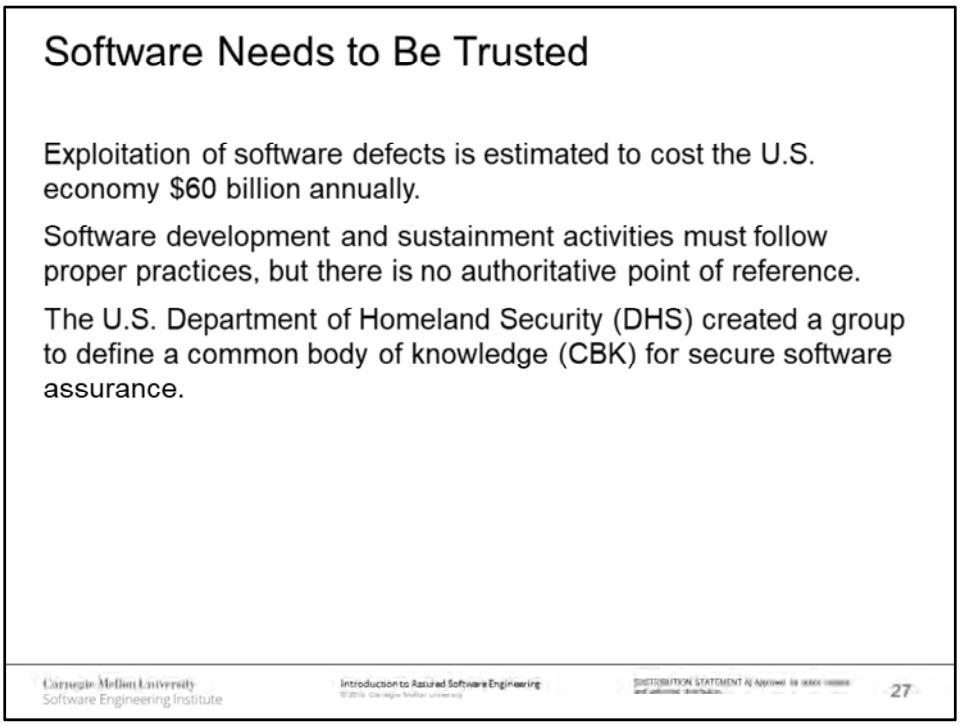


Need to ask "how much software does my organization purchase compared to how much it builds in-house?" to determine if the problem is outsourced assurance or including security in the SDLC.

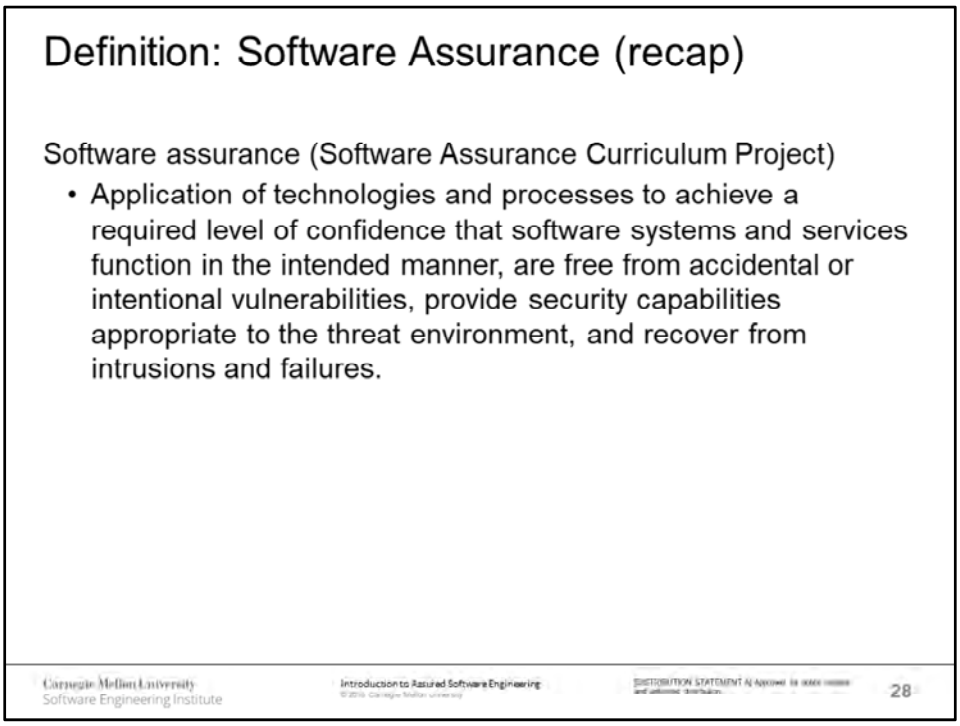
Security Perspectives

https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices

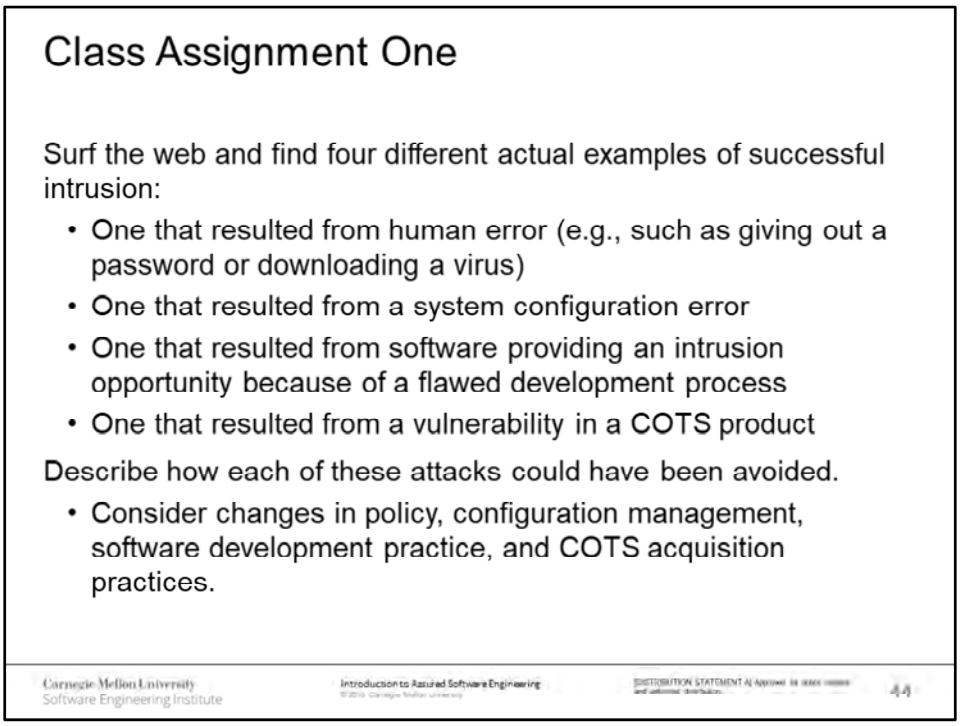
Attackers have been learning how to exploit software for several decades; the same is not true for software engineers.



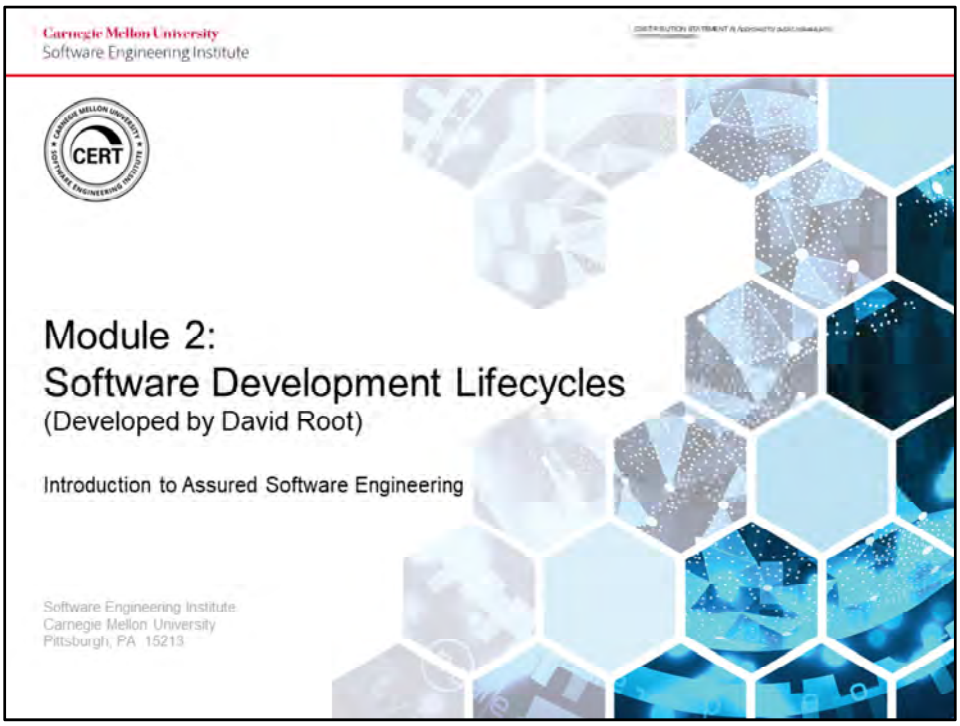
Instructor note: Suggest checking the above dollar amount. It probably changes (increases) regularly.

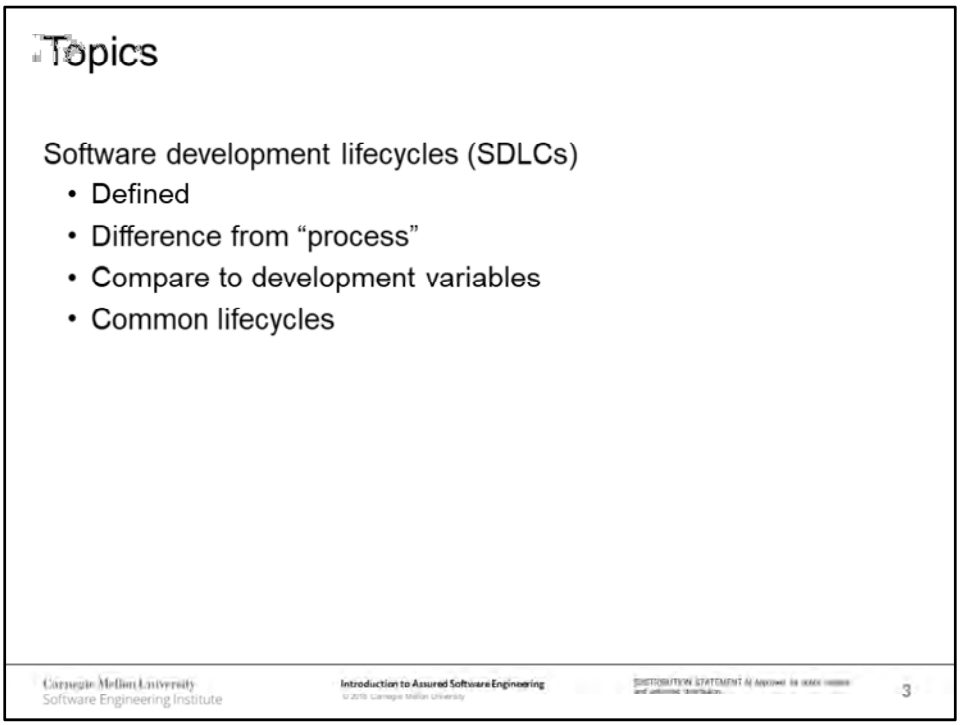


Match this to the discussion of “What is software assurance?” since the definition already appeared there.



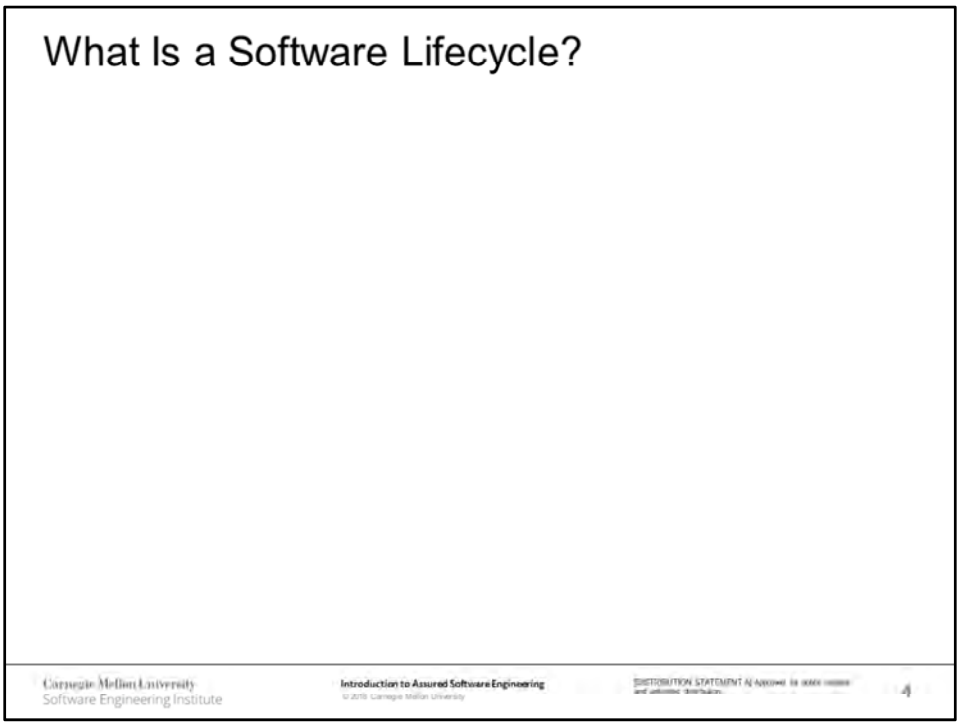
Could be an individual or a team assignment during the class.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 2:
Software Development Lifecycles
(Developed by David Root)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

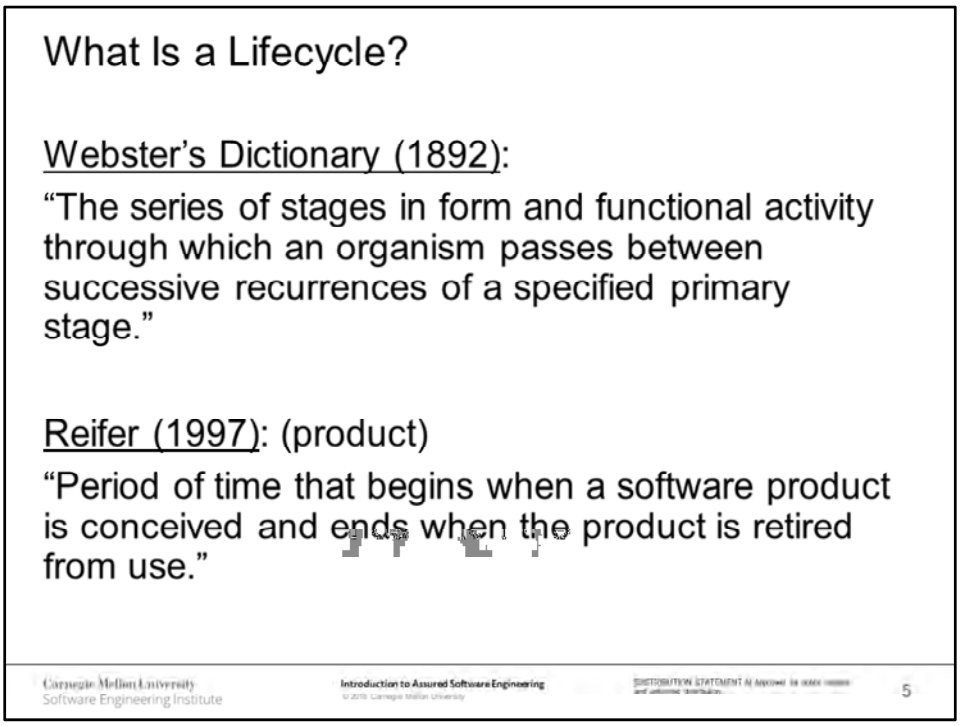


The student will be able to do the following:

- explain what a software development lifecycle (SDLC) is and give examples
- understand the difference between a SDLC and a process
- compare SDLCs to development variables and select an SDLC
- understand common lifecycles

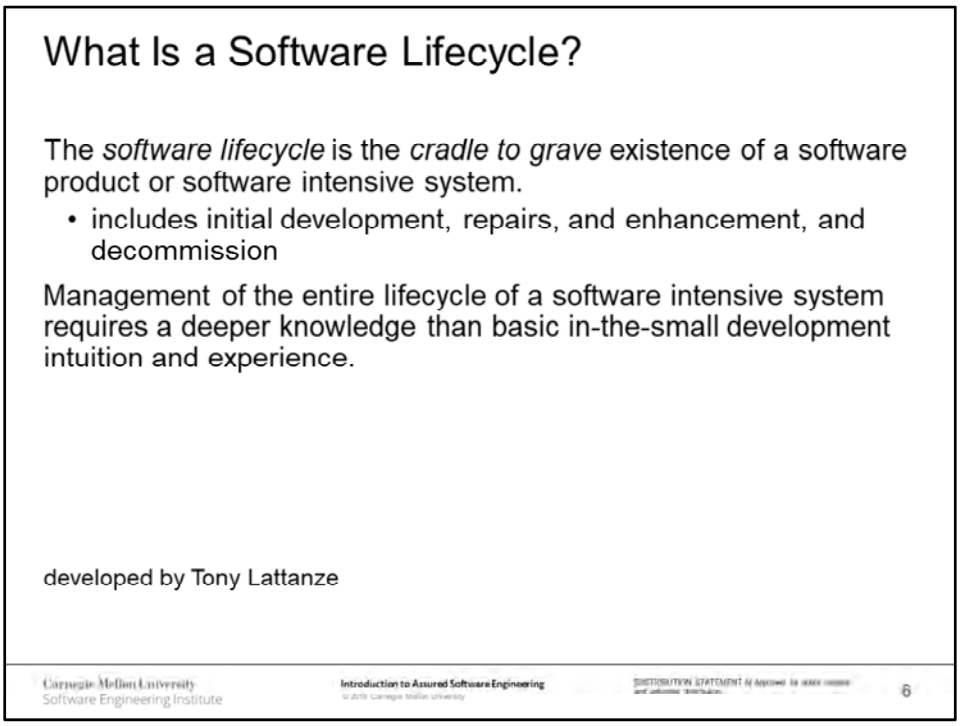


This is a chance to see how the students define this term. You may want to pass out post-it notes and have each student define the term or you may do this as an open discussion.



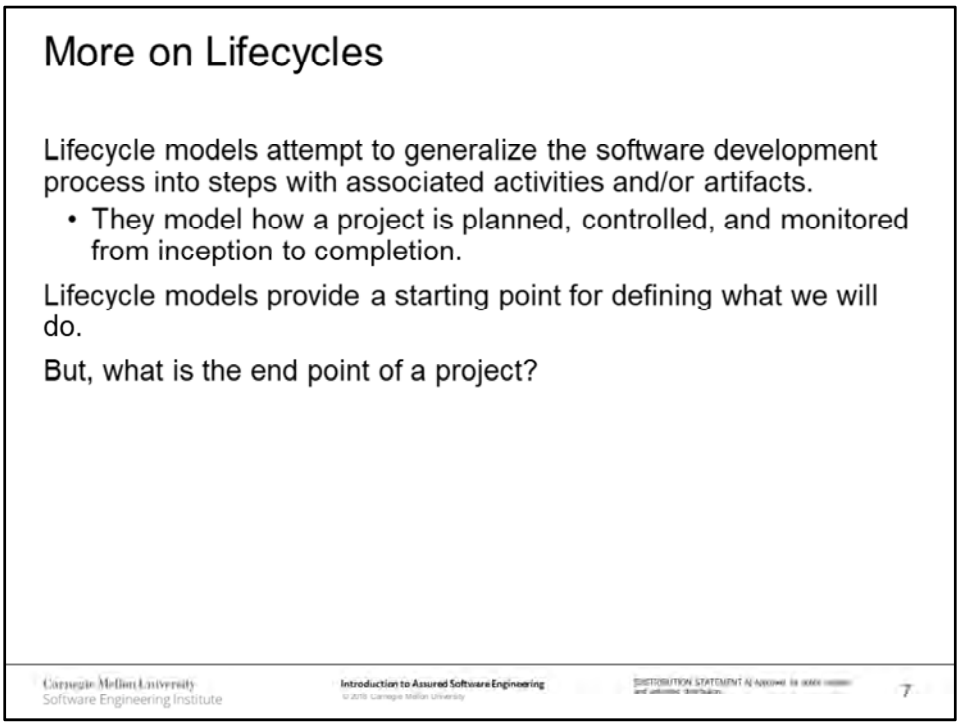
Webster definition provides a general description of lifecycle where the Reifer definition focuses on product lifecycle.

"Donald J. Reifer (Reifer Consultants Inc.) is one of the leading figures in the field of systems/software engineering and management with over 40 years of progressive experience. He has built businesses, steered troubled projects and served in executive positions in industry and government. His specializes in the area of metrics and measurement."



A framework that describes the activities performed at each stage of a software development project.

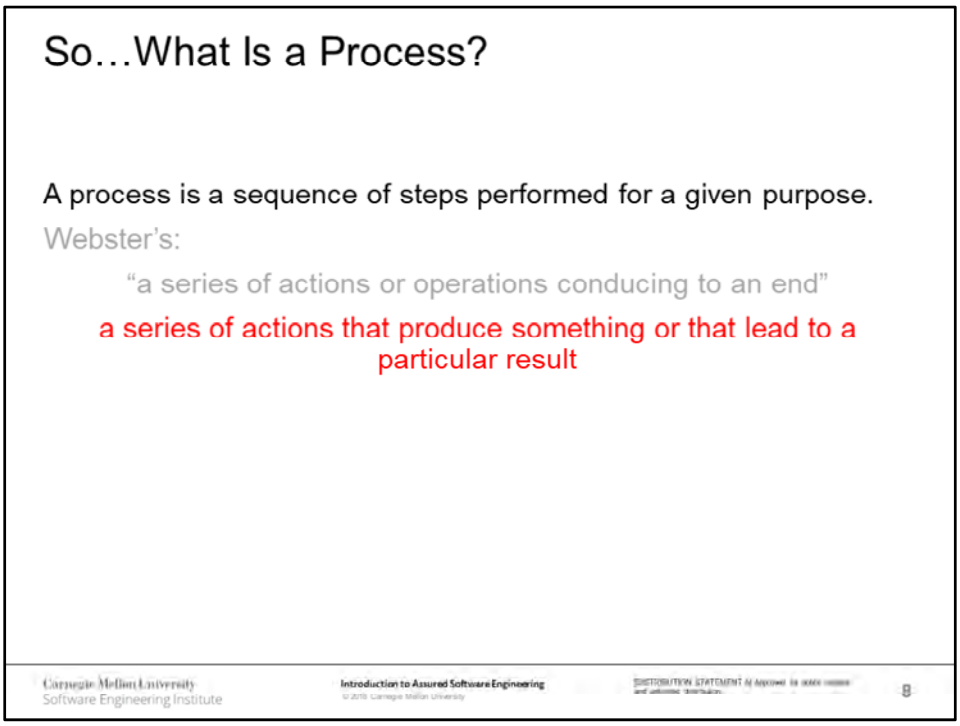
Systems engineers are often familiar with the product lifecycle so they think about a product from the beginning when it is an idea until it is no longer used and is removed (disposal). The software lifecycle is similar to a product lifecycle. However, a software *development* lifecycle is often a subset of the software lifecycle and starts with customer needs and ends when the software is delivered to the customer. It may only focus on the development activities and not be concerned with where the software fits within an overall product and may ignore maintenance, sustainment, and disposal activities.



IEEE 12207
A lifecycle model is defined as a framework of processes and activities concerned with the life of the product that may be organized into stages, which also acts as a common reference for communication and understanding.

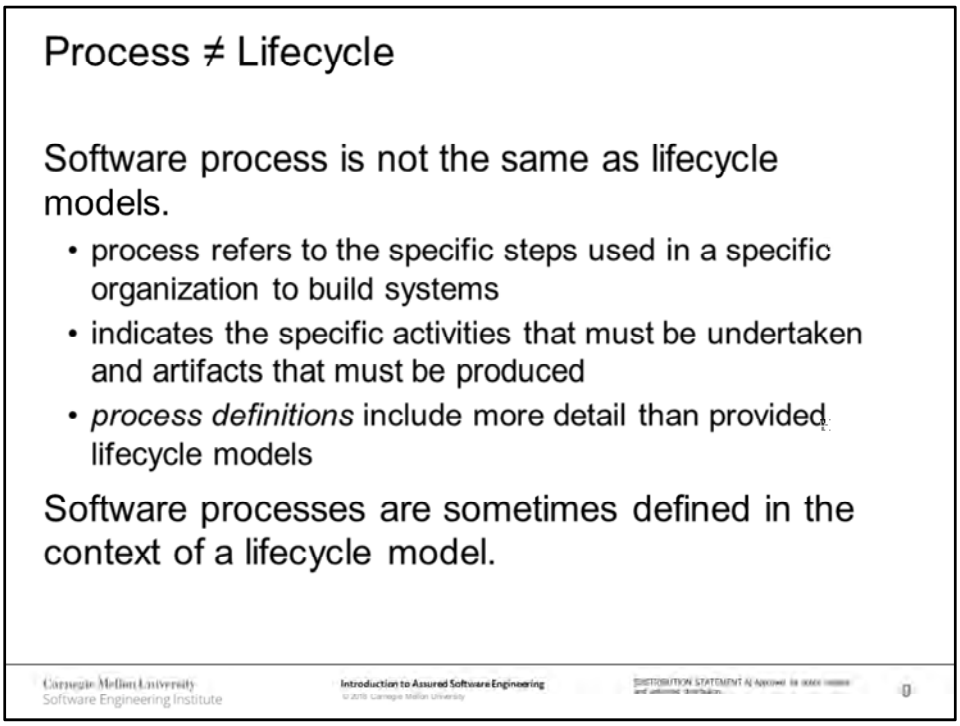
Just as the lifecycle model provides a starting point it will often provide an ending point too.

Lifecycle management methodologies aid in determining the sequence of major activities, provide a better understanding of the processes required for each activity, and serve as a starting point from which management decisions can be made. One thing to remember is that software development methodologies used should integrate with, and be consistent with, the systems engineering development methodologies used for the total product.

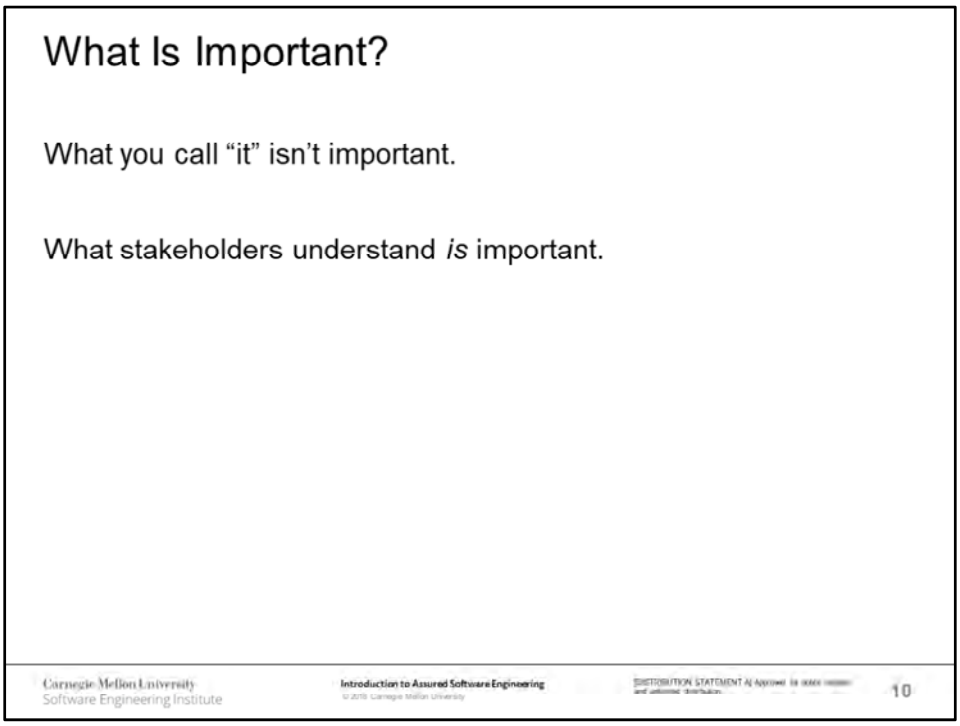


Have a discussion with your students about process. At a minimum, a process is a repeatable sequence of events to produce an outcome.

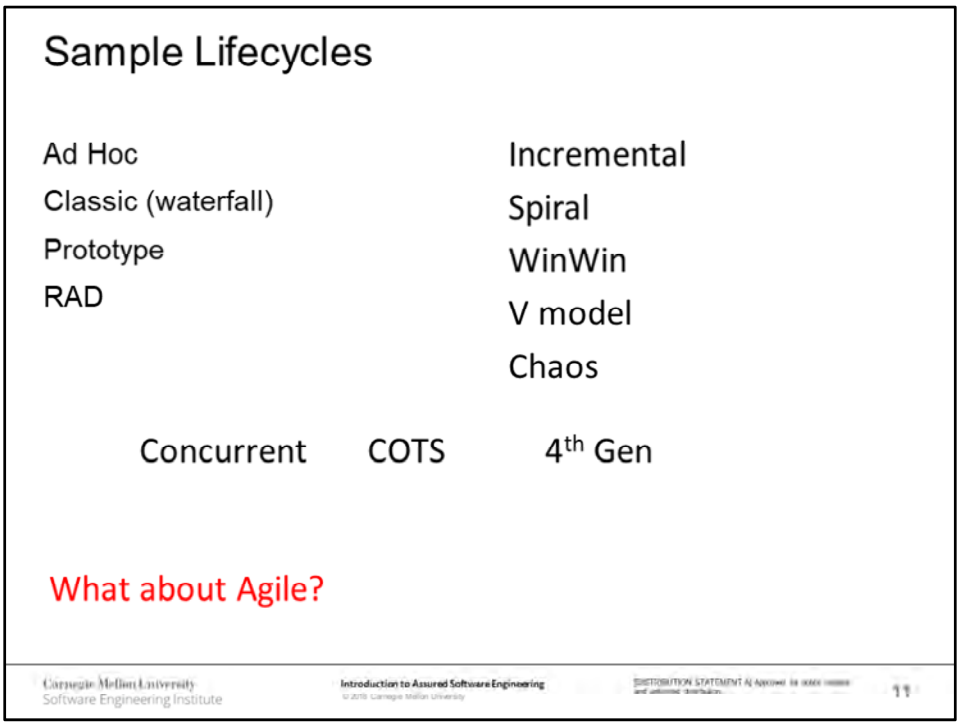
Ask how a process is different than a lifecycle?



Point out that lifecycle models define the phases that determine the sequence of major activities. Within the phases, processes define the series of steps that will be done within that phase.



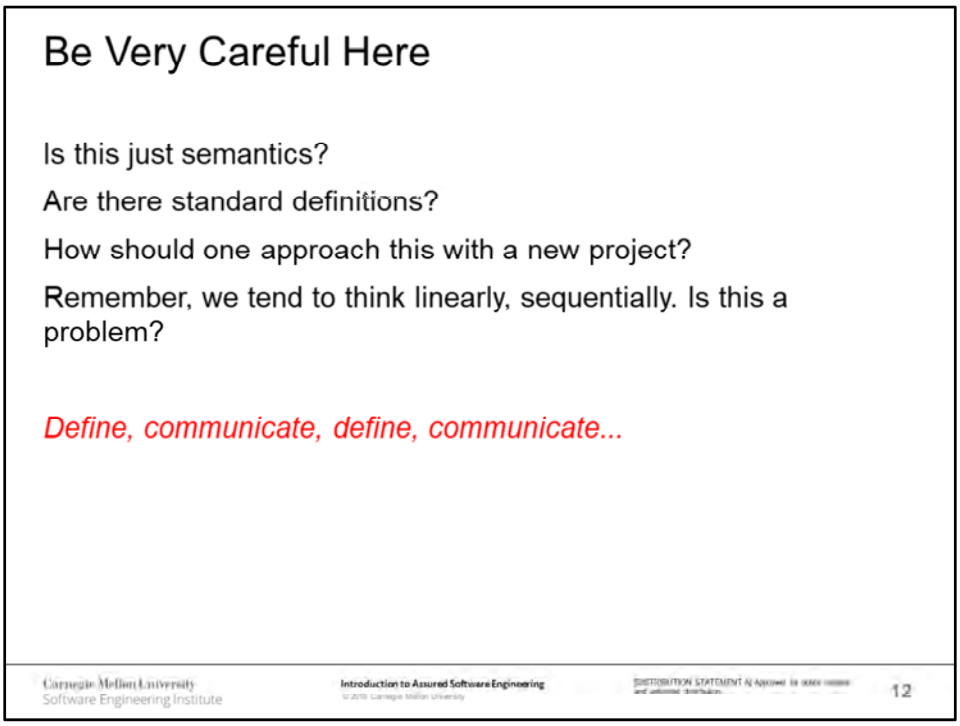
Understanding your software process and making tradeoffs between incorporating and deleting lifecycle components is crucially important for producing high quality software, on time, within budget.



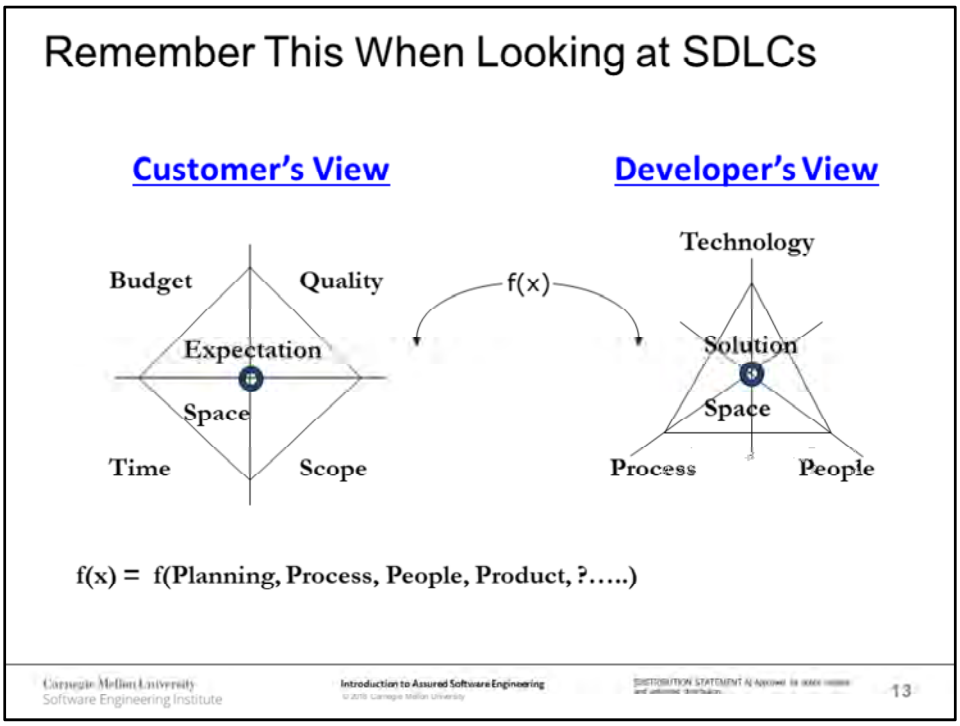
Ask to see if students recognize any of the lifecycle listed? Do they know of other lifecycles not listed?

Where does Agile fit in?

Note that each of the lifecycles on this chart will be described in this module and Agile will be discussed in the next module.

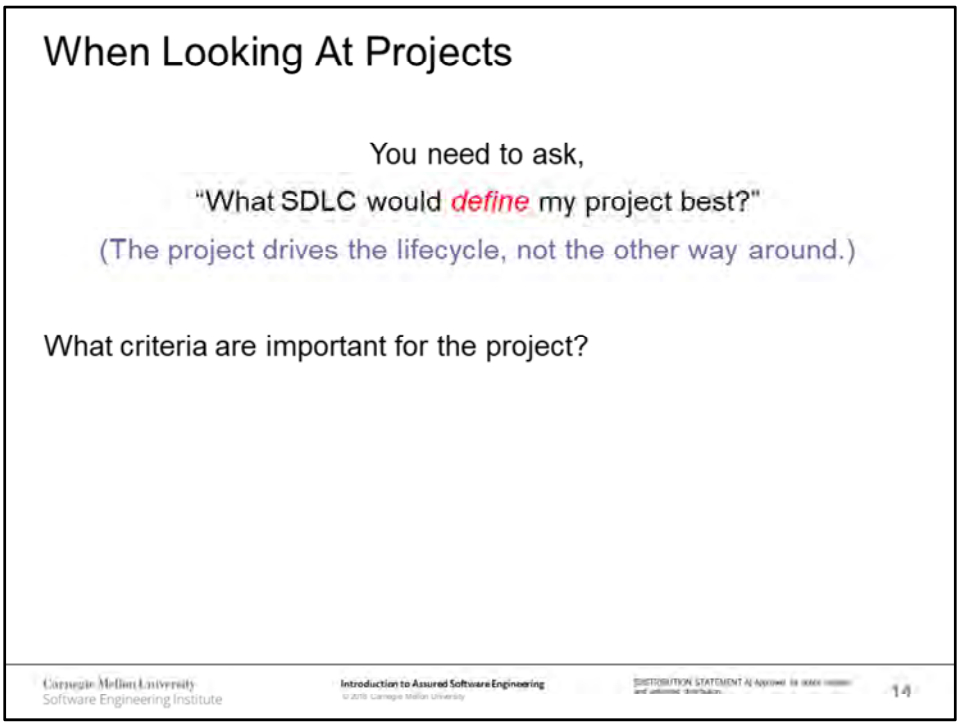


Different processes organize activities in different ways and are described at different levels of detail. Different organization may use different processes to produce the same types of product. However, some processes are more suitable than the others for some types of applications. If inappropriate processes or lifecycles are used, this will probably reduce the quality or the usefulness of the software product being developed.



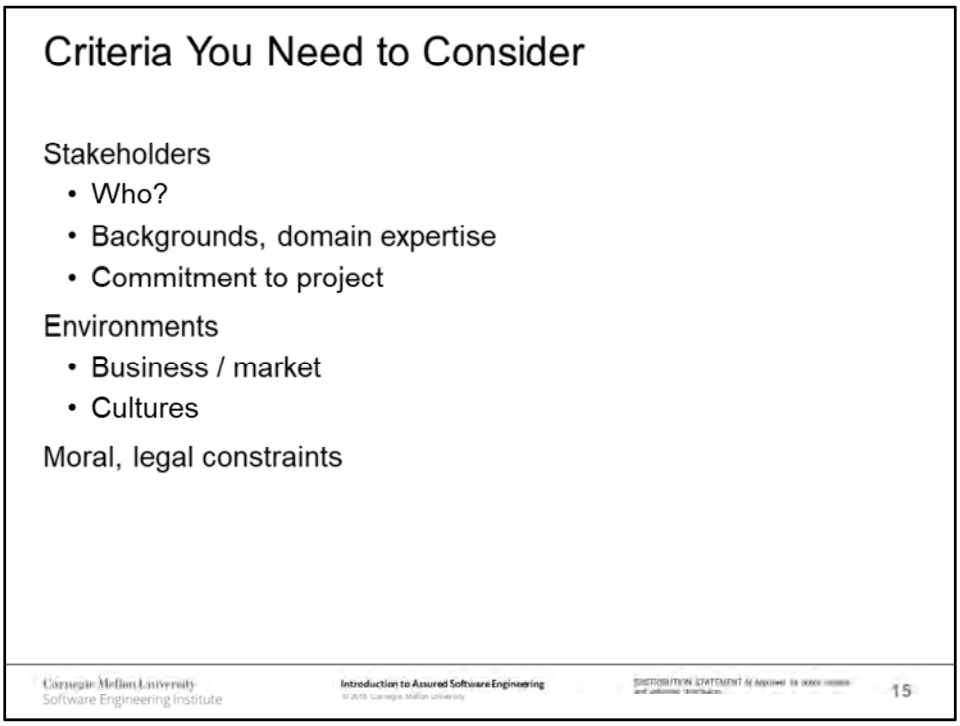
Customer's and developers often define the problem from two different perspectives. The customer is concerned about schedule, cost, features, and quality. They want to make sure that what they asked for was done on time, within budget, and will operate without issues. The customer defines the problem space. Whereas, developers look at the interactions between people, processes and the tools that will be used to solve the customer's problems. Cost, schedule, scope, and quality provide limitations. The developer looks at the solution space.

An SDLC can often help bridge the gap between these two perspectives. If the customer cannot communicate the problem succinctly, SDLCs allow the developer help the customer to define key activities that will help to understand the requirements and quality attributes necessary for the product or system.

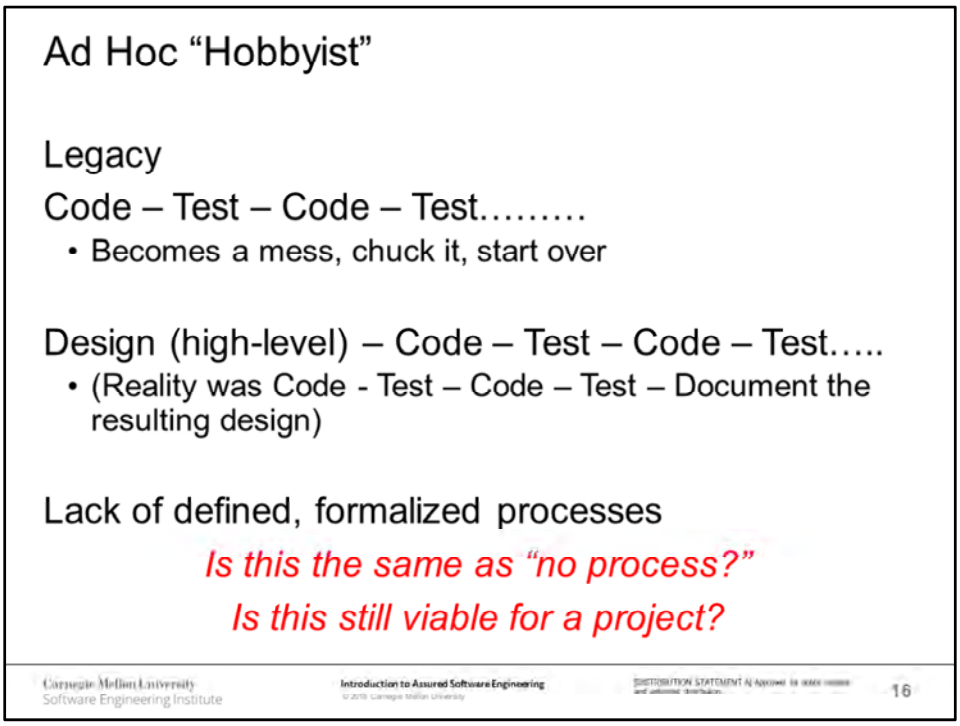


An SDLC should be chosen based on the nature of your program, software domain, the methods and tools used, and the controls and deliverables required.

The use of lifecycle management methodologies has proven to be extremely effective in controlling change and in managing the complexity of the development process. However, for any lifecycle methodology to be effective, it must be customized to specific program goals. Therefore, your selected methodology must be adapted and evolved, the same as the technical activities it ties together. Understanding your software process and making tradeoffs between incorporating and deleting lifecycle components is crucially important for producing high quality software, on time, within budget.

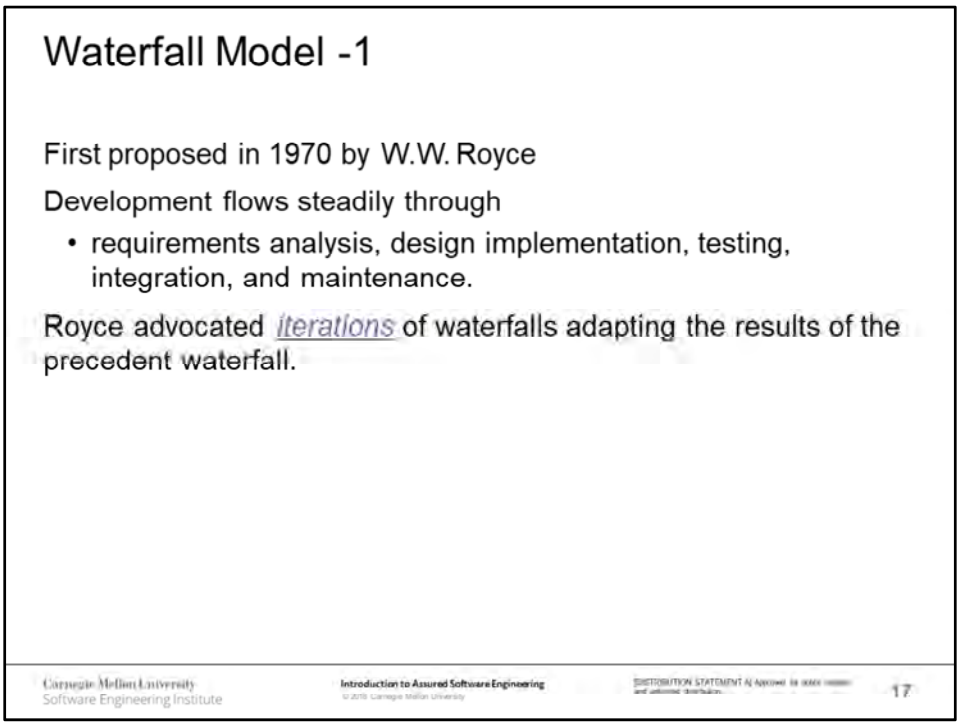


Any criteria missing?



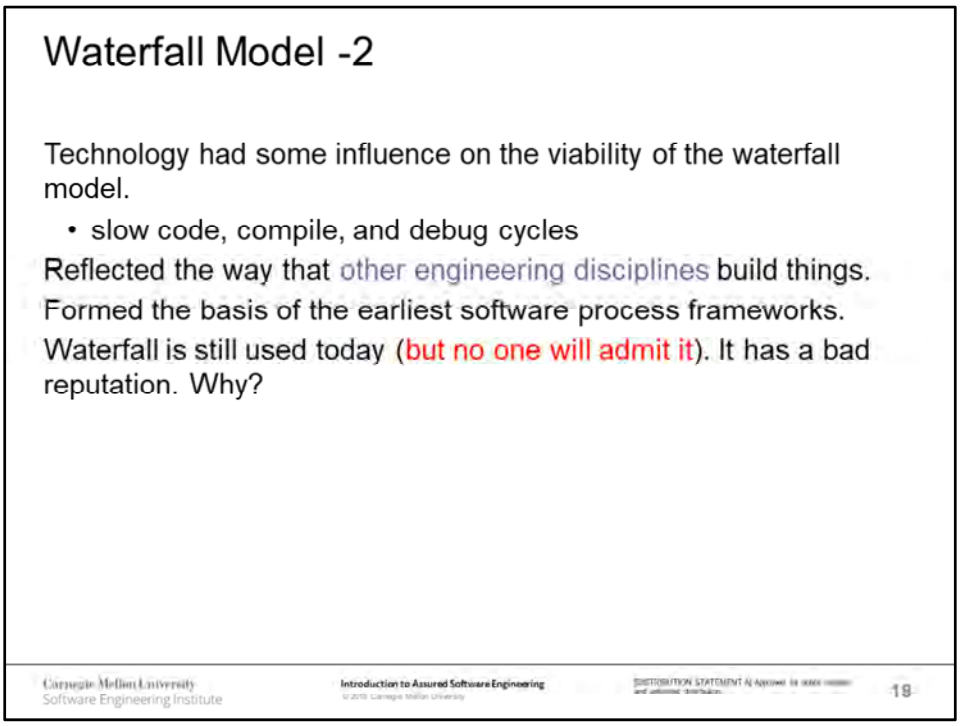
Without lifecycle models, processes, standards, etc., software was developed by code and fix .. and often times it was the user who performed the test.

Is this still done today on your project? What do you use?



One of the earliest lifecycle models was the Waterfall Model.

From the reading...
The waterfall model was first identified in 1970 as a formal alternative to the *code-and-fix* software development method prevalent at the time. [ROYCE70] The waterfall model was the first to formalize a framework for software development phases, and placed emphasis on upfront requirements and design activities and on producing documentation during early phases. The major drawback to this model is its inherent sequential nature — any attempt to go back two or more phases to correct a problem or deficiency would result in major increases in cost and schedule.



Waterfall Model Strengths:

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule

Waterfall Model Weaknesses:

- All requirements must be known upfront
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one big bang at the end
- Little opportunity for customer to preview the system (until it may be too late)

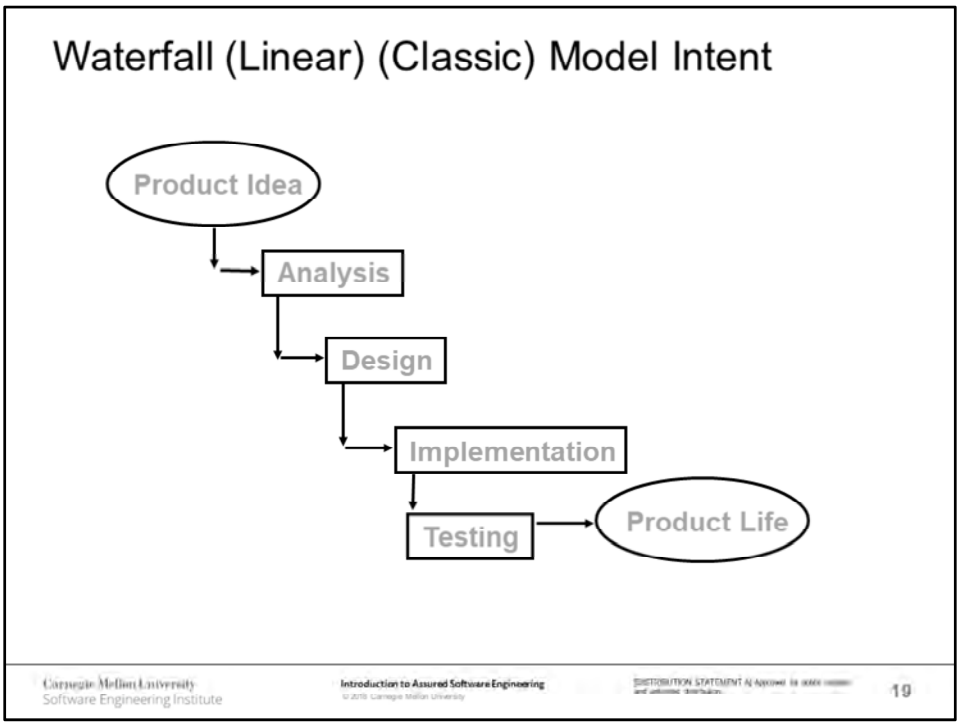
You may want to ask when would it be good to use a Waterfall Model?

When…

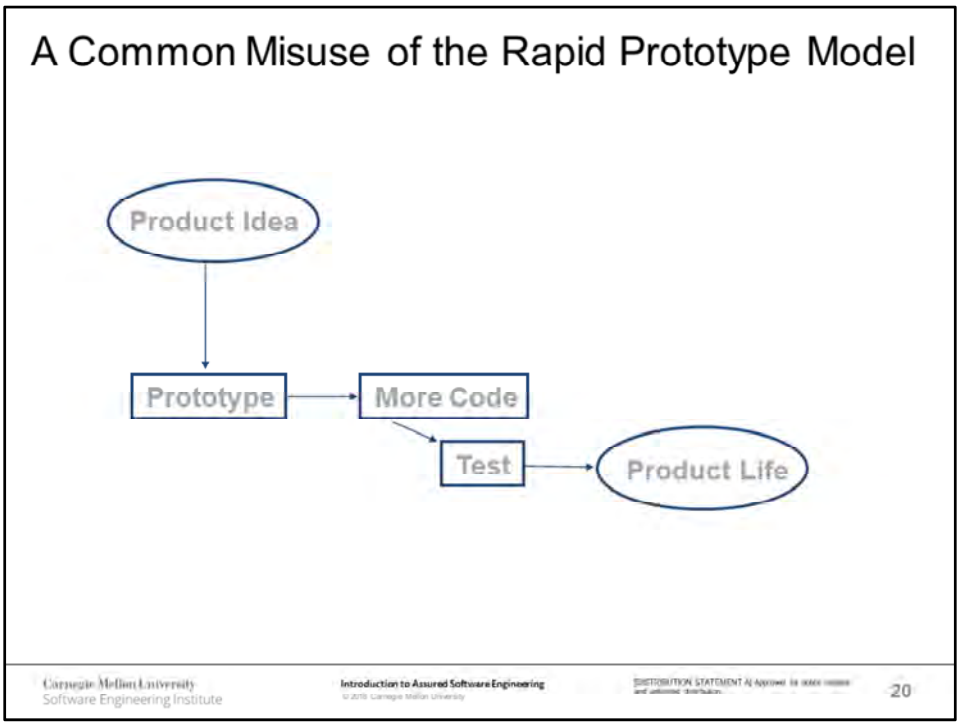
- Requirements are very well known
- Product definition is stable
- Technology is understood
- New version of an existing product
- Porting an existing product to a new platform

From the reading...

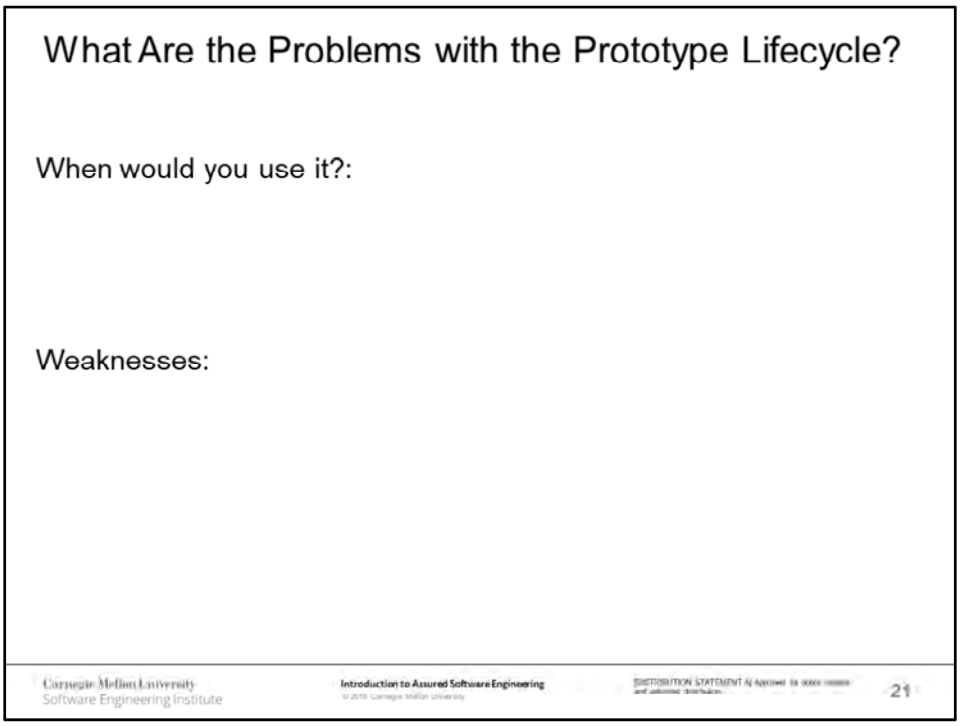
In the traditional waterfall model, each stage is a prerequisite for succeeding activities, making this method a risky choice for unprecedented systems because it inhibits flexibility. With a single pass through the process, integration problems usually surface too late. Also, a completed product is not available until the end of the process, discouraging user involvement. Taking these factors into account, *the other Lifecycle methods are recommended instead!*



This picture shows the phases of the classic Waterfall Model. Note that the arrows are sequential and go from one phase into the next and there are no feedback loops. One of the weaknesses of this model is that rework is usually needed and therefore development may be very costly when you use this lifecycle model.



Compare this to the Waterfall Model that we just discussed. What are the differences?

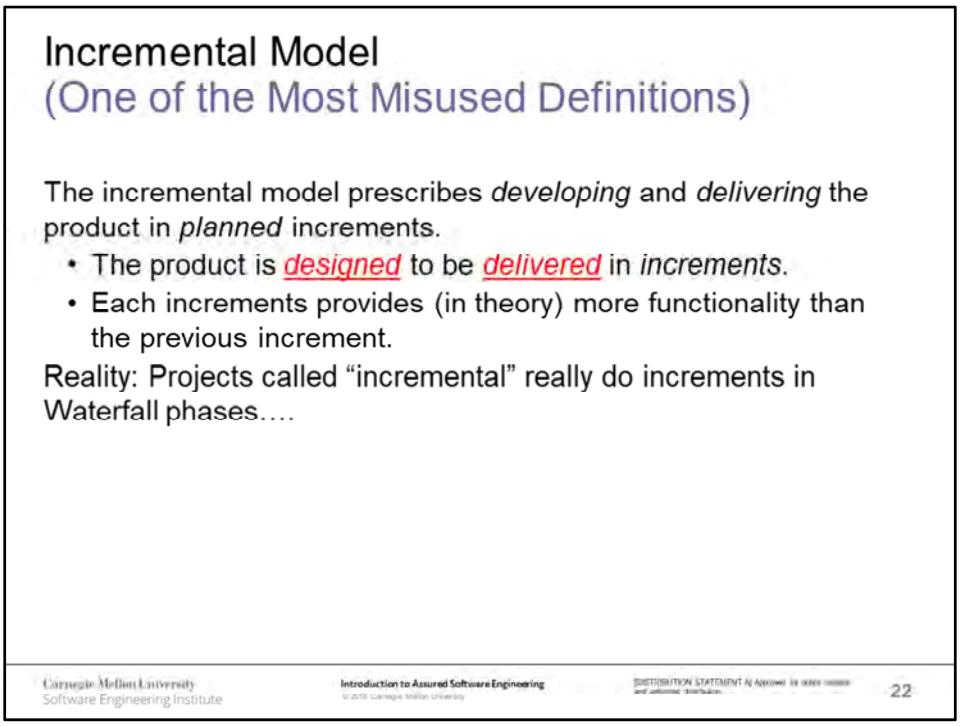


Prototypes are often used when the customer doesn't have a clear idea of what he/she wants and the requirements are general and ill-defined.

The advantages of prototyping is that it can improve the quality of requirements and specifications provided to developers. Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality.

Some of the weaknesses include insufficient analysis by the developers. The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model. The user can confuse the prototype and finished system. Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished.

As Watts Humphrey stated, since the users can only think in terms of the environment they know, the requirements for such systems are always stated in the current environment's terms. These requirements are thus necessarily incomplete, inaccurate, and misleading. The challenge for the system developer is to devise a development process that will discover, define, and develop to real requirements. This can only be done with intimate user involvement, and often with periodic prototype or early version field tests. Such processes always appear to take longer but invariably end up with a better system much sooner than with any other strategy."



Construct a partial implementation of a total system then slowly add increased functionality. The incremental model prioritizes requirements of the system and then implements them in groups. Each subsequent release of the system adds function to the previous release until all designed functionality has been implemented. The first increment is often a core product with many supplementary features. Users use it and evaluate it with more modifications to better meet the needs.

Strengths:

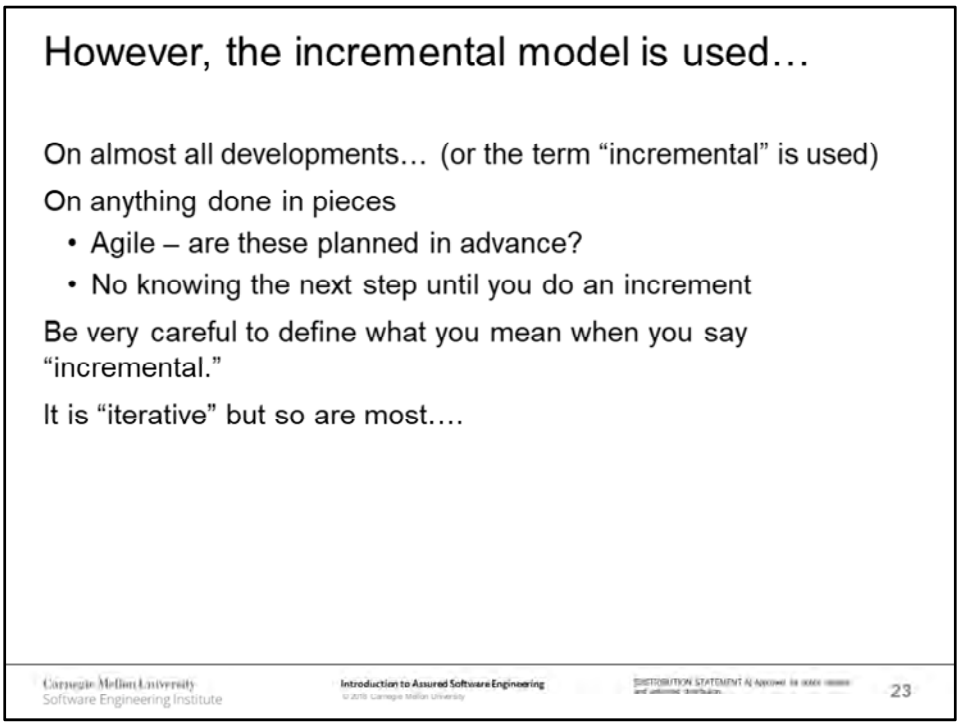
- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Uses "divide and conquer" breakdown of tasks
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced

Weaknesses:

- Requires good planning and design
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower

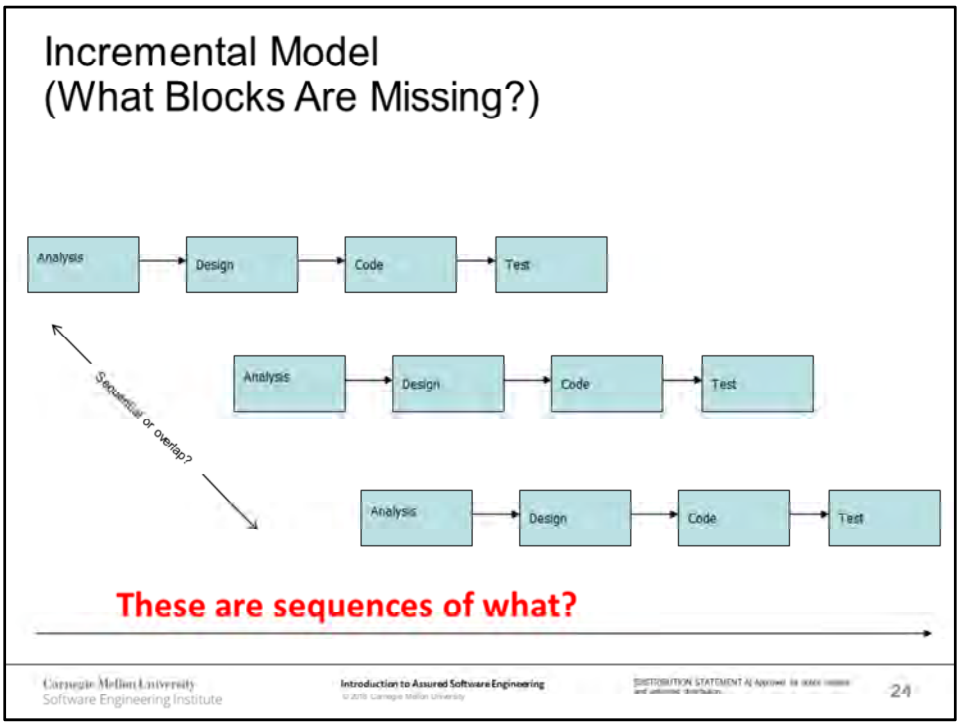
When to use it:

- Risk, funding, schedule, program complexity, or need for early realization of benefits.
- Most of the requirements are known up-front but are expected to evolve over time
- A need to get basic functionality to the market early
- On projects which have lengthy development schedules
- On a project with new technology
- A compelling need to expand a limited set of new functions to a later system release.



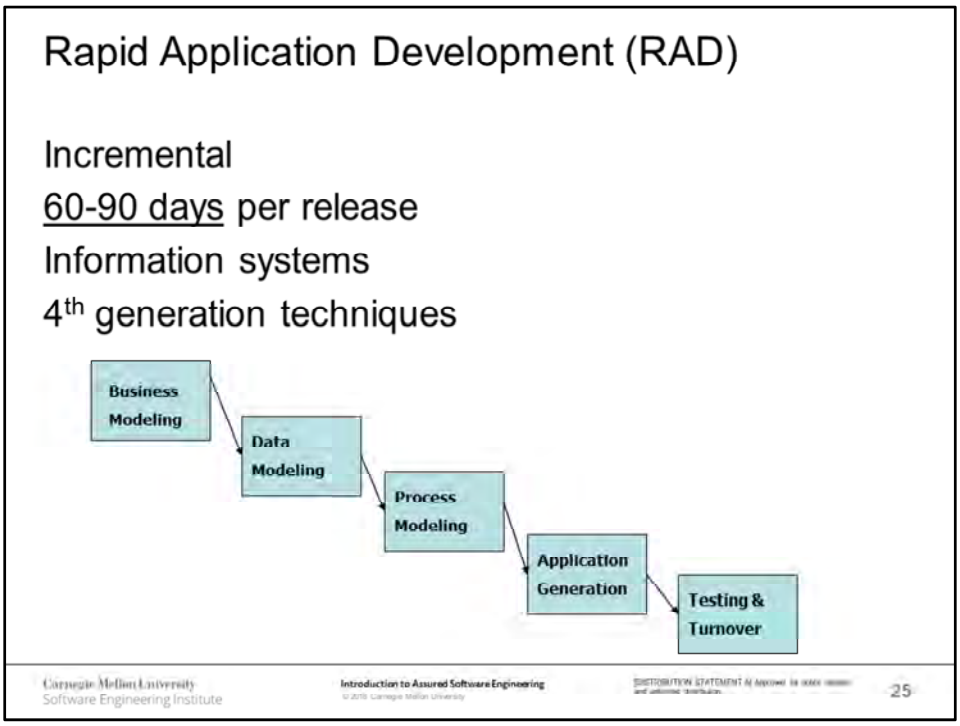
From the reading…
The incremental Lifecycle management method involves developing a software-intensive product in a series of increments of increasing functional capability. Benefits of the incremental method are:

- Risk is spread across several smaller increments instead of concentrating in one large development;
- Requirements are stabilized (through user buy-in) during the production of a given increment by deferring nonessential changes until later increments; and
- Understanding of the requirements for later increments becomes clearer based on the user's ability to gain a working knowledge of earlier increments.



The incremental Lifecycle method allows the user to employ part of the product and is characterized by a *build-a-little, test-a-little* approach to deliver an initial functional subset of the final capability. This subset is subsequently upgraded or augmented until the total scope of the stated user requirement is satisfied. The number, size, and phasing of incremental builds leading to program completion are defined in consultation with the user. An incremental methodology is most appropriate for low to medium-risk programs, when user requirements can be fully defined, or assessment of other considerations (e.g., risks, funding, schedule, size of program, early realization of benefits) indicate that a phased approach is the most prudent.

Compare this to the Waterfall Model. What is different?



Rapid application development (RAD) uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Considered an Agile method.

Strengths:

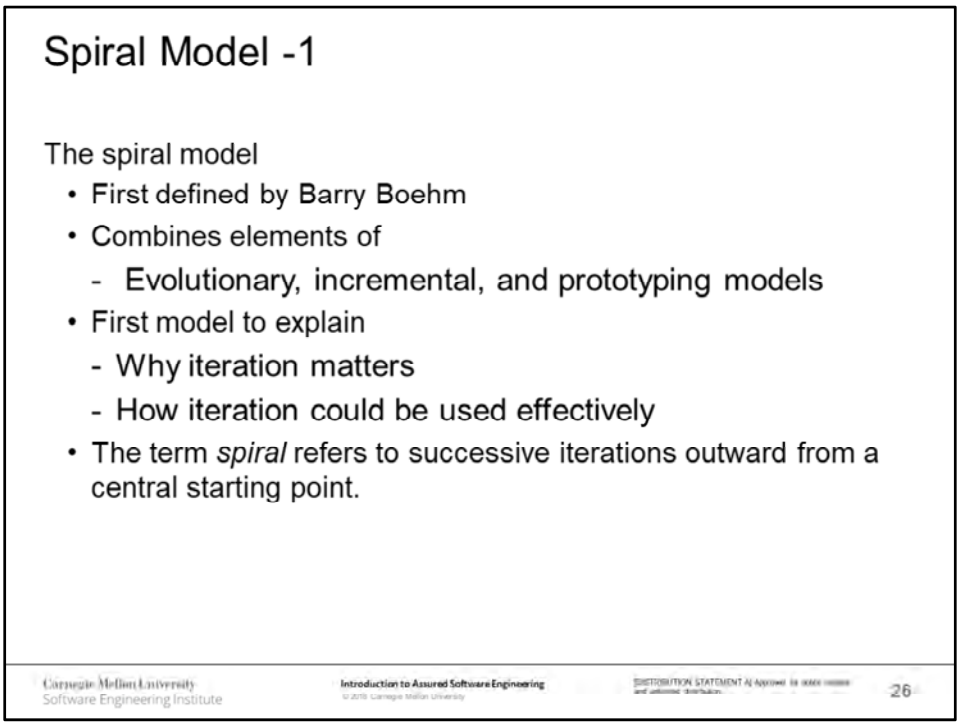
- Reduced cycle time and improved productivity with fewer people means lower costs
- Time-box approach mitigates cost and schedule risk
- Customer involved throughout the complete cycle minimizes risk of not achieving customer satisfaction and business needs
- Focus moves from documentation to code (WYSIWYG)
- Uses modeling concepts to capture information about business, data, and processes

Weaknesses:

- Accelerated development process must give quick responses to the user
- Risk of never achieving closure
- Hard to use with legacy systems
- Requires a system that can be modularized
- Developers and customers must be committed to rapid-fire activities in an abbreviated time frame

When to use RAD:

- Reasonably well-known requirements
- User involved throughout the Lifecycle
- Project can be time-boxed
- Functionality delivered in increments
- High performance not required
- Low technical risks
- System can be modularized

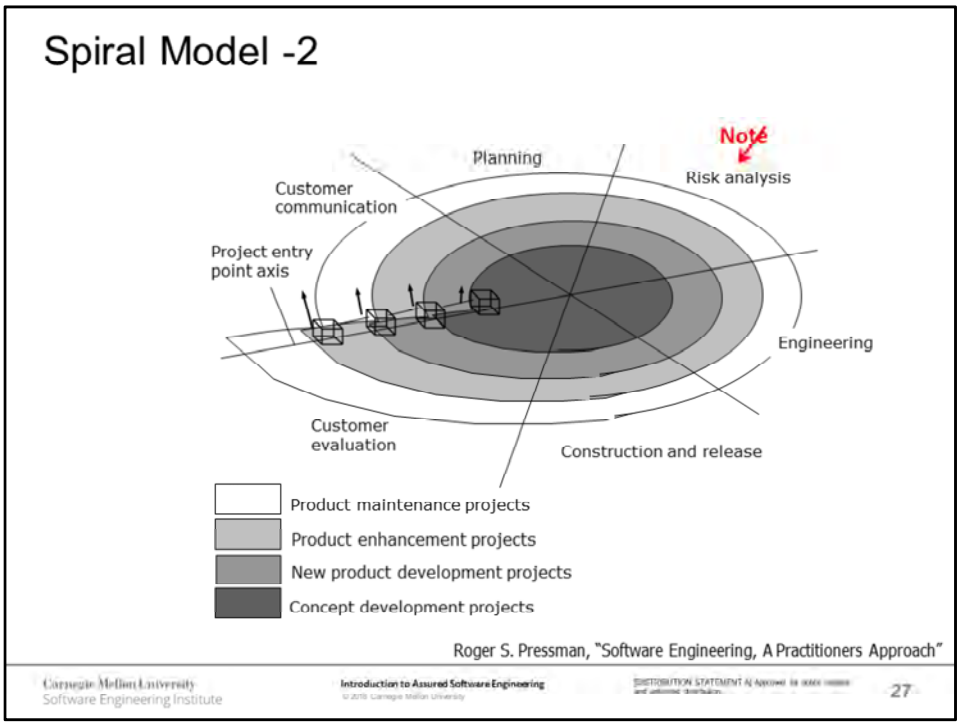


It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. Adds risk analysis, and 4 generation RAD prototyping to the waterfall model. Each cycle involves the same sequence of steps as the Waterfall Model.

From the reading…
The Spiral Model developed by Barry Boehm, provides a risk reducing approach to the software lifecycle. In the Spiral Model the radial distance is a measure of effort expended, while the angular distance represents progress.

It combines basic waterfall building block and evolutionary/incremental prototype approaches to software development.

The spiral model emphasizes the evaluation of alternatives and risk assessment. A review at the end of each phase ensures commitment to the next phase, or if necessary, identifies the need to rework a phase. The advantages of the spiral model are its emphasis on procedures, such as risk analysis, and its adaptability to different lifecycle approaches.



In the Spiral Model the radial distance is a measure of effort expended, while the angular distance represents progress.

The spiral model is divided into a number of framework activities (regions):

- customer communication
- planning (resources, timelines, etc.)
- risk analysis
- engineering
- construction and release
- customer evaluation

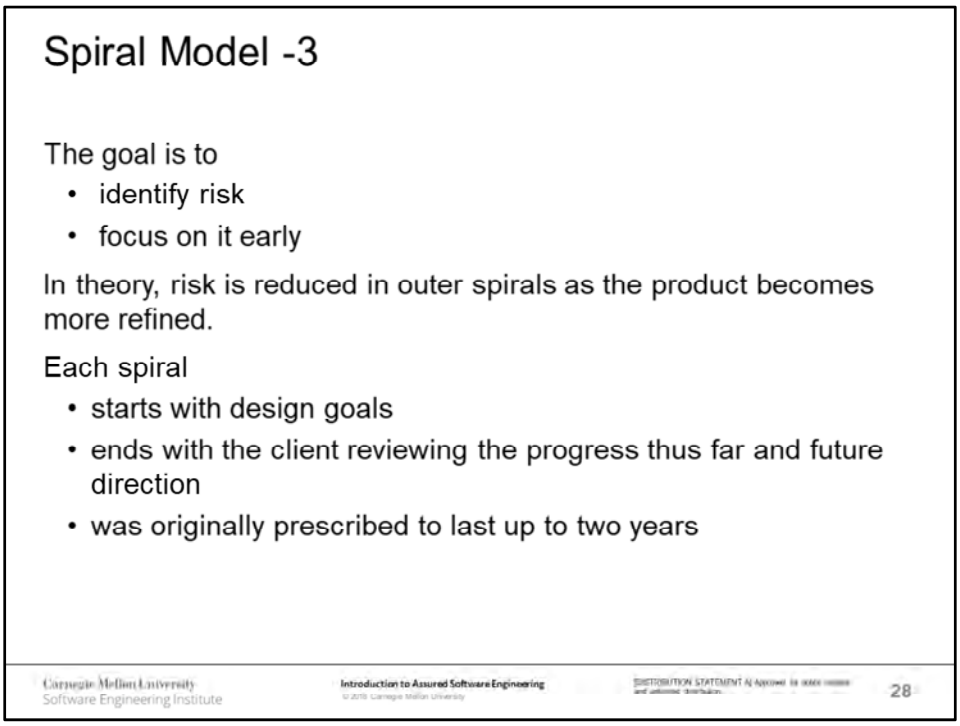
Each region is populated by a series of work tasks.

Strengths:

- Provides early indication of insurmountable risks, without much cost
- Users see the system early because of rapid prototyping tools
- Critical high-risk functions are developed first
- The design does not have to be perfect
- Users can be closely tied to all lifecycle steps
- Early and frequent feedback from users
- Cumulative costs assessed frequently

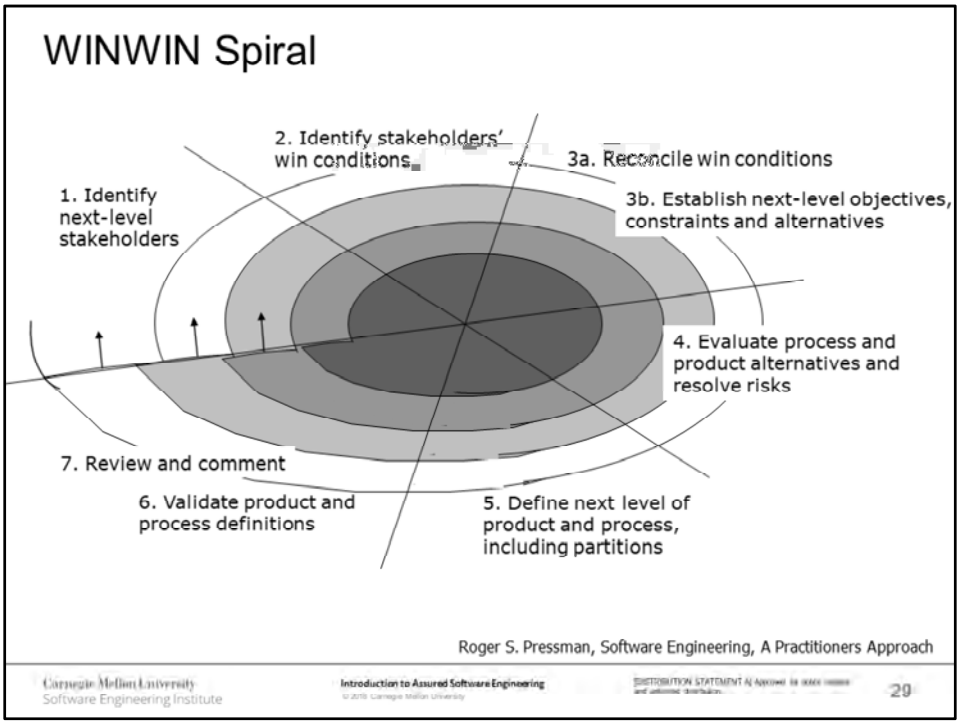
Weaknesses:

- Time spent for evaluating risks too large for small or low-risk projects
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- The model is complex
- Risk assessment expertise is required
- Spiral may continue indefinitely
- Developers must be reassigned during non-development phase activities
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration



When to use Spiral Model:

- When creation of a prototype is appropriate
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

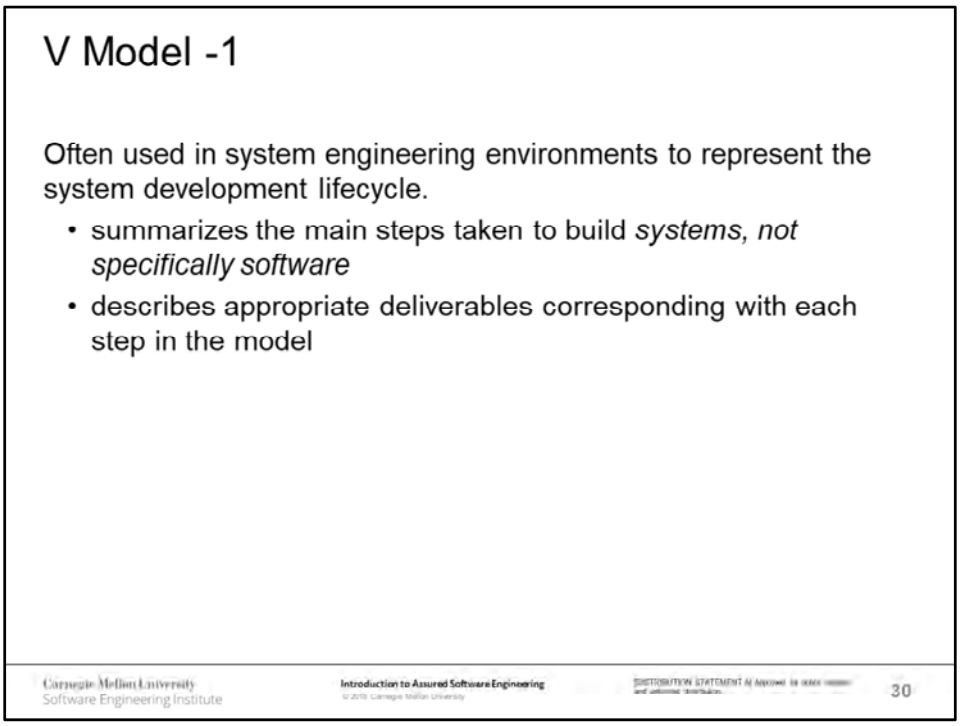


The spiral model suggests a framework activity that addresses customer communication. In reality, the customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market. The best negotiations strive for a "win-win" result. That is, the customer wins by getting the system or product that satisfies the majority of the customer's needs and the developer wins by working to realistic and achievable budgets and deadlines.

Boehm's WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following activities are defined:
1. Identification of the system or subsystem's key "stakeholders."

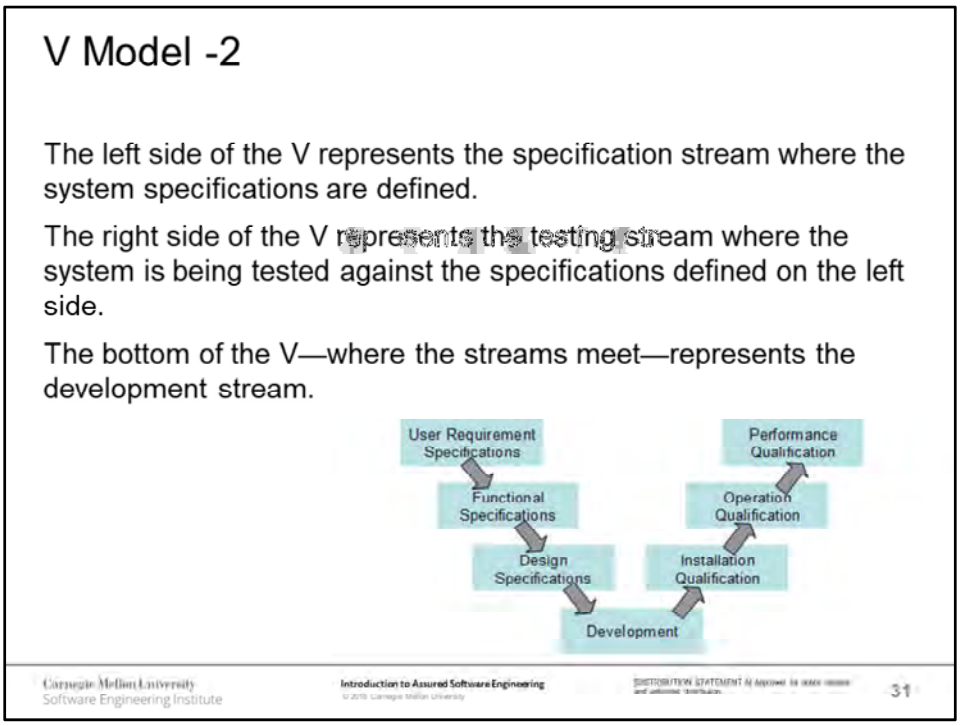
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team).
Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.
In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestones, called anchor points, that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.

In essence, the anchor points represent three different views of progress as the project traverses the spiral. The first anchor point, lifecycle objectives (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements. The second anchor point, lifecycle architecture (LCA), establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions. Initial operational capability (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software



A variant of the Waterfall that emphasizes the verification and validation of the product. Testing of the product is planned in parallel with a corresponding phase of development

The V model is a simple variant of the traditional Waterfall Model of system or software development. The V model builds on the waterfall model by emphasizing verification and validation. The V model takes the bottom half of the waterfall model and bends it upward into the form of a V, so that the activities on the right verify or validate the work products of the activity on the left. More specifically, the left side of the V represents the analysis activities that decompose the users' needs into small, manageable pieces, while the right side of the V shows the corresponding synthesis activities that aggregate (and test) these pieces into a system that meets the users' needs.
Donald Firesmith, SEI Blog, November 11, 2013



Like the waterfall model, the V model has both advantages and disadvantages. On the positive side, it clearly represents the primary engineering activities in a logical flow that is easily understandable and balances development activities with their corresponding testing activities. On the other hand, the V model is a gross oversimplification in which these activities are illustrated as sequential phases rather than activities that typically occur incrementally, iteratively, and concurrently, especially on projects using evolutionary (agile) development approaches. *Donald Firesmith, SEI Blog, November 11, 2013*

Strengths:

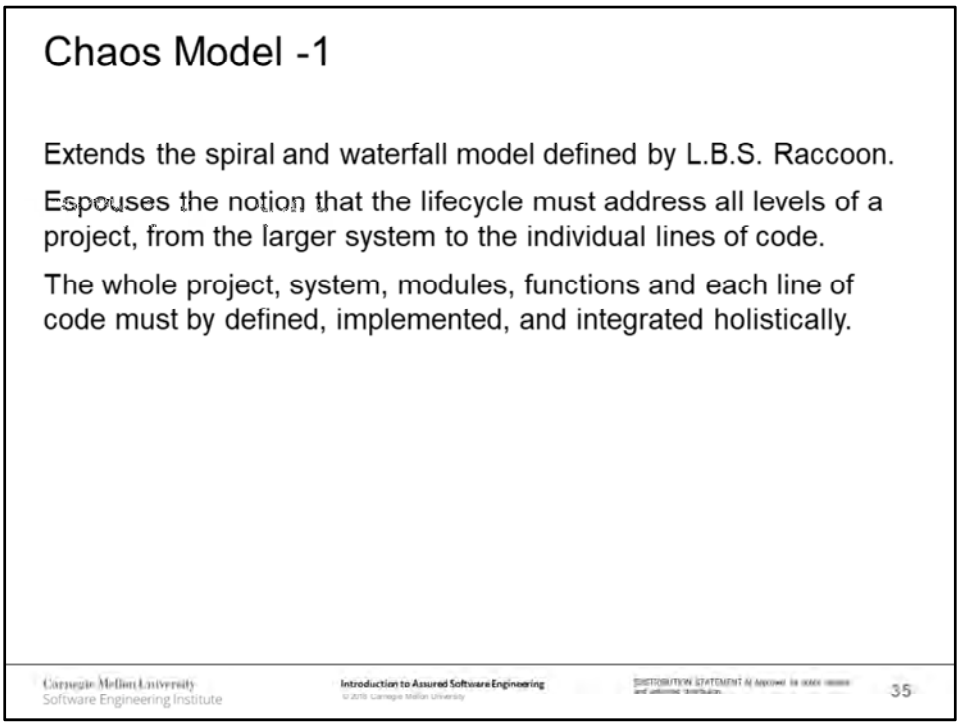
- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use

Weaknesses:

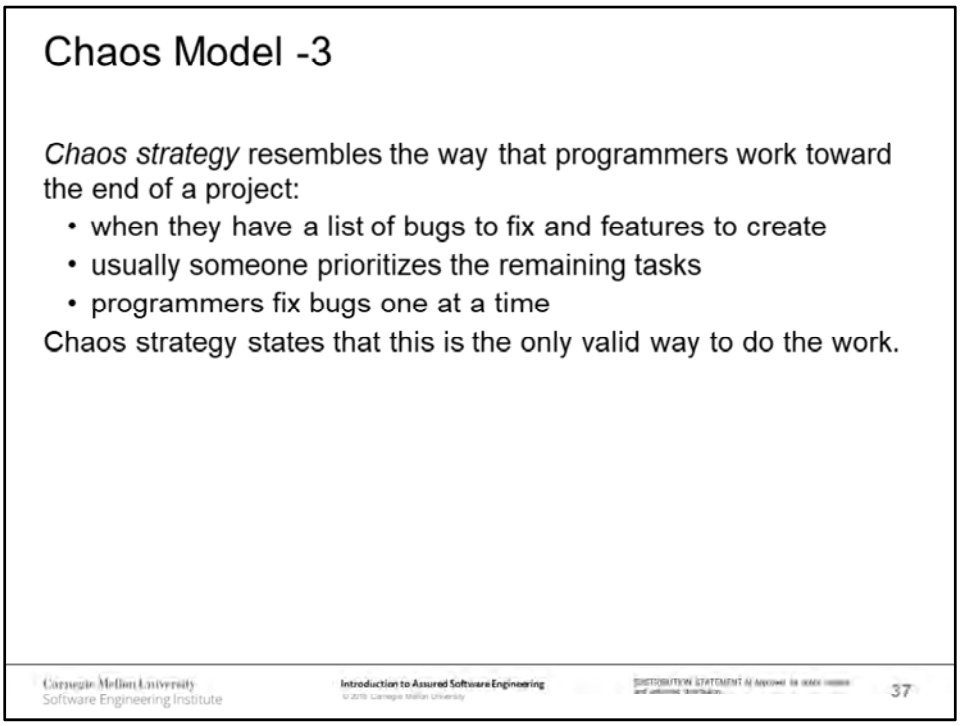
- Does not easily handle concurrent events
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis activities

When to use the V Model

- For systems requiring high reliability – hospital patient control applications
- When all requirements are known up-front
- When it can be modified to handle changing requirements beyond analysis phase
- When solution and technology are known



L.B.S. Raccoon, noted that project management models such as the spiral model and waterfall model, while good at managing schedules and staff, didn't provide methods to fix bugs or solve other technical problems. At the same time, programming methodologies, while effective at fixing bugs and solving technical problems, do not help in managing deadlines or responding to customer requests. The structure attempts to bridge this gap. Chaos theory was used as a tool to help understand these issues.
Wikipedia



The chaos model notes that the phases of the lifecycle apply to all levels of projects, from the whole project to individual lines of code.

- The whole project must be defined, implemented, and integrated.
- Systems must be defined, implemented, and integrated.
- Modules must be defined, implemented, and integrated.
- Functions must be defined, implemented, and integrated.
- Lines of code are defined, implemented and integrated.

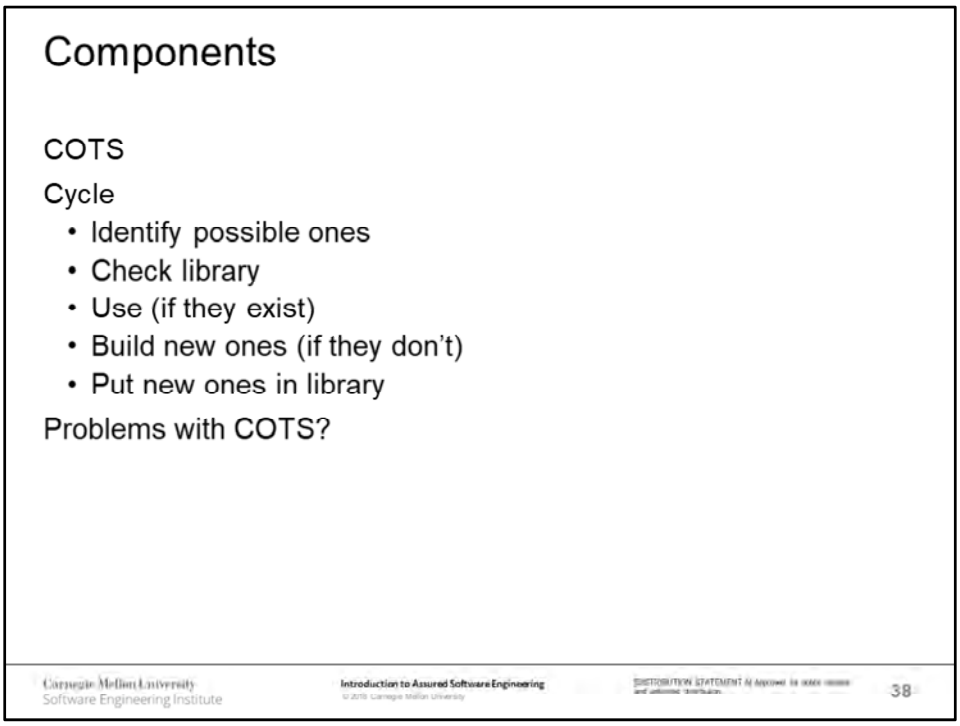
One important change in perspective is whether projects can be thought of as whole units, or must be thought of in pieces. Nobody writes tens of thousands of lines of code in one sitting. They write small pieces, one line at a time, verifying that the small pieces work. Then they build up from there. The behavior of a complex system emerges from the combined behavior of the smaller building blocks.

The chaos strategy is a strategy of software development based on the chaos model. The main rule is always resolve the most important issue first. An *issue* is an incomplete programming task. The most important issue is a combination of big, urgent, and robust.

- *Big* issues provide value to users as working functionality.
- *Urgent* issues are timely in that they would otherwise hold up other work.
- *Robust* issues are trusted and tested. Developers can then safely focus their attention elsewhere.

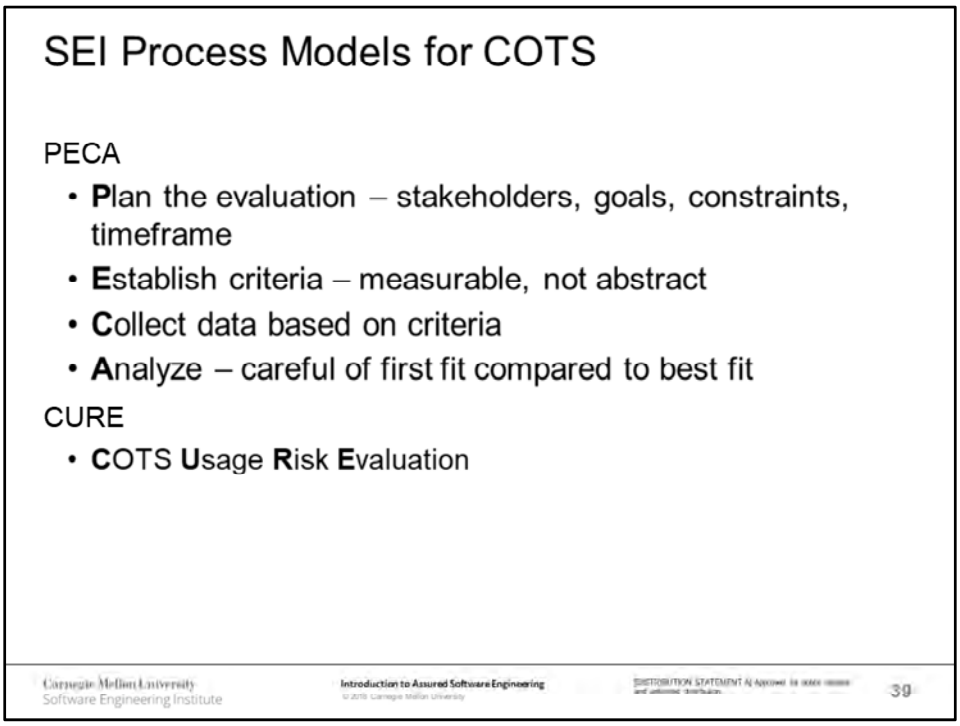
To resolve means to bring it to a point of stability. The chaos strategy resembles the way that programmers work toward the end of a project, when they have a list of bugs to fix and features to create. Usually someone prioritizes the remaining tasks, and the programmers fix them one at a time. The chaos strategy states that this is the only valid way to do the work.

Wikipedia



The SEI defines COTS product as one that is:

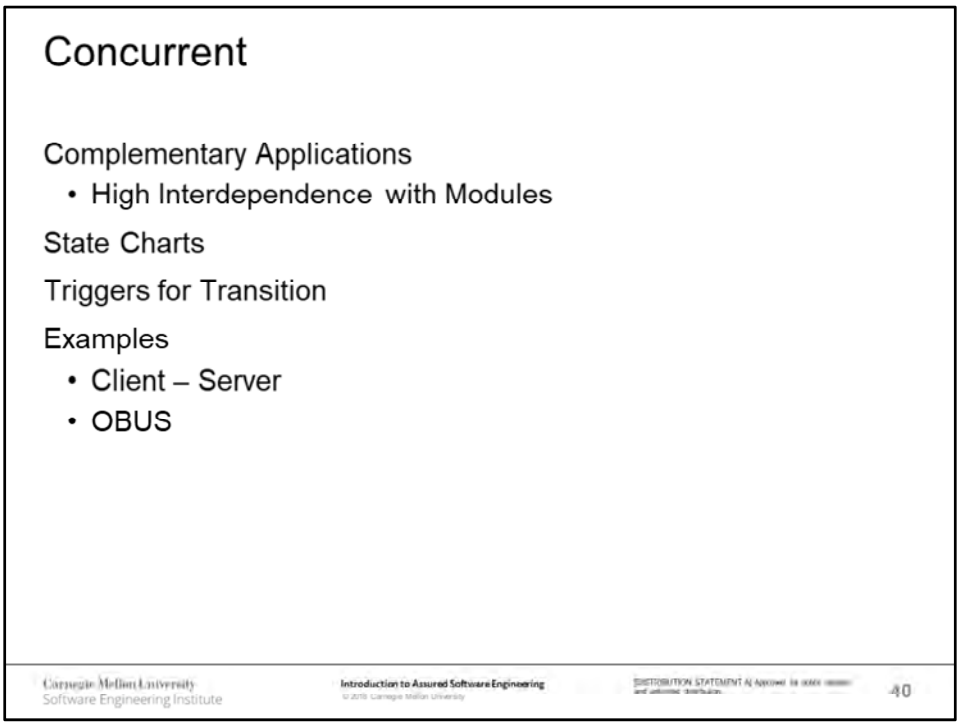
- sold, leased, or licensed to the general public
- offered by a vendor trying to profit from it
- supported and evolved by the vendor, who retains the intellectual property rights
- available in multiple, identical copies
- used without modification of the internals



An evaluation process defined by the SEI and National Research Council Canada (NRC), called PECA (Plan, Establish, Collect, Analyze), helps organizations make carefully reasoned and sound product decisions. The process can be tailored by each organization to fit its particular needs, and is flexible enough to be used within many organizations and with many COTS-based development processes.

Although the PECA process was derived in part from ISO 14598, the process was freely adapted to fit the needs of COTS software product evaluation. The process begins with initial planning for an evaluation of a COTS product (or products) and concludes with a recommendation to the decision maker. The decision itself is not considered part of the evaluation process—the aim of the process is to provide all of the information necessary for a decision to be made.

The COTS Usage Risk Evaluation (CURE) has been developed to assist organizations in avoiding common mistakes in COTS-based acquisitions. CURE is ideally given during the early stages of a program, when the major key decisions relating to use of COTS products have not yet been made. CURE is a useful technology for any organization that is preparing for a project that is critically dependent on commercial software; it provides insight and understanding into the potential risks associated with such a program.



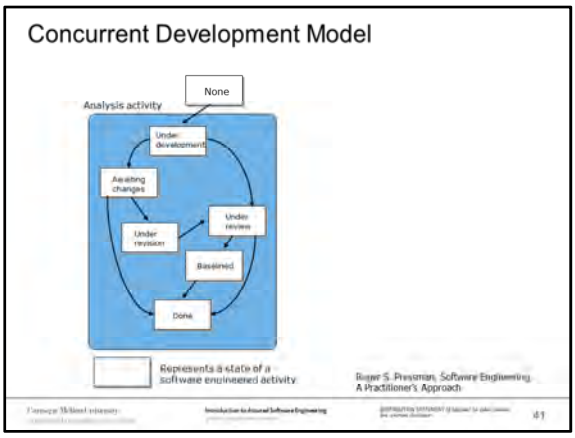
The concurrent development model, called concurrent engineering, provides an accurate state of the current state of a project.

Focus on concurrent engineering activities in a software engineering process such as prototyping, analysis modeling, requirements specification and design. Represented schematically as a series of major technical activities, tasks, and their associated states. Defined as a series of events that trigger transitions from state to state for each of the software engineering activities.

Two ways to achieve the concurrency:

- system and component activities occur simultaneously and can be modeling using the state-oriented approach
- a typical client/server application is implemented with many components, each can be designed and realized concurrently.

Applies to all types of software development.

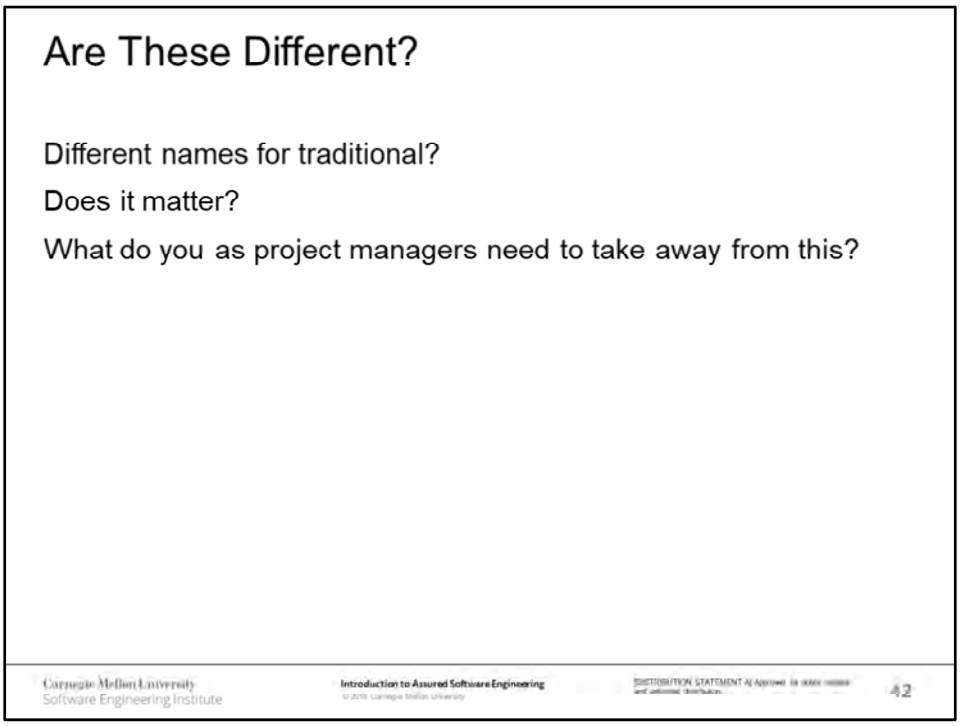


The concurrent development model, sometimes called concurrent engineering, has been described in the following manner by Davis and Sitaram :
Project managers who track project status in terms of the major phases [of the classic lifecycle] have no idea of the status of their projects. These are examples of trying to track extremely complex sets of activities using overly simple models. Note that although . . . [a large] project is in the coding phase, there are personnel on the project involved in activities typically associated with many phases of development simultaneously. For example, . . personnel are writing requirements, designing, coding, testing, and integration testing [all at the same time]. Software engineering process models by Humphrey and Kellner (SEI) have shown the concurrency that exists for activities occurring during any one phase. Kellner's work uses statecharts [a notation that represents the states of a process] to represent the concurrent relationship existent among activities associated with a specific event (e.g., a requirements change during late development), but fails to capture the richness of concurrency that exists across all software development and management activities in the project. . . . Most software development process models are driven by time; the later it is, the later in the development process you are. [A concurrent process model] is driven by user needs, management decisions, and review results.
The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For example, the engineering activity defined for the spiral model is accomplished by invoking the following tasks: prototyping and/or analysis modeling, requirements specification, and design.

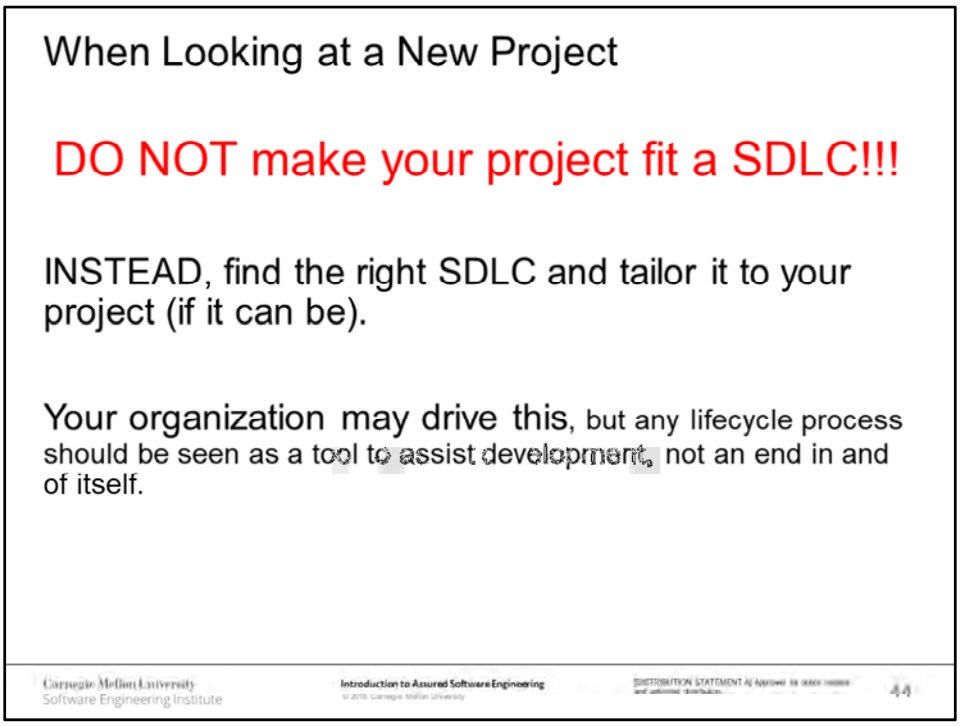
The activity—analysis—may be in any one of the states noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the customer communication activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state. The analysis activity (which existed in the none state while initial customer communication was completed) now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the analysis activity moves from the under development state into the awaiting changes state.
The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example, during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event analysis model correction which will trigger the analysis activity from the done state into the awaiting changes state.
The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions : a system dimension and a component dimension. System level issues are addressed using three activities: design, assembly, and use. The component dimension is addressed with two activities: design and realization.
Concurrency is achieved in two ways:

(1) system and component activities occur simultaneously and can be modeled using the state-oriented approach described previously;

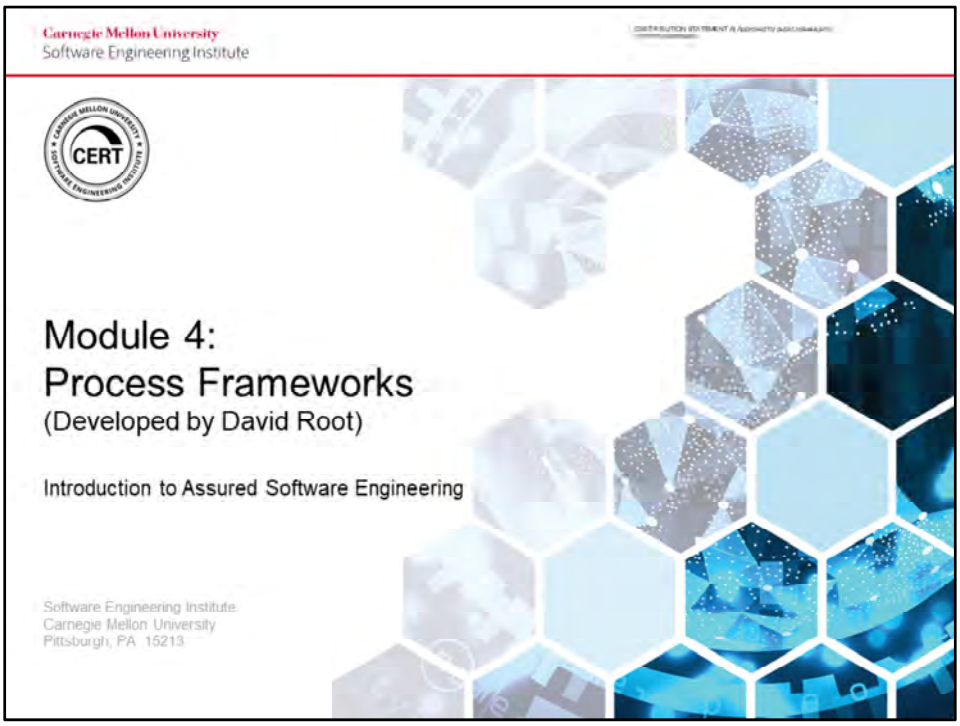
(2) a typical client/server application is implemented with many components, each of which can be designed and realized concurrently.
In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities to a sequence of events, it defines a network of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the states of an activity.

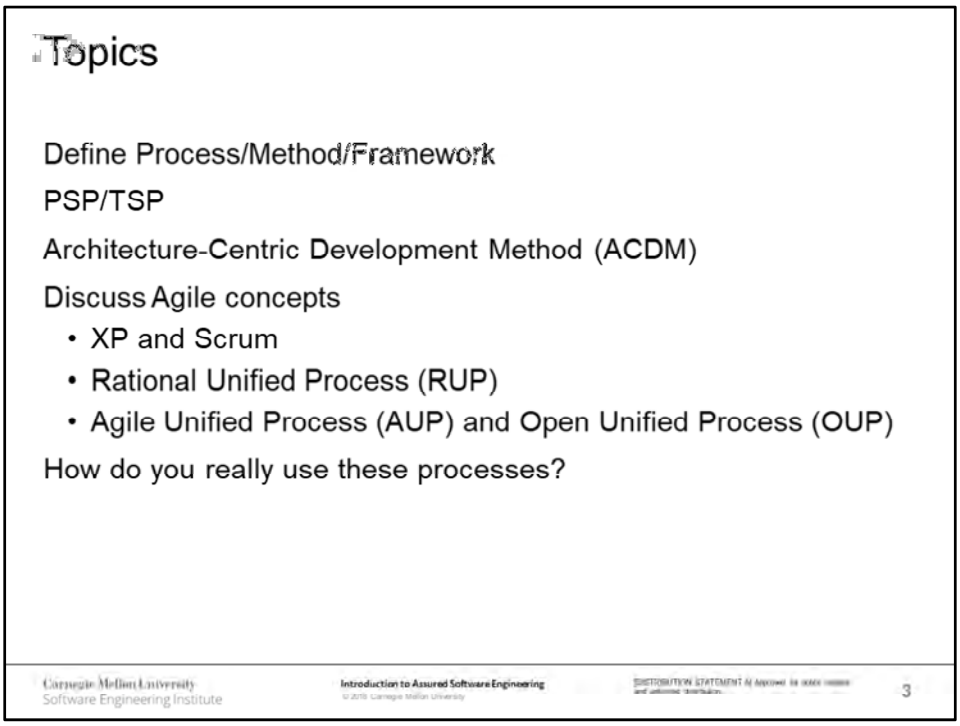


Are these SDLCs different? Use the questions on the slide to discuss with students.



Any one model does not fit all projects. If there is nothing that fits a particular project, pick a model that comes close and modify it for your needs. Project should consider risk but if complete spiral is too much – start with spiral and modify it. Project delivered in increments but there are serious reliability issues – combine incremental model with the V-shaped model. Each team must pick or customize a SDLC model to fit the project.

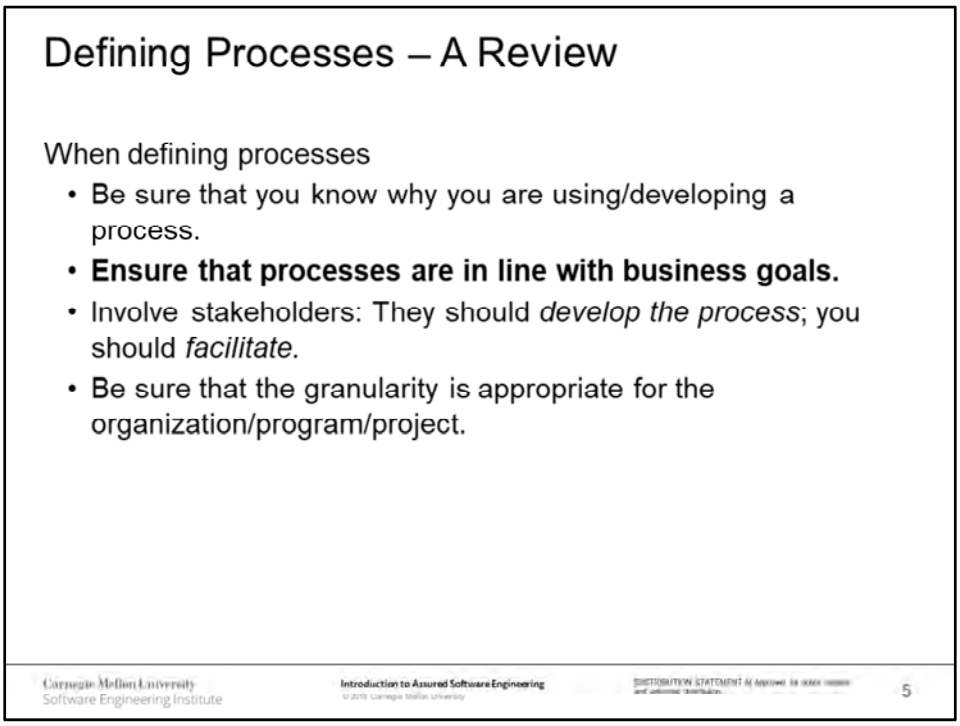

Carnegie Mellon University
Software Engineering Institute
CERT
Module 4:
Process Frameworks
(Developed by David Root)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



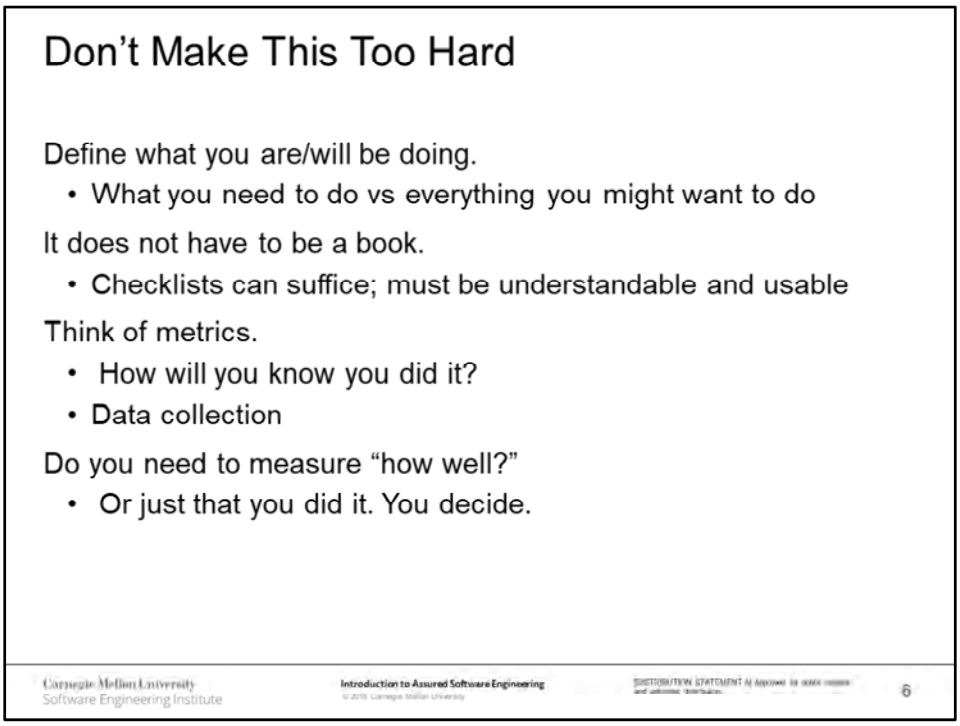
List the learning objectives for this particular session.

The student will learn

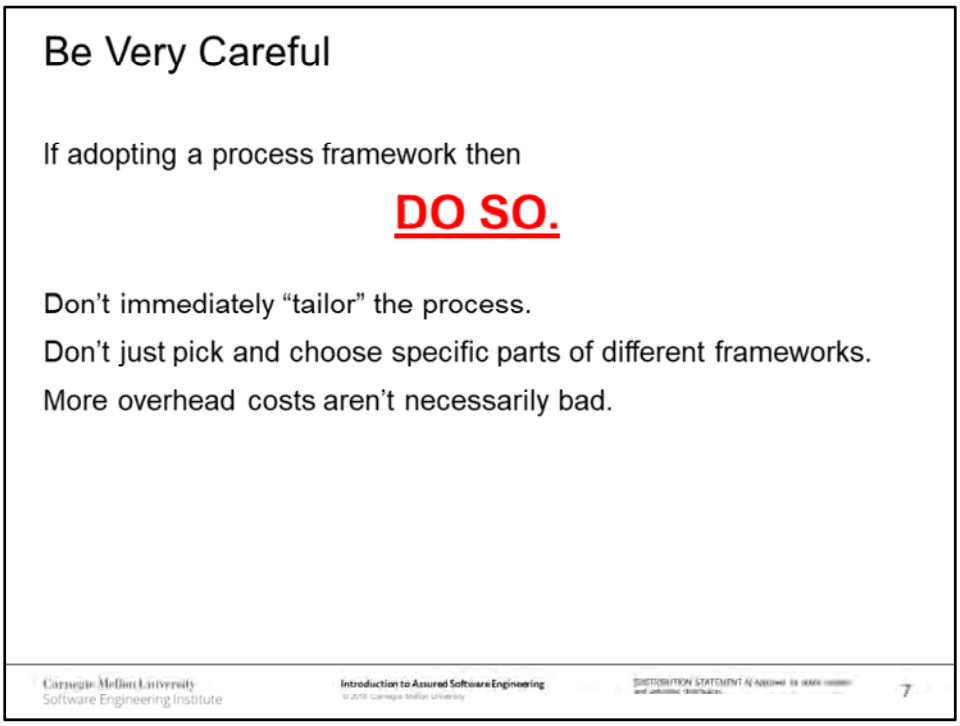
- the difference between process, method, and framework
- the concepts of PSP/TSP/SCDM
- the fundamentals of Agile methods



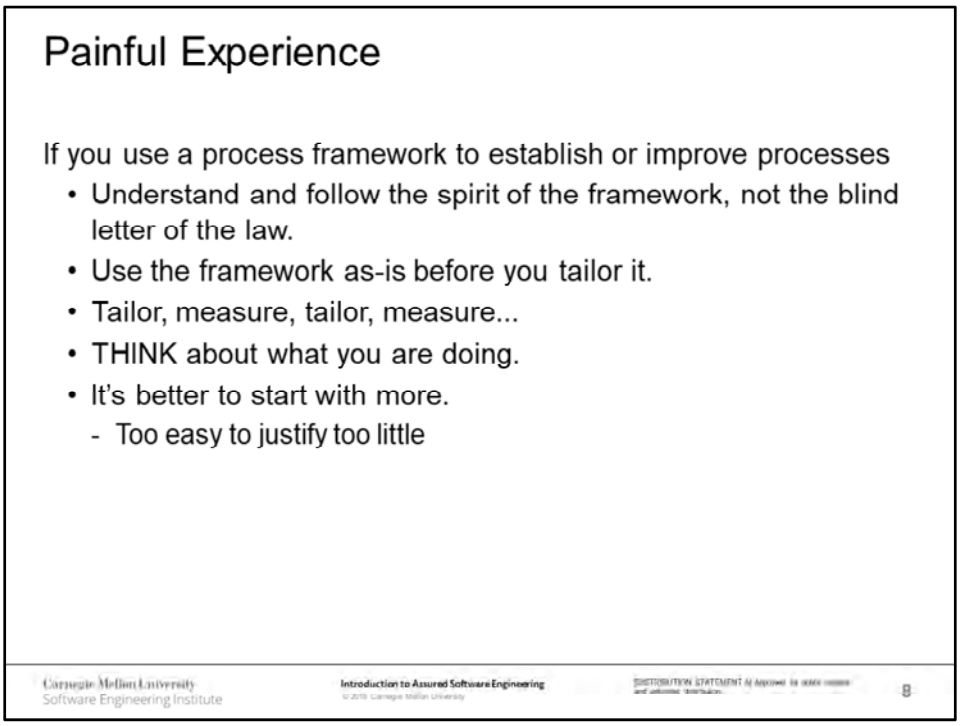
Process was defined in the previous lecture.



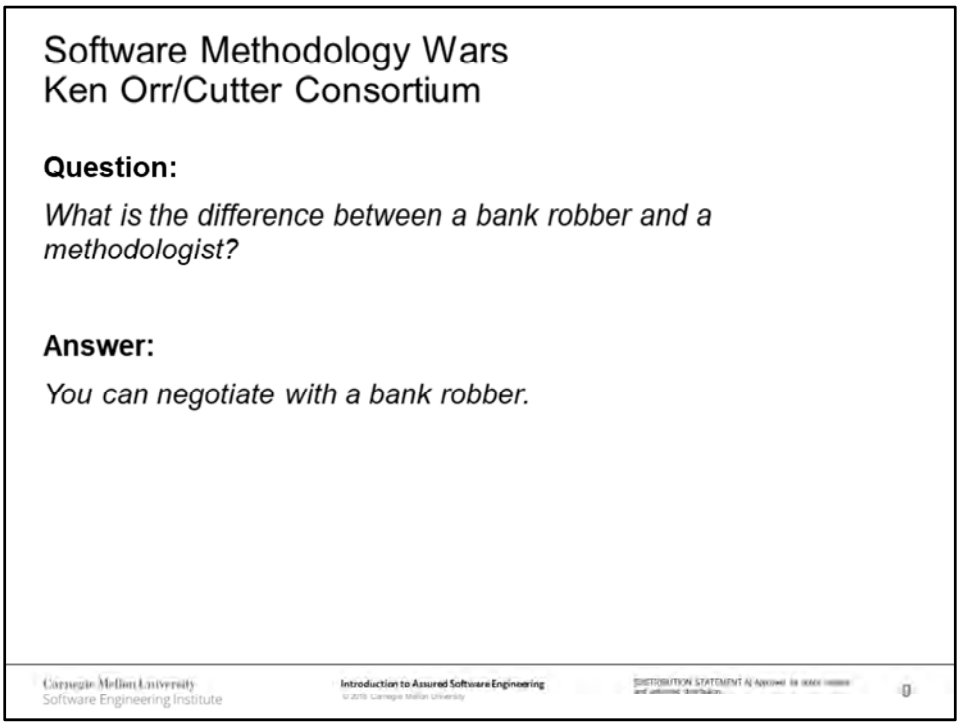
When you define processes try to keep them simple. They should contain enough information that you understand the main activities but at the same time processes should be usable by multiple projects in multiple environments with tailoring as appropriate.



Most process frameworks will require some adaptation (tailoring) to your environment. It is advisable to first run through the process to understand what needs to be tailored. Process tailoring should be based on criteria and not be a pick and choose approach.



If you are using a framework, understand the strengths and weaknesses. Use the framework "as is" before you try to tailor it. When you tailor your processes, measure the results and see if that is what is expected Make sure you don't tailor too much or your measures will not provide much insight. What you really will have is a different process.



Ken Orr is a software engineer, executive and consultant, known for his contributions in the field of software engineering to structured analysis and with the Warnier/Orr diagram.

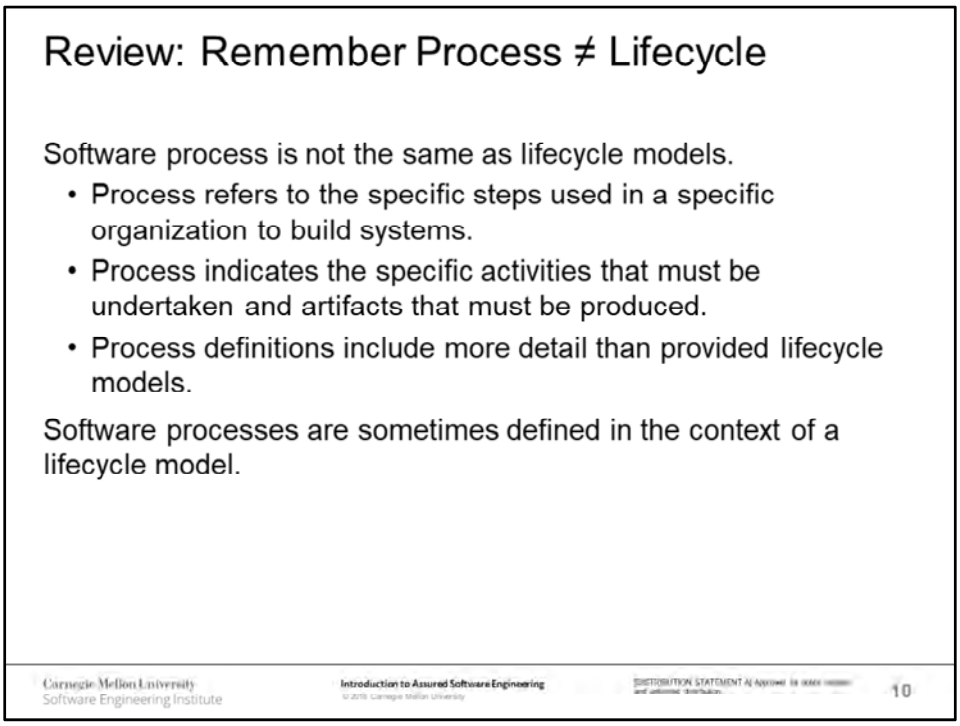
He wrote the article "CMM versus agile development: Religious Wars and Software Development." *Agile Project Management Executive Report* 3.7 (2002). The article starts out with the following quote.

"Today, a new debate rages: agile software development versus rigorous software development."

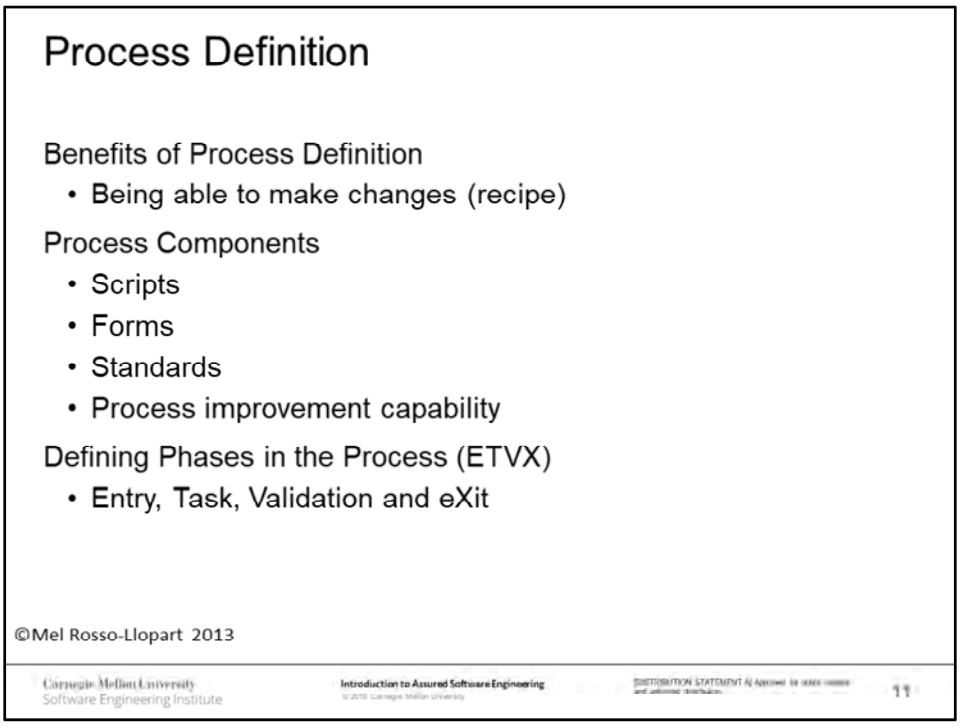
-- Jim Highsmith, Fellow, Cutter Business Technology Council

An excerpt from the article…

"Every decade or so, there seems to be yet another software development methodology struggle. During the 1970s, the battle was between various forms of structured and traditional development; in the 1980s, the struggle was between various forms of data modeling and traditional development; during the 1990s, it was between warring factions of object-oriented (OO) design and traditional development. Like most religious wars, the most intense conflict of methodology wars has been between either closely related methods or between those that are furthest apart. Today, in the first decade of the 21st century, the current software development war is between those supporting agile methods and those supporting CMM -- a representative of the traditional "waterfall" software development approaches."



This slide is from the previous lecture. Point out that life-cycle models define the phases that determine the sequence of major activities. Within the phases, processes define the series of steps that will be done within that phase.



The concept of processes is at the heart of software and systems engineering. Software process models integrate software engineering methods and techniques and are the basis for managing projects. High product quality routinely results from high process quality. The quality of products is dependent on the QUALITY of the PROCESSES used to produce them. Focus on process helps people work “smarter” rather than “harder”, to use technology effectively and to achieve better outcomes

The benefits of defined processes include:

- Improves communication and understanding of current practices
- Enables capture of “corporate know-how” and best practices
- Establishes a baseline for analysis and improvement of the process
- Identifies where and how to measure the process
- Facilitates common training and understanding; facilitates teamwork, where appropriate

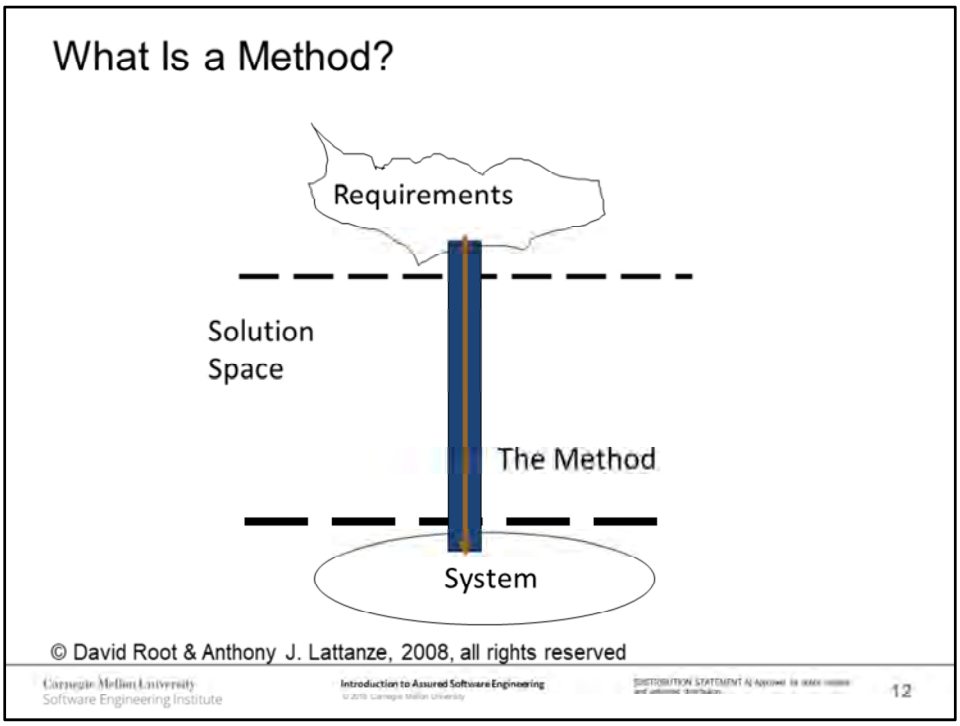
One of the core aspects of defining a process is using historical data to analyze and improve process performance. Therefore, data collection is supported by four main elements:

- Scripts – document the process entry criteria, phases/steps, and exit criteria. The purpose is to guide you through the process.
- Measures – measure the process and the product. They provide insight into how the process is working and the status of the work.
- Standards – provide consistent definitions that guide the work and for gathering the data.
- Forms – provide a convenient and consistent framework for gathering and retaining data.

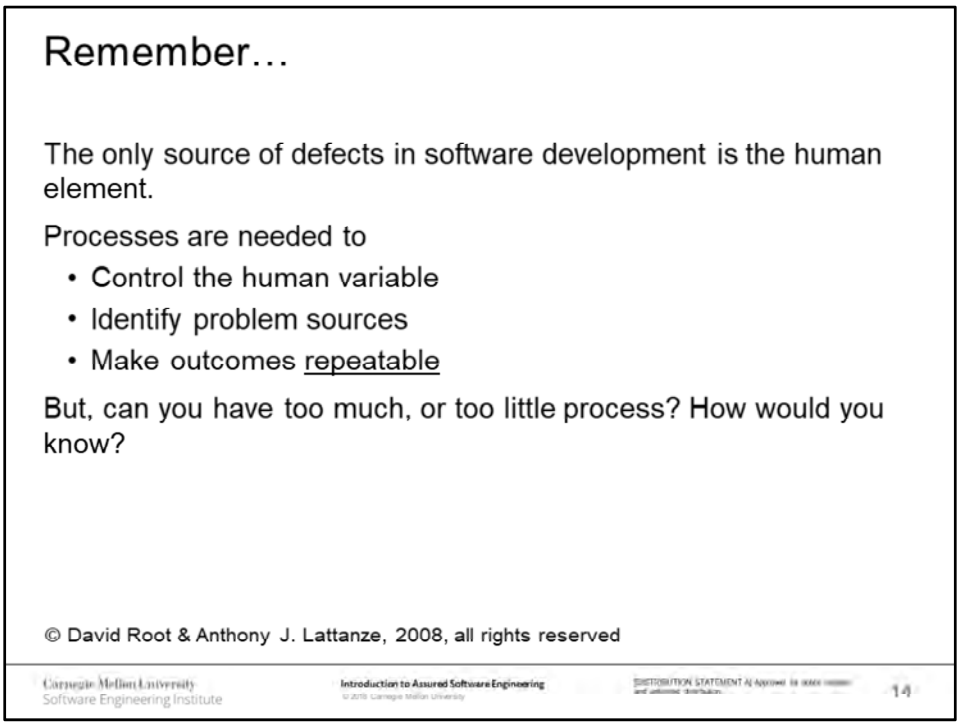
By managing your process you are able to then improve it.

There are many ways to describe process components. One way is ETVX.

ETVX describes activities as a set of tasks to be performed. It uses a simple technique of describing four phases Entry Criteria, Tasks to be performed, Exit Criteria and Validation, Verification Conditions for each task. The purpose of using a process definition method such as ETVX is to develop an operational definition of the activity under a process architecture. These operational definitions can then be adapted or tailored for use on all projects which uses the defined process architecture.



The method is the way or approach that you are going to use to build the system. It will be used to structure, plan and control your activities.



Too little process?

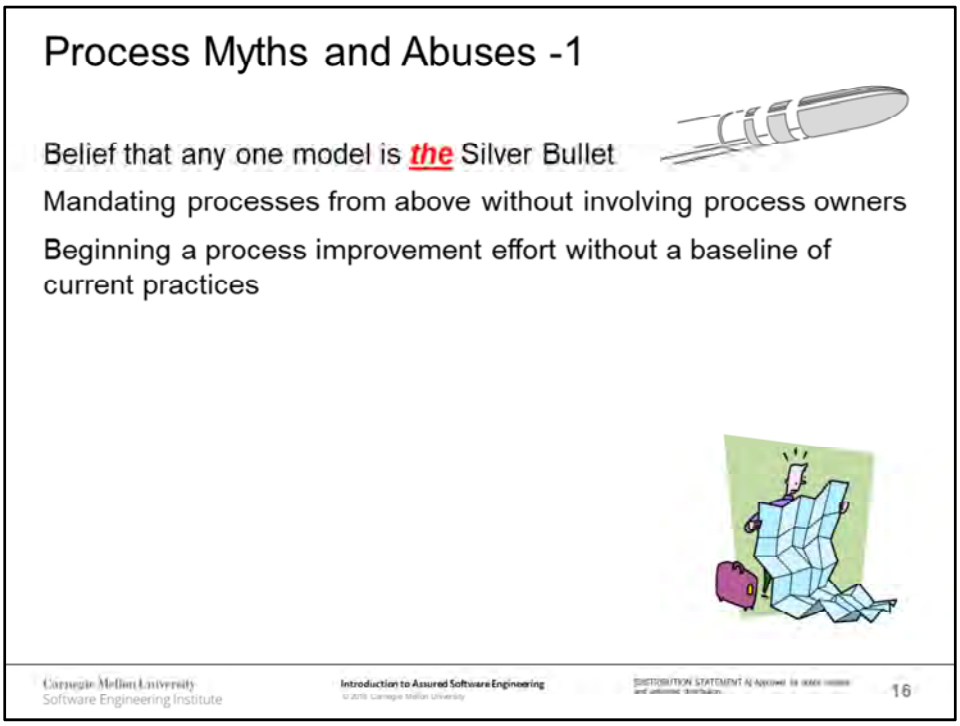
- Everyone can do what they want.

Too much process?

- There are so many rules and guidelines that the developer feels constrained by what they do and do feel that they have the flexibility to think and do a good job.

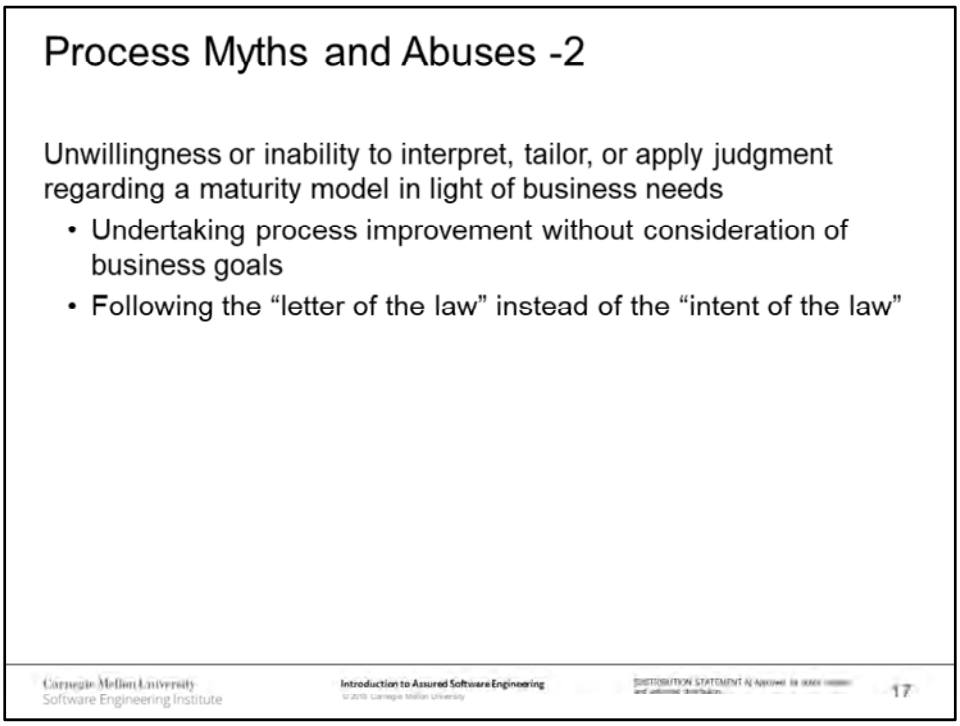
How would you know?

- ASK
- Observe
- Metrics
- Quality of the results



Processes are not a silver bullet. They are frameworks that allow an organization to provide guidance for key activities. It is best to have buy-in and involve the people that will be using the process.

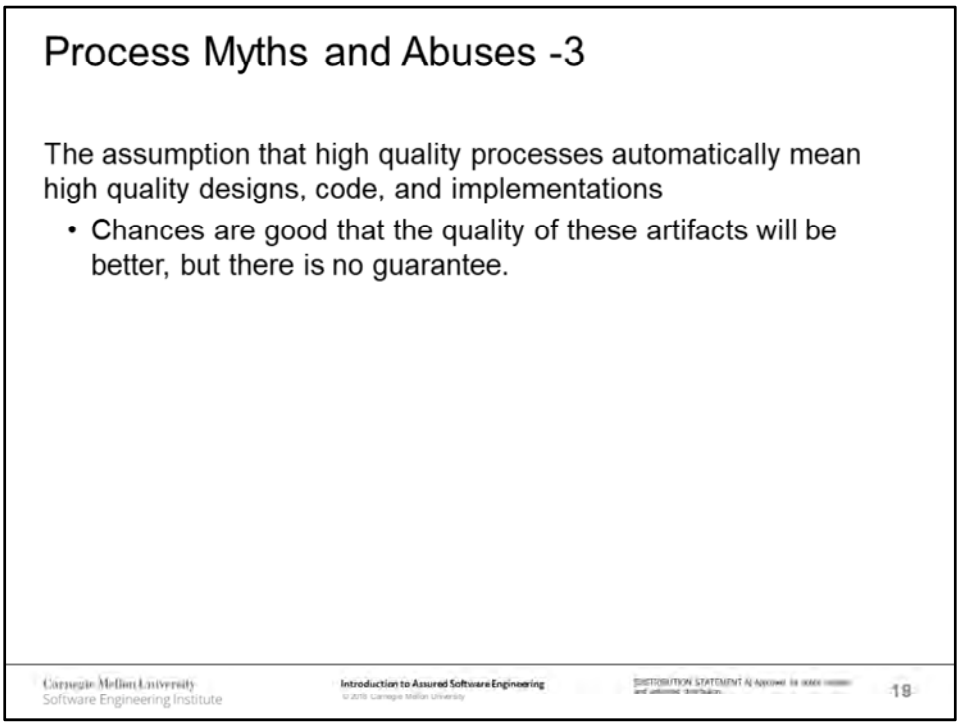
Watt's Humphrey had a quote, "If you don't know where you are, a map won't help." If you assume you know the current practice without doing some type of evaluation to baseline the current practice, you may find yourself "lost in the woods" in your organization and changing things that are working and not having an accurate picture of what needs to be fixed.



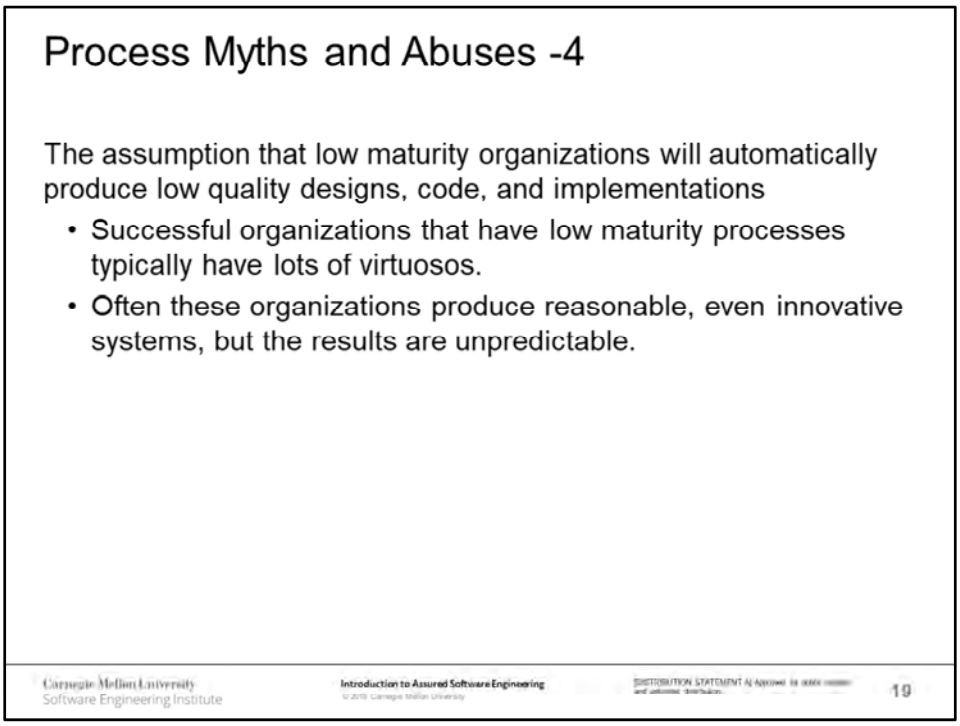
Process improvement occurs within the context of the organization.

- strategic plan
- business objectives
- organizational structure
- technologies in use
- culture
- management

To have a successful effort, you should tie it back to what is going on in other parts of your organization.



Everyone realizes the importance of having a motivated, quality work fore but even our finest people can't perform at their best when the process is not understood or operating at its best.



Heroes are made by people's actions. Many Ad Hoc "Hobbysts" can come in and save the day, but can this be sustained overtime? Can that hero be available to work on all projects?

Process Myths and Abuses -5

High-maturity-level organizations are guaranteed to enjoy high profitability.

- Royal Enfield example…improved 1950's design

High maturity can only be achieved through high "ritualization."

- Red Bead experiment

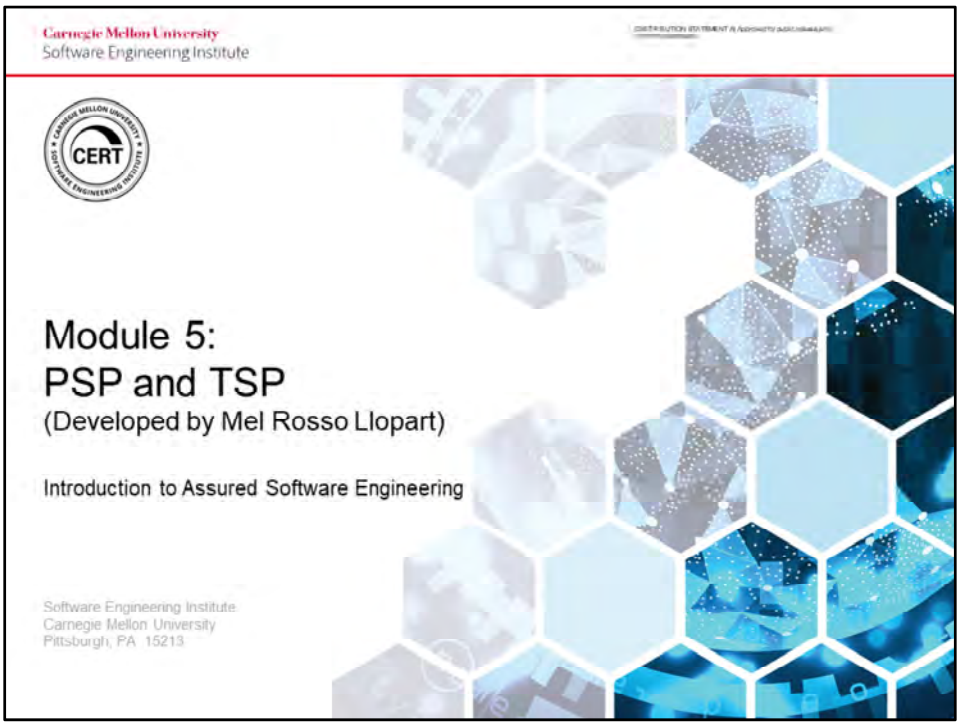
Carnegie Mellon University
Software Engineering Institute

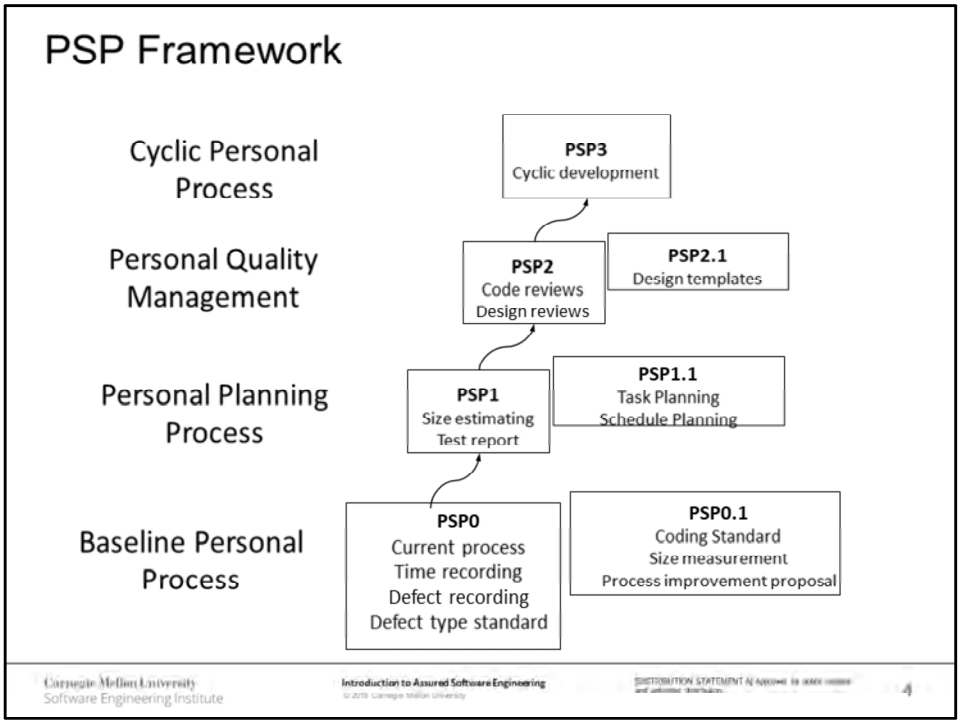
Introduction to Assured Software Engineering

20

The Royal Enfield Bullet, in its present form, was first introduced in Britain in 1949 as a 350cc bike. It incorporated an innovative design element: swing arm suspension. This feature along with its extremely strong single cylinder engine allowed it to excel as a trials bike. The 500cc model was introduced in Britain during the 1950's, winning hundreds of races. This brought the Bullet international recognition and orders came into the factory in Redditch, England from all over the world.

Dr. Deming used the Red Bead Experiment to clearly and dramatically illustrate several points about poor management practices. This includes the fallacy of rating people and ranking them in order of performance for next year, based on previous performance. The Red Bead Experiment uses statistical theory to show that even though a "willing worker" wants to do a good job, their success is directly tied to and limited by the nature of the system they are working within. Real and sustainable improvement on the part of the willing worker is achieved only when management is able to improve the system. http://maaw.info/DemingsRedbeads.htm


Carnegie Mellon University
Software Engineering Institute
CERT
Module 5:
PSP and TSP
(Developed by Mel Rosso Llopart)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



The PSP aims to provide software engineers with disciplined methods for improving personal software development processes. The PSP helps software engineers to:

- Improve their estimating and planning skills.
- Make commitments they can keep.
- Manage the quality of their projects.
- Reduce the number of defects in their work.

The goal of the PSP is to help developers produce zero-defect, quality products on schedule.

PSP training follows an evolutionary improvement approach: an engineer learning to integrate the PSP into his or her process begins at the first level, PSP0, and progresses in process maturity to the final level – PSP3. Each Level has detailed scripts, checklists and templates to guide the engineer through required steps and helps the engineer improve his own personal software process. Humphrey encourages proficient engineers to customize these scripts and templates as they gain an understanding of their own strengths and weaknesses.

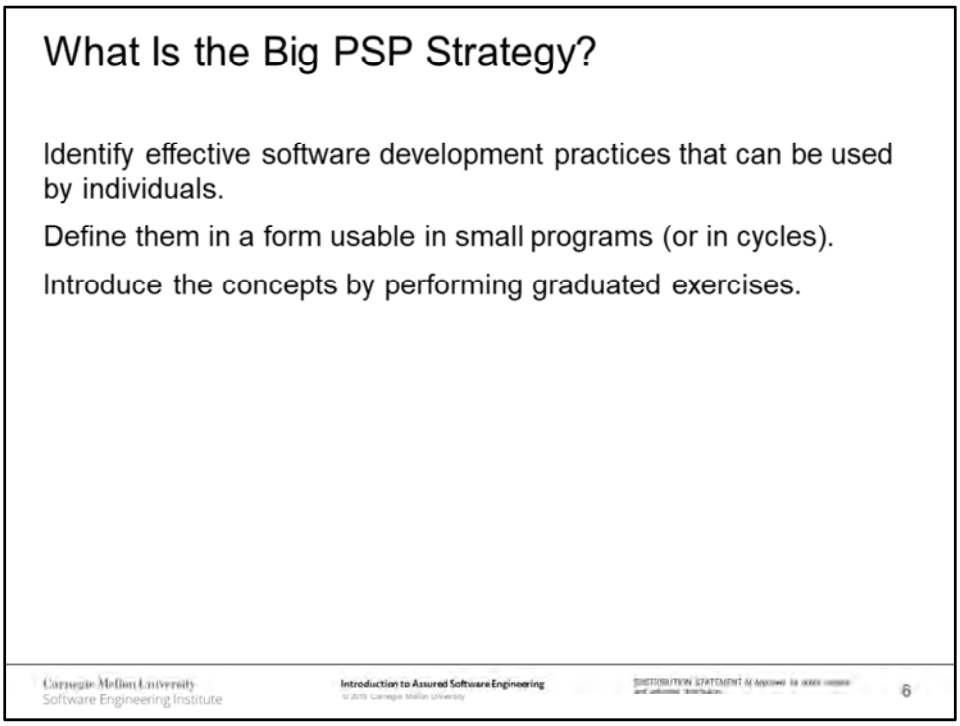
The input to PSP is the requirements; requirements document is completed and delivered to the engineer.

PSP0, PSP0.1 (Introduces process discipline and measurement)PSP0 has 3 phases: planning, development (design, coding, test) and a post mortem. A baseline is established of current process measuring: time spent on programming, faults injected/removed, size of a program. In a post mortem, the engineer ensures all data for the projects has been properly recorded and analyzed. PSP0.1 advances the process by adding a coding standard, a size measurement and the development of a personal process improvement plan (PIP). In the PIP, the engineer records ideas for improving his own process.

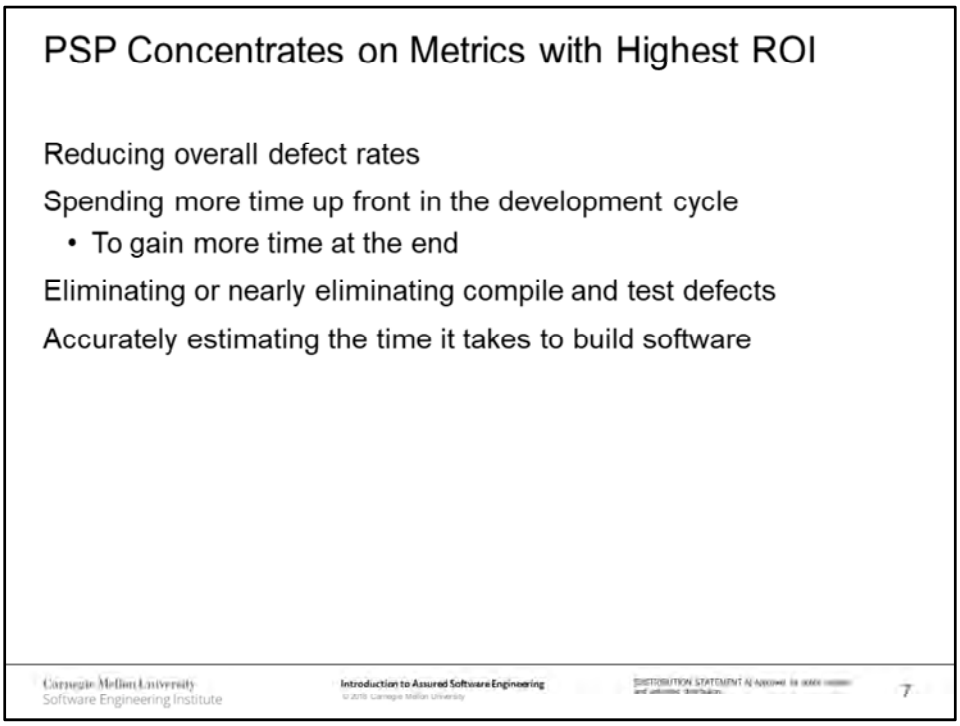
PSP1, PSP1.1 (Introduces estimating and planning)Based upon the baseline data collected in PSP0 and PSP0.1, the engineer estimates how large a new program will be and prepares a test report (PSP1). Accumulated data from previous projects is used to estimate the total time. Each new project will record the actual time spent. This information is used for task and schedule planning and estimation (PSP1.1).

PSP2, PSP2.1 (Introduces quality management and design)PSP2 adds two new phases: design review and code review. Defect prevention and removal are the focus at the PSP2. Engineers learn to evaluate and improve their process by measuring how long tasks take and the number of defects they inject and remove in each phase of development. Engineers construct and use checklists for design and code reviews. PSP2.1 introduces design specification and analysis techniques.

PSP3 is a legacy level that has been superseded by TSP.



These graduated exercises not only allow students to learn the PSP methods but help them to discover their own personal habits.

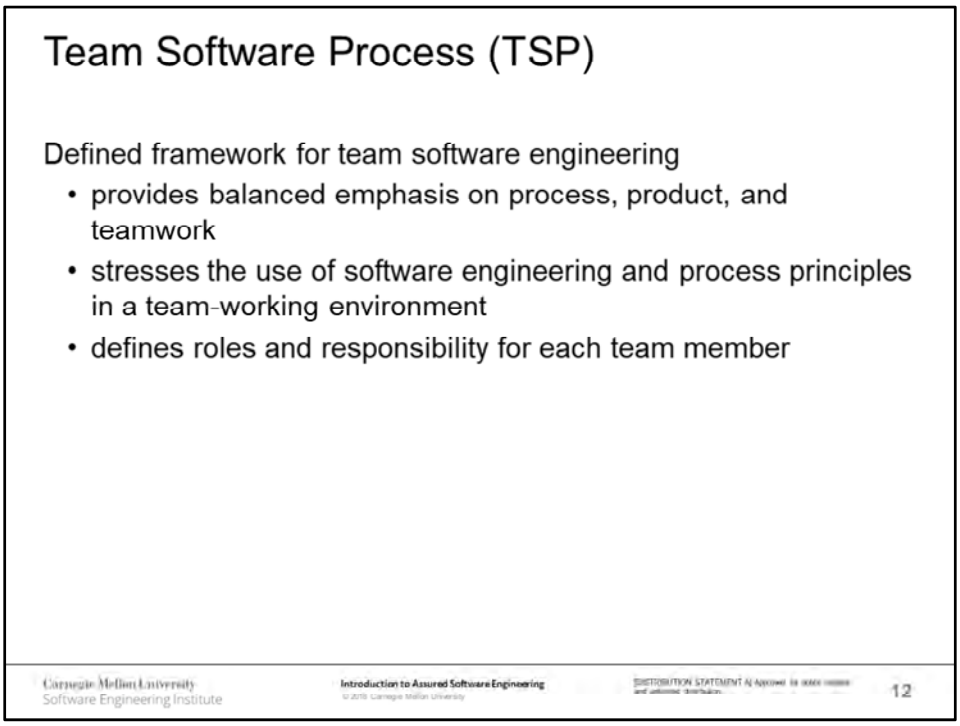


High-quality software is the goal of the PSP, and quality is measured in terms of defects. For the PSP, a quality process should produce low-defect software that meets the user needs. The PSP phase structure enables PSP developers to catch defects early. By catching defects early, the PSP can reduce the amount of time spent in later phases, such as Test.
The PSP theory is that it is more economical and effective to remove defects as close as possible to where and when they were injected, so software engineers are encouraged to conduct personal reviews for each phase of development. Therefore the PSP phase structure includes two review phases:

- Design Review
- Code Review

The key data collected in PSP are time, defect, and size data – the time spent in each phase; when and where defects were injected, found, and fixed; and the size of the product parts. Software developers use many other measures that are derived from these three basic measures to understand and improve their performance. The PSP is intended to help a developer improve their personal process; therefore PSP developers are expected to continue adapting the process to ensure it meets their personal needs

Logging time, defect, and size data is an essential part of planning and tracking PSP projects, as historical data is used to improve estimating accuracy. The PSP uses the PROxy-Based Estimation (PROBE) method to improve a developer's estimating skills for more accurate project planning. For project tracking, the PSP uses the Earned Value method.
The PSP also uses statistical techniques, such as correlation, linear regression, and standard deviation, to translate data into useful information for improving estimating, planning and quality. These statistical formulas are calculated by the PSP tool.



In combination with the Personal Software Process (PSP), the Team Software Process (TSP) provides a defined operational process framework that is designed to help teams of managers and engineers organize projects and produce software products that range in size of sizes beyond from small projects of several thousand lines of code (KLOC) to very large projects greater than half a million lines of code. The TSP is intended to improve the levels of quality and productivity of a team's software development project, in order to help them better meet the cost and schedule commitments of developing a software system.

The initial version of the TSP was developed and piloted by Watts Humphrey in the late 1990s and the Technical Report for TSP sponsored by the U.S. Department of Defense was published in November 2000. The book by Watts Humphrey, Introduction to the Team Software Process, presents a view the TSP intended for use in academic settings, that focuses on the process of building a software production team, establishing team goals, distributing team roles, and other teamwork-related activities.

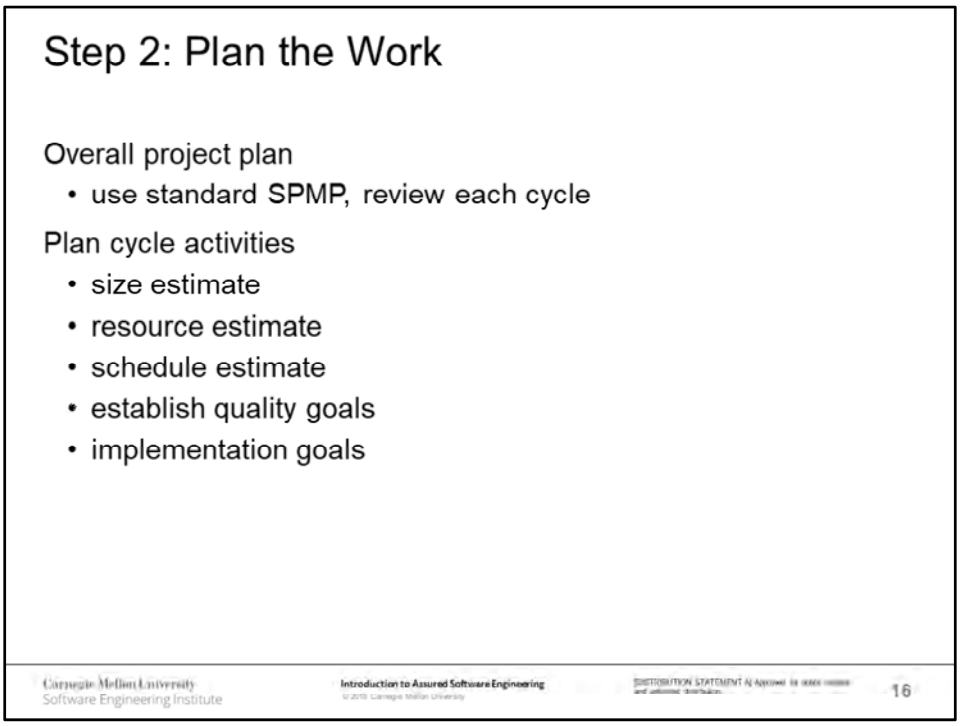
TSP starts with a project launch. The process of forming and building a team does not happen by accident, and it takes time. Teams need to establish their working relationships, determine member roles, and agree on goals. An hour or so spent on team-building issues at the beginning of the project saves time later.

Some of the key activities include:

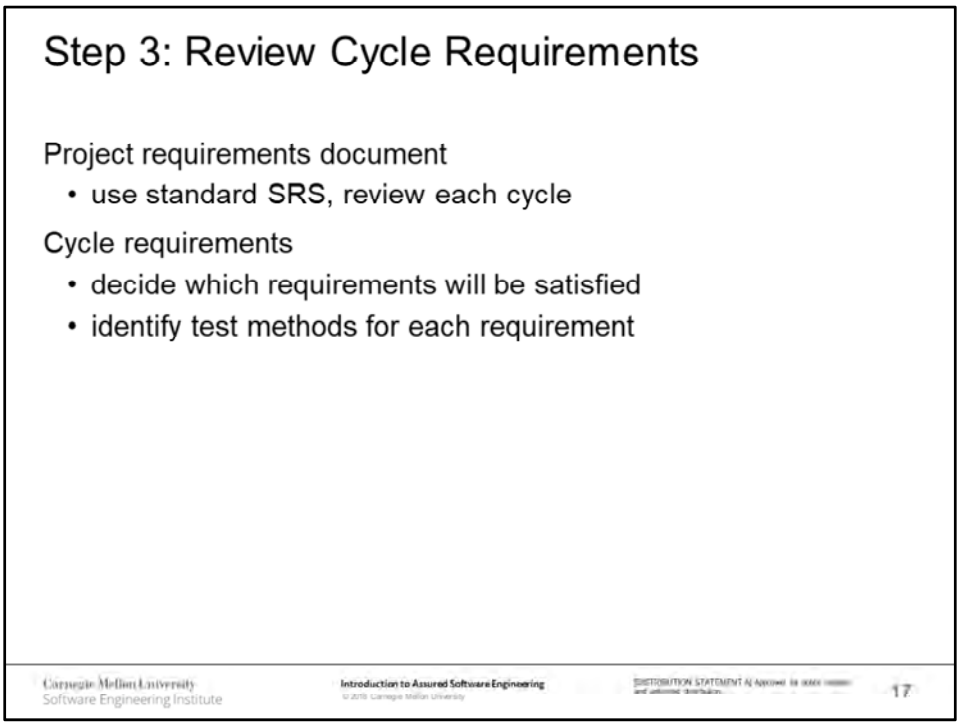
- Management presents expectations.
- Team estimates effort and determines whether it can meet management's expectations.
- Team makes quality plan to ensure a good product.
- Team and management negotiate if necessary.
- Team and management agree on plan, and work begins.

Step 1: Develop Strategy

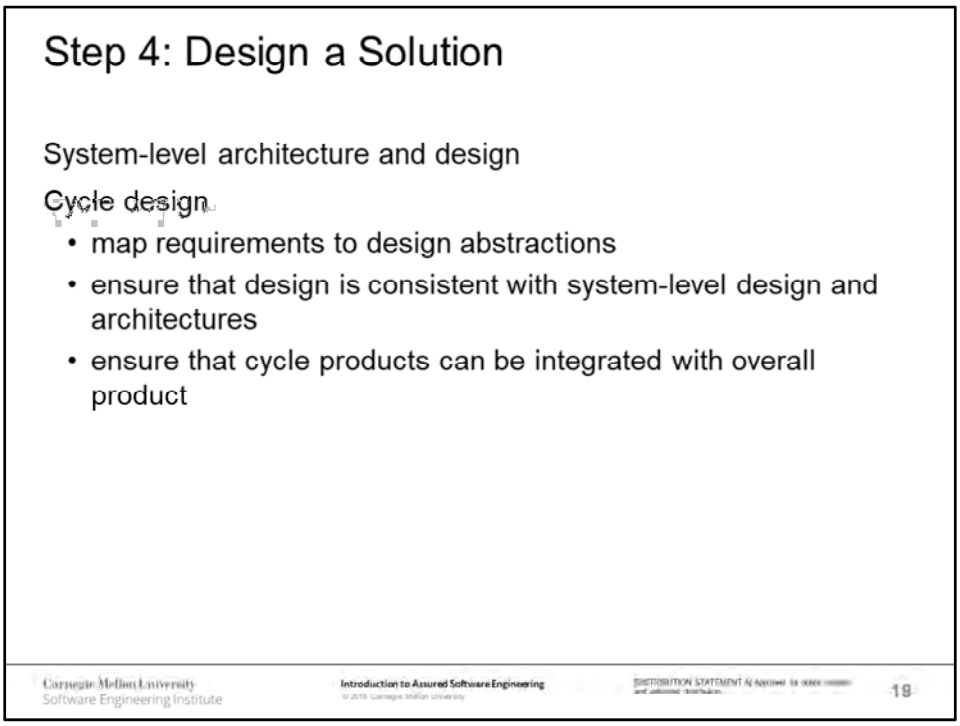
In the launch step, your team agreed on what a successful project would look like and established measureable role and team goals. In this step, you devise a strategy for doing the work, create a conceptual product design, and make a preliminary estimate of the product's size and development time. If the estimate indicates that the work will take longer than the time you have available, you revise the strategy until the work fits the time available. Finally, you document the strategy. This step is done before you do project planning.



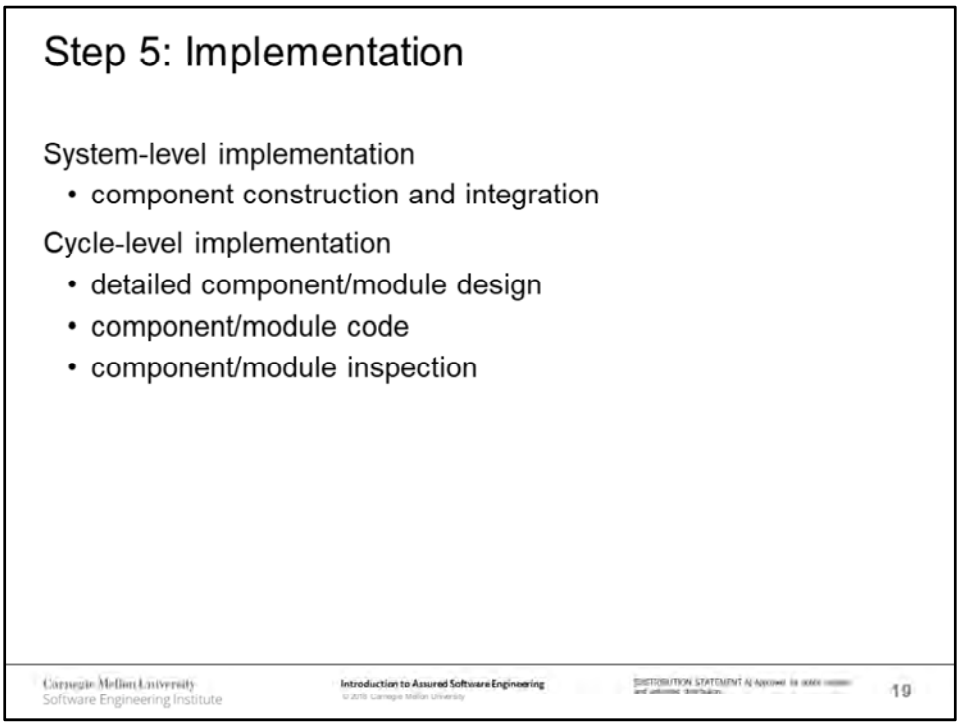
Step 2 creates the plans you will need to manage your project. The complexity of a development plan is governed by the complexity of thee work that you plan to do. A completed TSP plan has several forms that contain size and time estimates, the schedule , and a quality plan. You will use Earned Value and learn how to make defect-injection and yield estimates to see whether the plan meets your team's quality objectives.



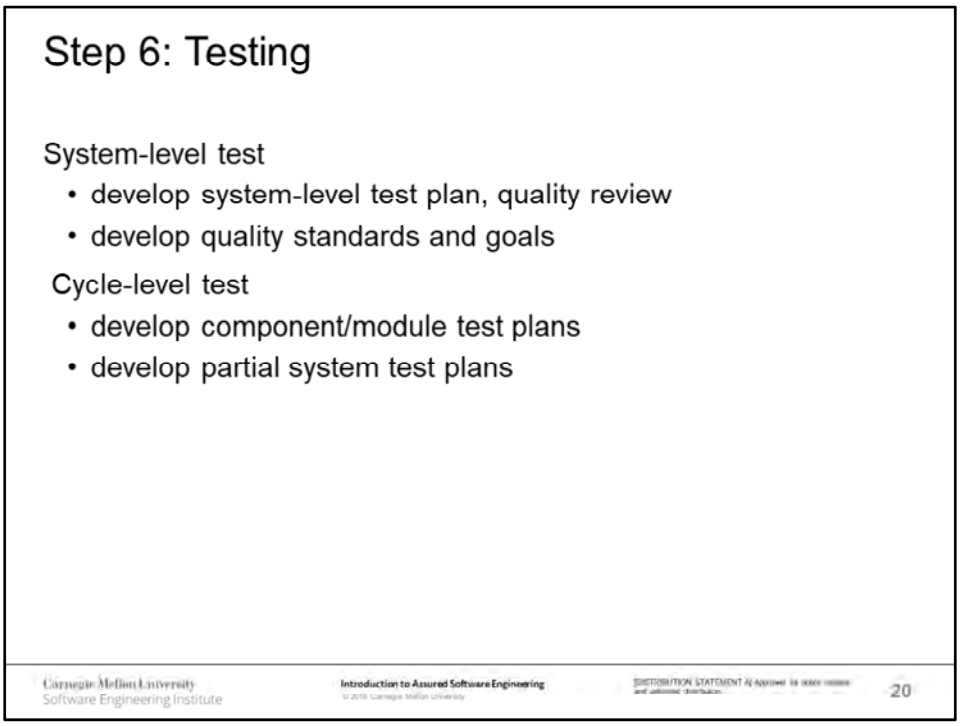
In this step, the software requirements specification is produced. This specification describes the functions you intend the product to perform. It will also provide clear and unambiguous criteria for evaluating the finished product.



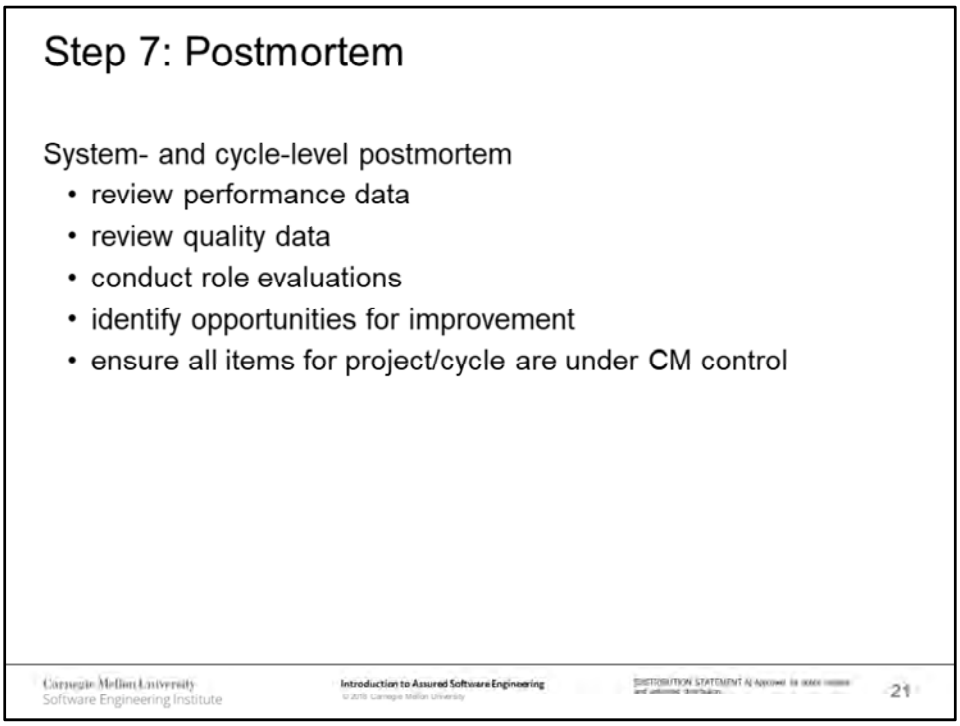
This step is to ensure that engineers produce thorough, high-quality designs. When designing with teams, you first produce the overall design structure and then divide this overall product into principal components. The team members then separately design these components and provide their designs to the development manager, who combines them in the system design specification. It is also important to produce and inspect the integration test plan.



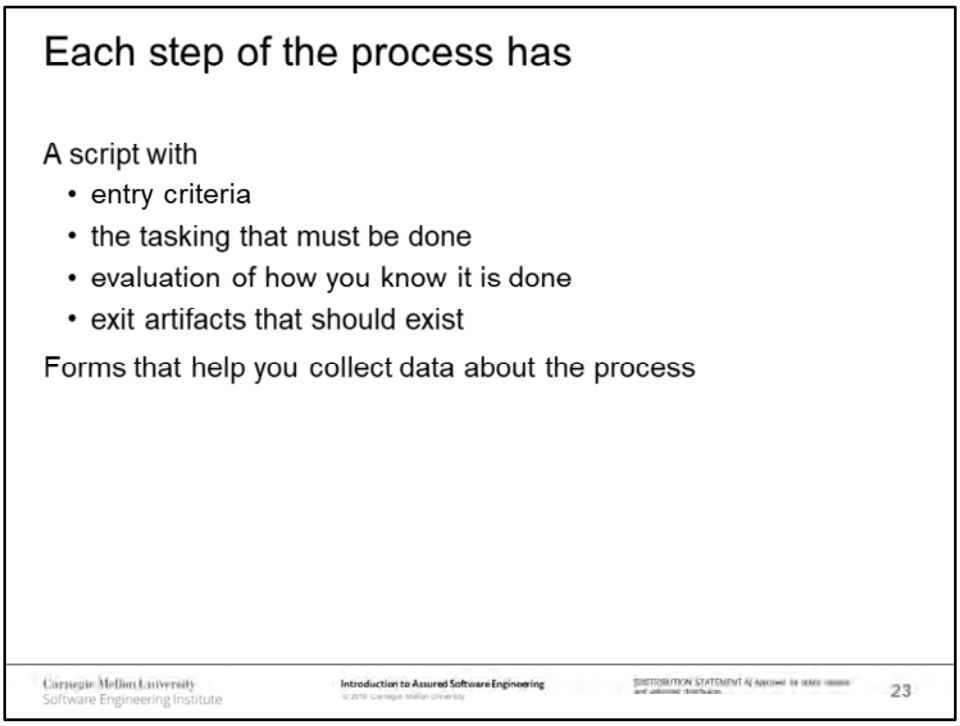
The principal activities in the implementation process are implementation planning, detailed design, detailed-design inspection, coding, code inspection, unit testing, component quality review, and component release. Implementation standards are also developed to add to and extend the standards defined in the design phase.



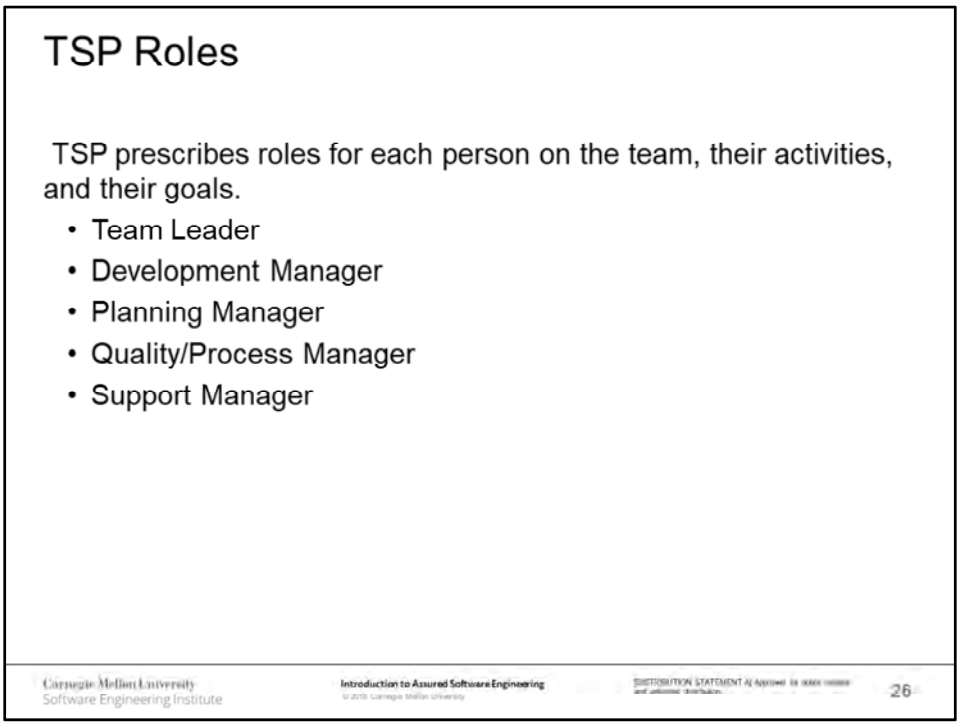
This step covers both testing and documentation. The purpose of testing is to asses the product, not to fix it. With TSP you have the data to judge which parts of the system are most likely to be defect-prone. Typically, those modules with the most defects in test are likely to have the most defects remaining after test.



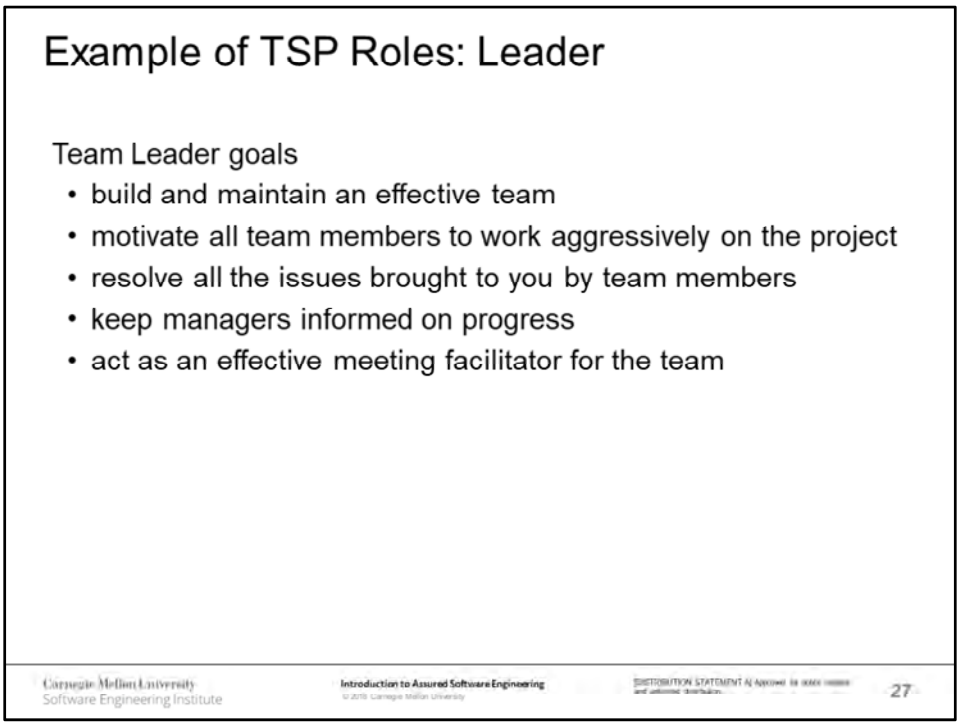
This is the final step in the PSP process. In this step, you review the team's work to ensure that you have completed all the needed tasks and recorded all the required data. You also reexamine what you did in this cycle, both to learn what went right and wrong and to see how to do the job better the next time.



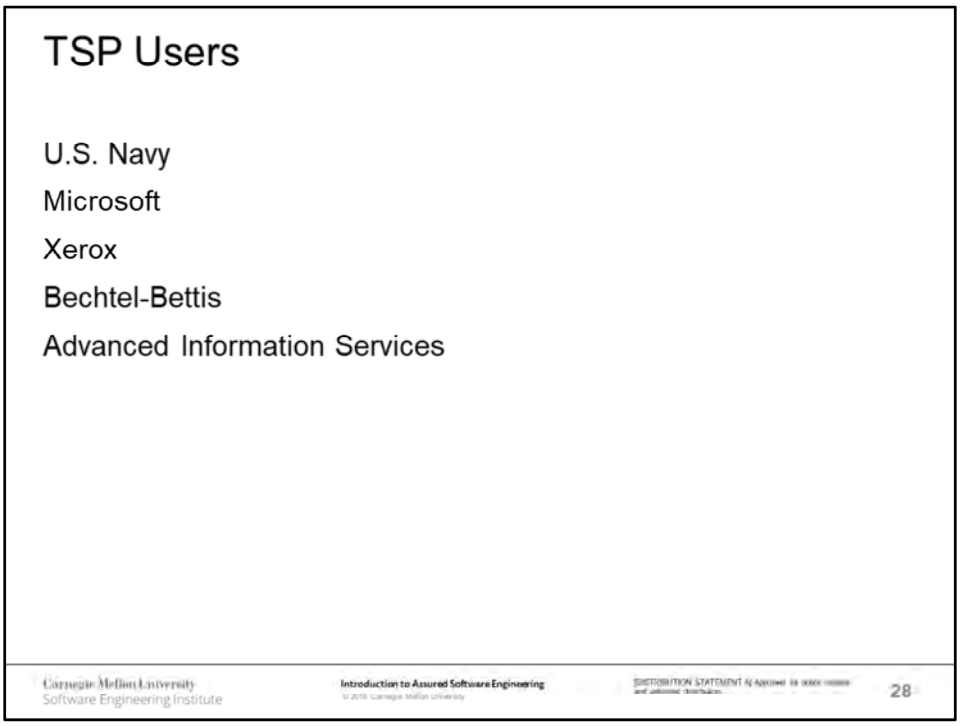
The scripts and example forms are described in the TSP book.



Each role represents a single facet of the overall team's activities.

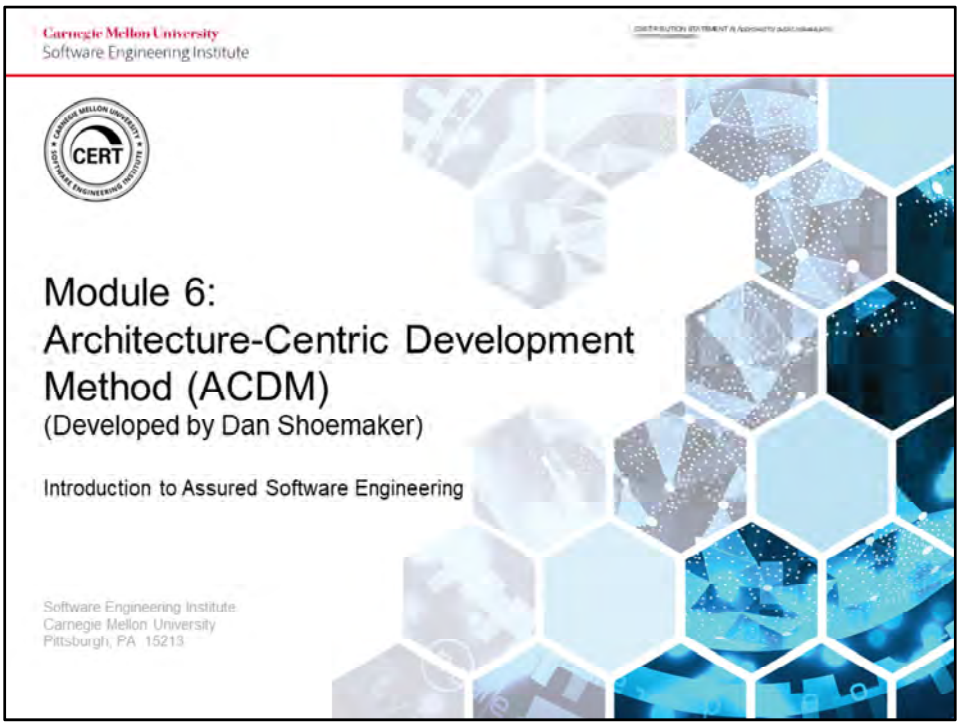


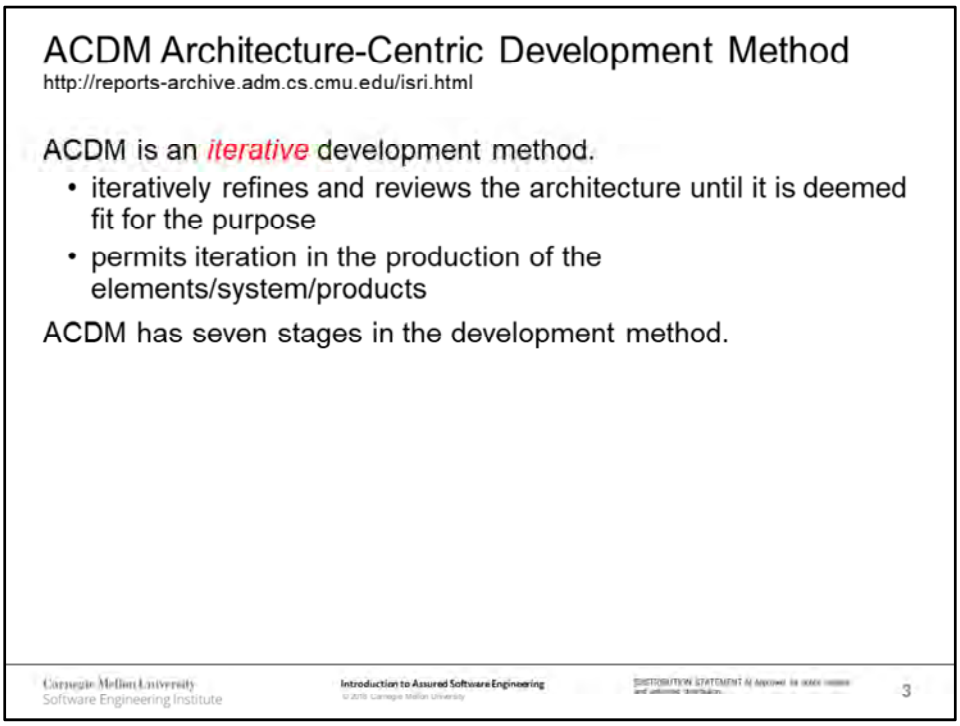
The team leader's goals and measures are discussed in the TSP book.



There are many companies that use TSP as you can see below.

Adobe Systems Incorporated, Advanced Information Services Inc., Altran Praxis Ltd, CAE USA Inc., CGI Group, Inc., Composite Engineering Inc., Davis Systems, DEK International GmbH, Delaware Software, European Software Institute – Center, Eastern Europe Expert Software Consultants Ltd. Faculdade de Engenharia da Universidade do Porto Fuji Xerox Co., Ltd. FUJIFILM Corporation FUJIFILM Software Co., Ltd. Hitachi Solutions, Ltd. HP Enterprises Services, LLC Instituto Tecnologico y de Estudios Superiores de Monterrey (TEC) It Era S.A. de C.V. Johannesburg Centre for Software Engineering Kernel Corporativo S.A. de C.V. Kyushu Institute of Technology Kyushu University Mitsubishi Space Software Co., Ltd. Neoris de Mexico S.A. de C.V. Next Process Institute Ltd. Oracle Corporation PersonalSoft S.A.S. Procesix Colombia Ltda. Process & Project Health Services PS&J Consulting Services Inc. QuarkSoft, S.C. SEONTI S.A. de C.V. Siemens AG SILAC Ingenieria de Software S.A. de C.V. SKIZCorp Technology Softtek Integration Systems, Inc. Software Engineering Competence Center (SECC) Software Industry Excellence Center de Mexico SC Software Park Thailand - NSTDA Software Technology Process & People, Inc. Strongstep-Innovation in Software Quality Team Management Consulting Organization, Inc. Technology and Software Provider S.A. de C.V. Tektronix Communications The Wall Group Toshiba Corporation Towa Software S.A. de C.V. U.S. Air Force CRSIP STSC U.S. Naval Air Systems Command (NAVAIR) U.S. Naval


Carnegie Mellon University
Software Engineering Institute
CERT
Module 6:
Architecture-Centric Development Method (ACDM)
(Developed by Dan Shoemaker)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

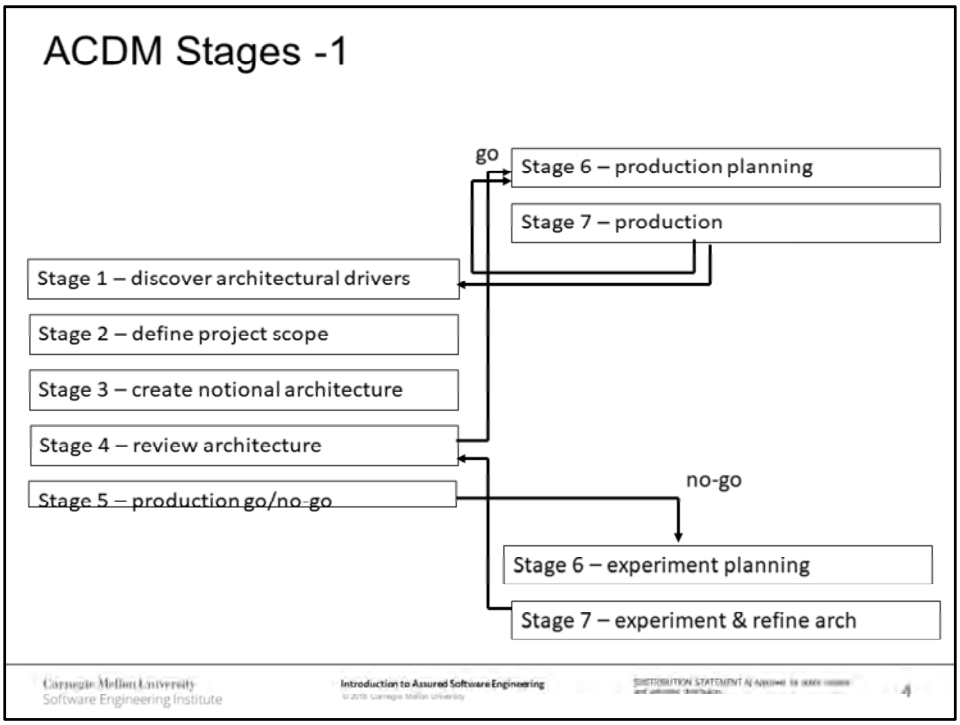


The architecture centric design method is a method for software architectural design developed in 2005 by Anthony J. Lattanze of the SEI at Carnegie Mellon University. It places the software architecture at the center of a development effort rather than software processes. ACDM weaves together product, technology, process, and people into a cohesive lightweight, scalable development method.

The key goals of ACDM are to help software development teams:

- Get the information from stakeholders needed to define the architecture as early as possible.
- Create, refine, and update the architecture in an iterative way throughout the lifecycle whether the lifecycle is waterfall or iterative.
- Validate that the architecture will meet the expectations once implemented.
- Define meaningful roles for team members to guide their efforts.
- Create better estimates and schedules based on the architectural blueprint.
- Provide insight into project performance.
- Establish a lightweight, scalable, tailorable, repeatable process framework.

“The Architecture Centric Development Method,” Anthony J. Lattanze, February 2005, CMU-ISRI-05-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA



ACDM Stages

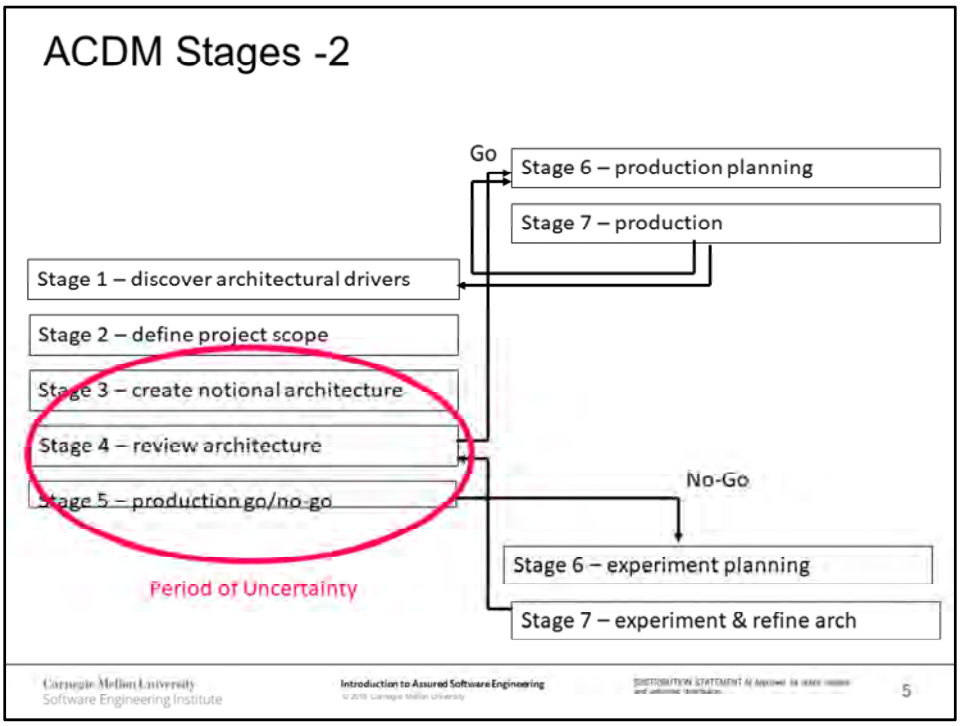
Stage 1: Discover Architectural Drivers - Meet with client stakeholders to discover and document architectural drivers: high-level functional requirements, constraints and quality attributes.
Stage 2: Establish Project Scope - Distill architectural drivers into an architectural drivers specification. Create a Statement of Work and Preliminary Project Plan.
Stage 3: Create Notional Architecture - Create the initial architecture which includes a run-time view, code view, and physical view of the system.
Stage 4: Architectural Review - Review the notional architecture to discover and document risks and issues.
Stage 5: Production Go/No-Go - Prioritize and list the risks and issues discovered during the architecture review and decide whether the architecture is ready for production (production step 6) or whether it needs to be refined (refine step 6).

Refine

Stage 6: Experiment Planning - Team creates experiments to mitigate risks and/or issues that were discovered during the review. Experiments are targeted, planned, technical prototypes that are for the purpose of exploring technical issues associated with the architecture or to further explore the architectural drivers.
Stage 7: Experiment Execution and Architecture Refinement - The team carries out the experiments and documents the results. The architecture is refined based on the results of the experiments.
Return to Stage 4, Architectural Review to review the refined architecture.

Production

Stage 6: Production Planning - Team creates a detailed plan for the construction of the system based on the refined architecture. Each element of the architecture has an “owner” and shepherds the construction of the element to completion. The plan schedules time and resources for detailed element design, reviews, construction, test, and so forth.
Stage 7: Production - The team executes the production plan and is actively engaged in building the system. Production includes construction of the elements of the architecture, integration of the system, as well as element and system test. Production may result in producing the whole system, parts of the system, or in deliverable increments of the system.



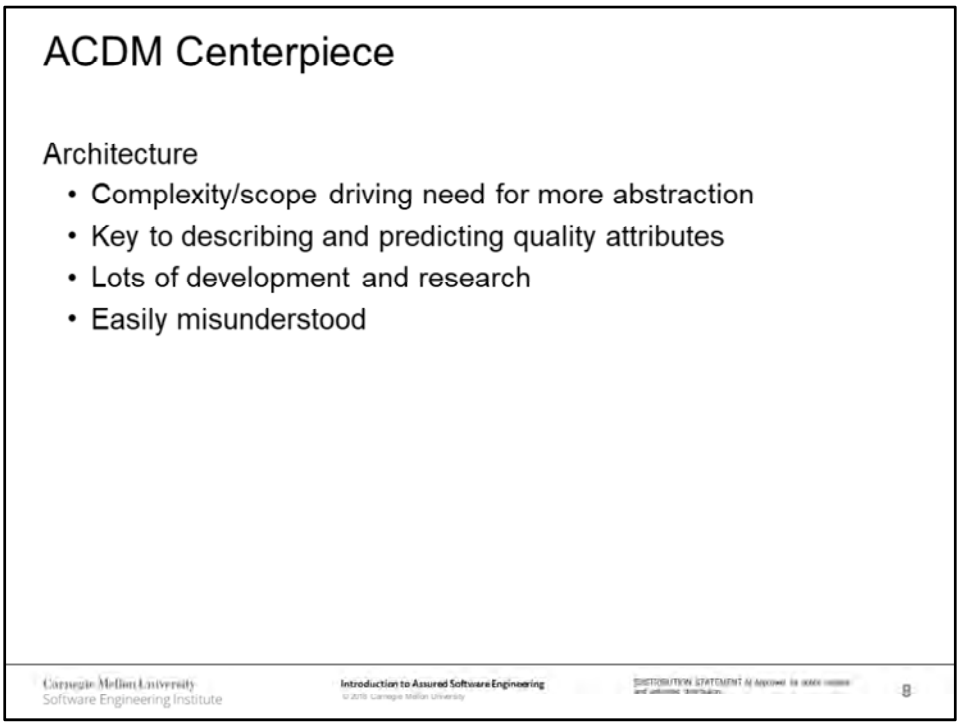
Stage 1-5 focuses on

- how long it will take to discover the architectural drivers
- create the notional architecture
- how many experiments
- refining the architecture for production

Stage 6-7 focus on

- mapping architectural elements to tasks, schedules, and personnel
- how long it will take to design, construct, and test each element
- how long it will take to integrate the elements of the architecture into a system

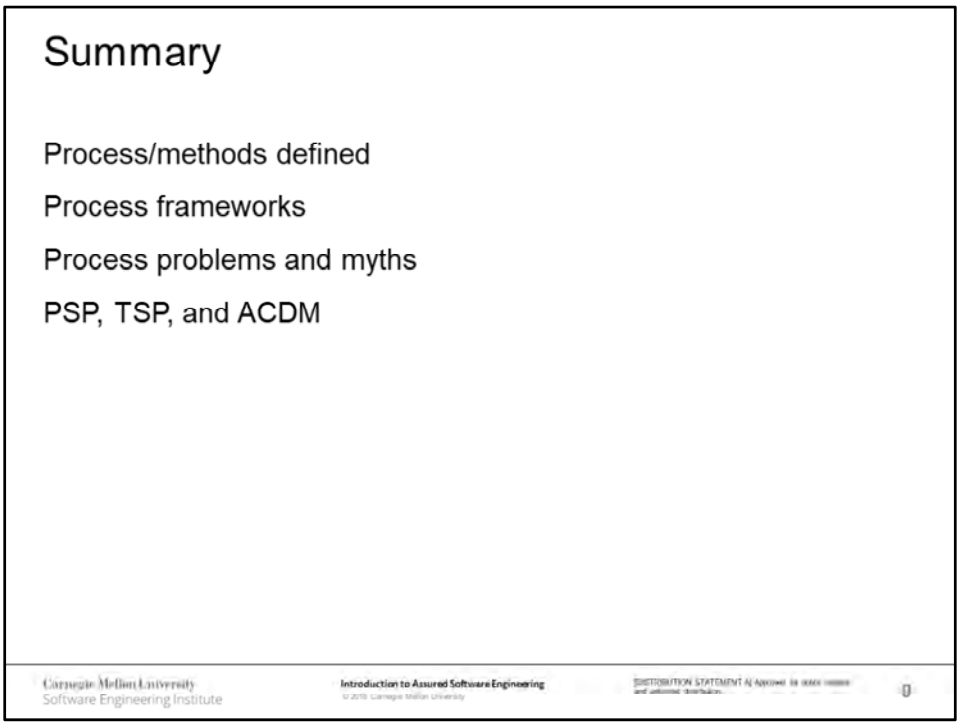
Stage 5 is roughly on the mid-point. Those activities prior to stage 5 are discovery oriented where developers gather information to build, refine, and baseline the architecture. Since not much is known about the product, project, or client stakeholders; this period of time is characterized as the Period of Uncertainty. ACDM activities prior to stage 5 are designed to overcome the Period of Uncertainty as quickly as possible. Those activities occurring after stage 5 are detailed design and construction oriented. The architecture is base-lined and should embody the needs and desires of the stakeholders; this period of time is characterized as the Period of Certainty. The focus of the Preliminary Project Plans is determining how long the team will spend creating and refining the architecture NOT on building the final product. Philosophically speaking, ACDM works best when the team defines the notional architecture, reviews it, and baselines the architecture as quickly as possible. The benefit of this approach is that the Period of Uncertainty is shortened, and the Period of Certainty is reached earlier. Once the Period of Certainty is reached, more accurate estimates for production can be made.



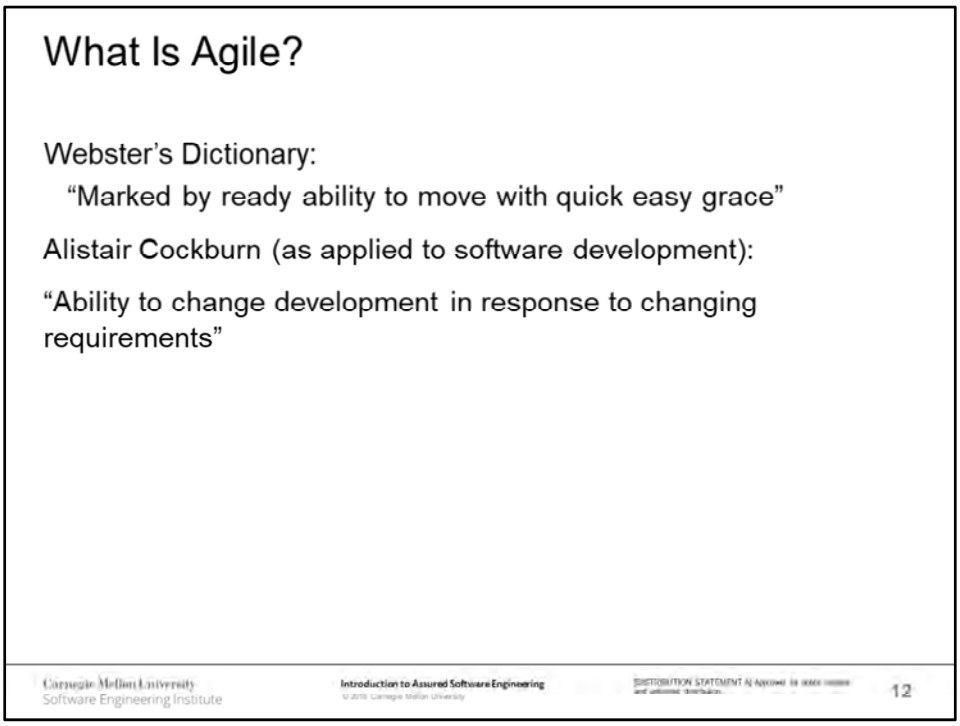
Functionality is a measure of how well a system does the work it was intended to do, but functionality is not all that matters in software development. Properties like interoperability, modifiability, and portability also matter as much as functionality does. These properties are determined primarily by the software structure – or the software architecture.

While many structures can satisfy functionality, few can satisfy the required functionally and the quality attribute properties needed in a system. Achieving quality attributes in a predicable way can only be accomplished by deliberately selecting the appropriate structures early in the development process. This is a radical departure from high speed, lightweight programming methodologies (e.g. XP) that focuses on functionality and prescribes writing software until a product emerges – architectures also emerge in this paradigm.

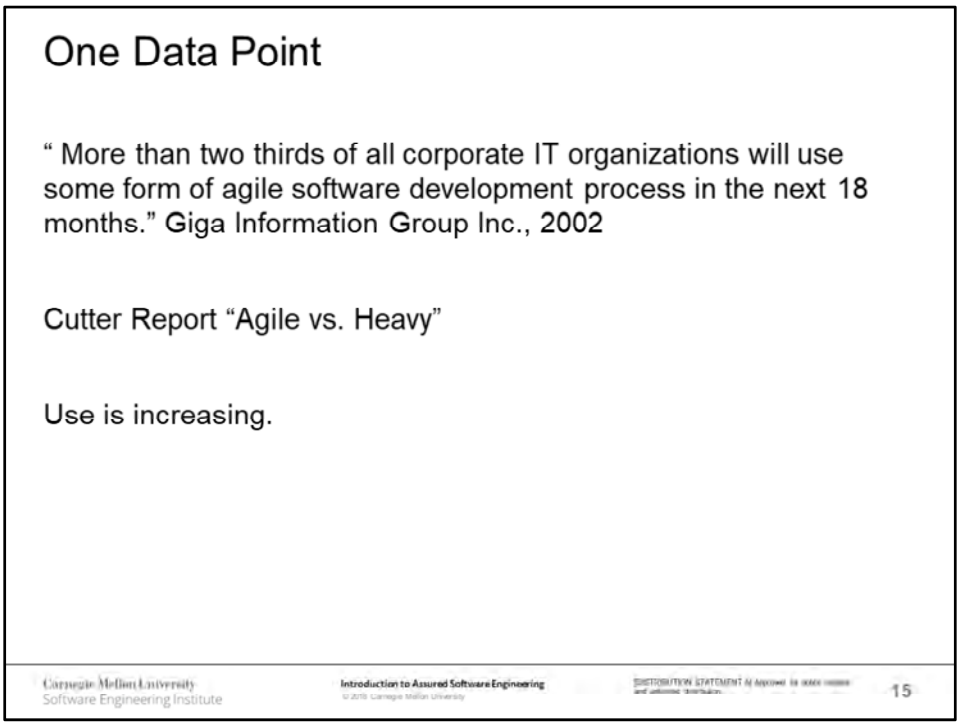
Emergent architectural structures may or may not meet the expectations of the broader stakeholders. Other methods espouse high ceremony processes and heavy emphasis on document production. The Architecture Centric Development Method (ACDM) can be differentiated from these extremes in that ACDM places the software architecture at the center of a development effort rather than software processes. Like architectures in the building and construction industries, ACDM prescribes using the architecture design to drive not only the technical aspects of the project, but also the programmatic issues of a development effort as well. ACDM weaves together product, technology, process, and people into a cohesive lightweight, scalable development method.



Summarize the main points of the talk.



Alistair Cockburn was one of the developers of the Agile Development Manifesto.



Rapid development and delivery is now often the most important requirement for software systems

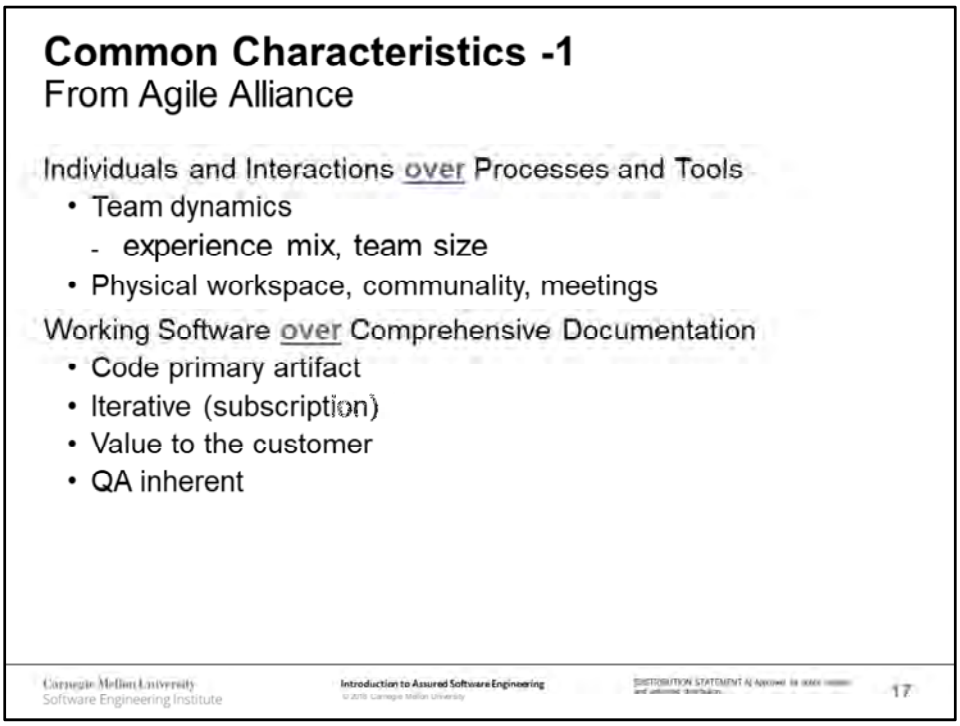
- Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
- Software has to evolve quickly to reflect changing business needs.

Rapid software development

- Specification, design and implementation are inter-leaved
- System is developed as a series of versions with stakeholders involved in version evaluation

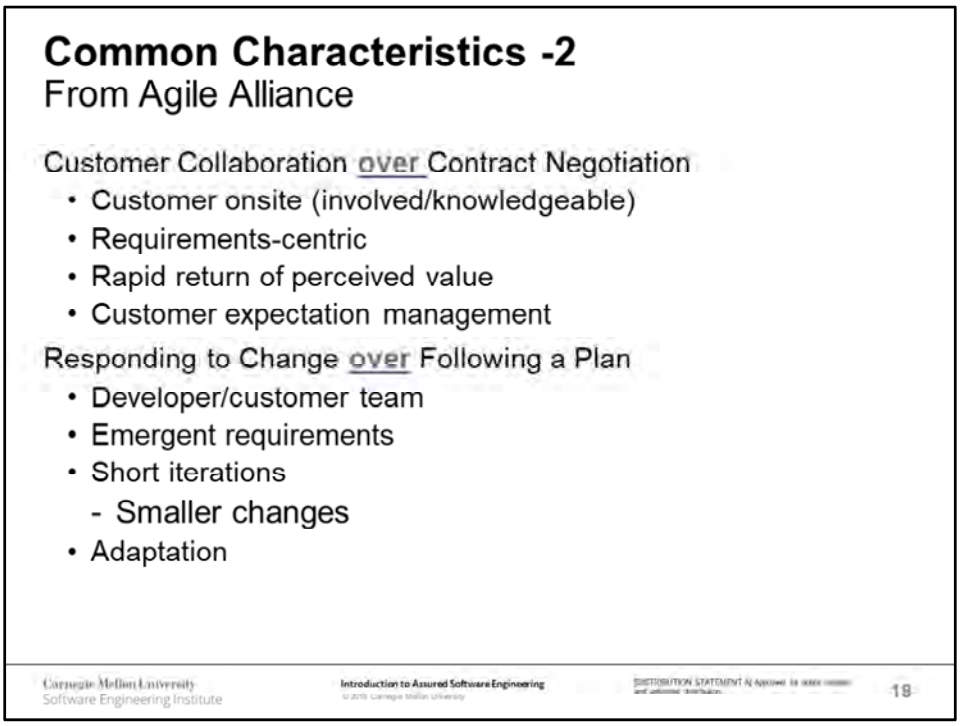
And .. Chaos Report published in 2012.

- 49% of businesses say most of their company is using Agile development
- 52 % of customers are happy or very happy with Agile projects
- The number of those who plan to implement agile development in future projects has increased from 59% in 2011 83% in 2012.
- The most popular Agile method used is Scrum (52%)



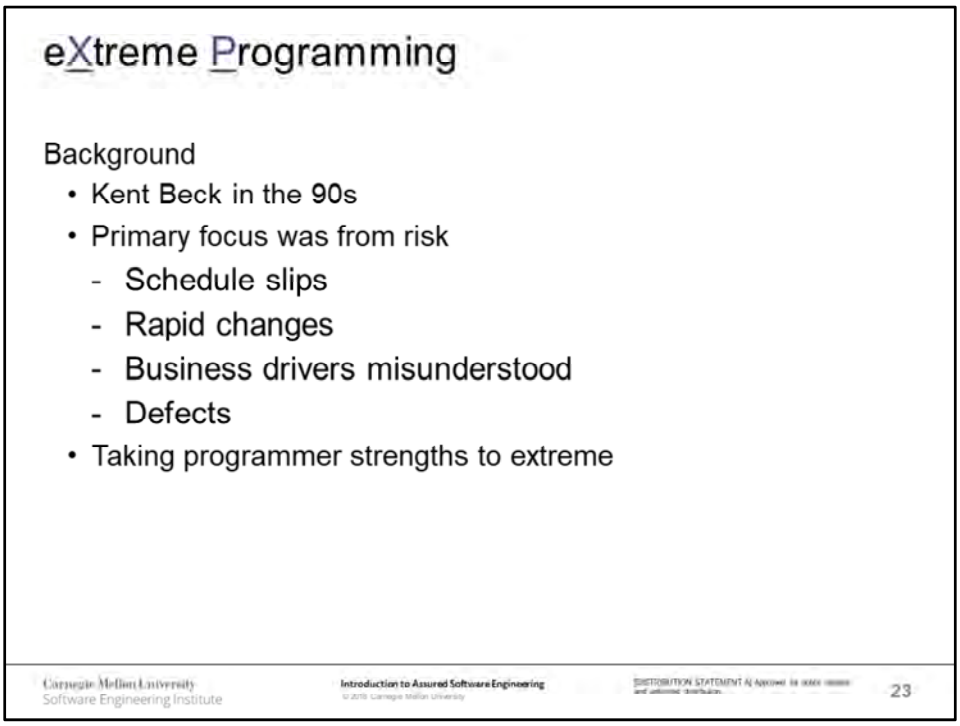
Statements taken from the Manifesto for Agile Software Development

"That is, while there is value in the items on
the right, we value the items on the left more."



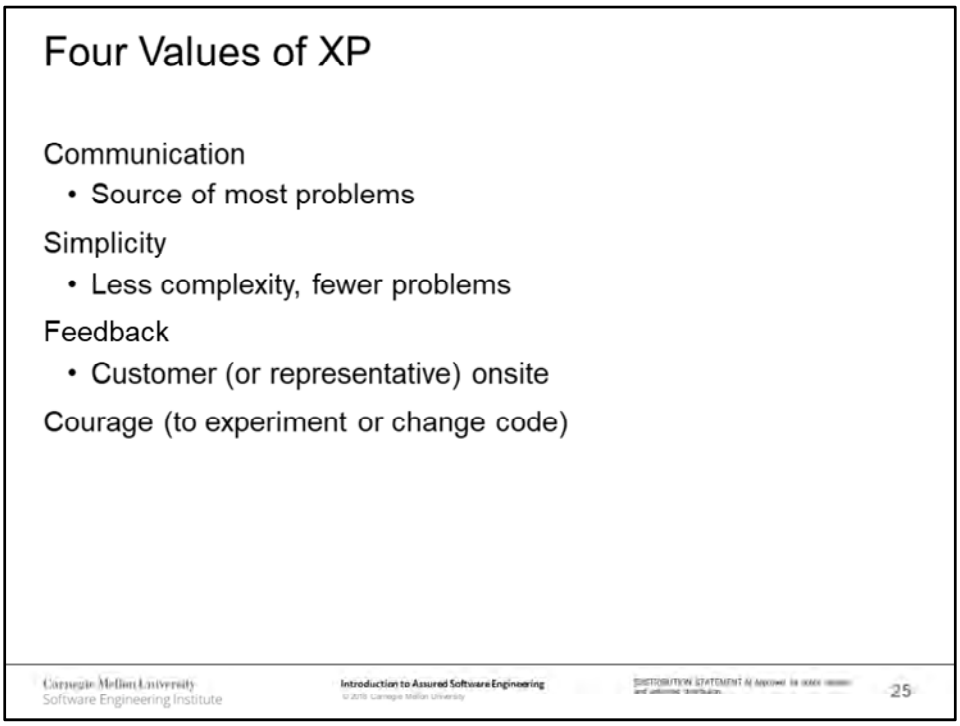
Statements taken from the Manifesto for Agile Software Development

"That is, while there is value in the items on
the right, we value the items on the left more."



Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of Extreme Programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed continuously, i.e. the practice of pair programming. Wikipedia

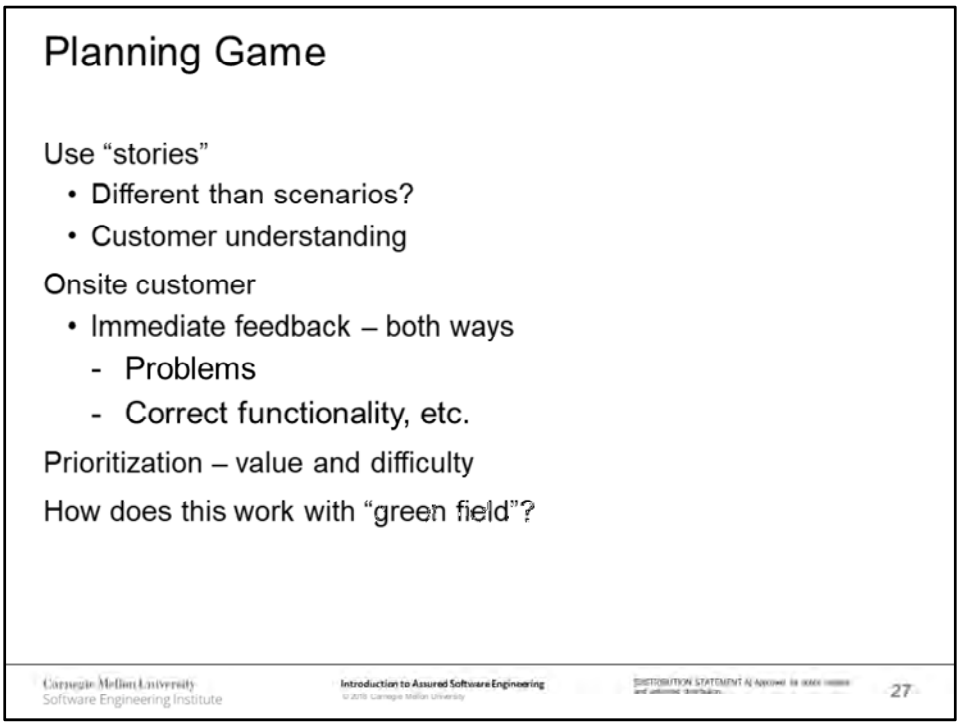


XP makes a big issue about its core value of Communication. This is wonderful, as communication is definitely a key factor to the success of any project, XP or otherwise. Unfortunately, XP also makes a big issue about not doing any documentation (or at least very little, or none at all). I think this is partly why XP has such a broad appeal amongst earnest young programmers.

Simplicity - XP promotes a throwaway approach to source code (i.e. write, toss, rewrite). And so, through countless waves of throwaway code, the design gradually evolves. To counter this time-consuming, high risk approach, we are encouraged to keep everything simple. The theory is that if you are constantly re-evaluating where the project needs to go, then you might lose only a week or two down any particular design blind alley. From Extreme Programming Explained, "XP is making a bet. It is betting that it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway.“

Feedback - XP values feedback as a way of determining the current state of the system. From Extreme Programming Explained, "The business people say, "I had no idea that was so expensive. Just do this one third of it. That will do fine for now.“

Courage - From Extreme Programming Explained, "Go home at 5PM... Notice that nothing is hanging over your head: everything you've done for the day is integrated or tossed.“ Kent Beck, "Maybe you have three design alternatives. So, code a day's worth of each alternative, just to see how they feel. Toss the code and start over on the most promising design."



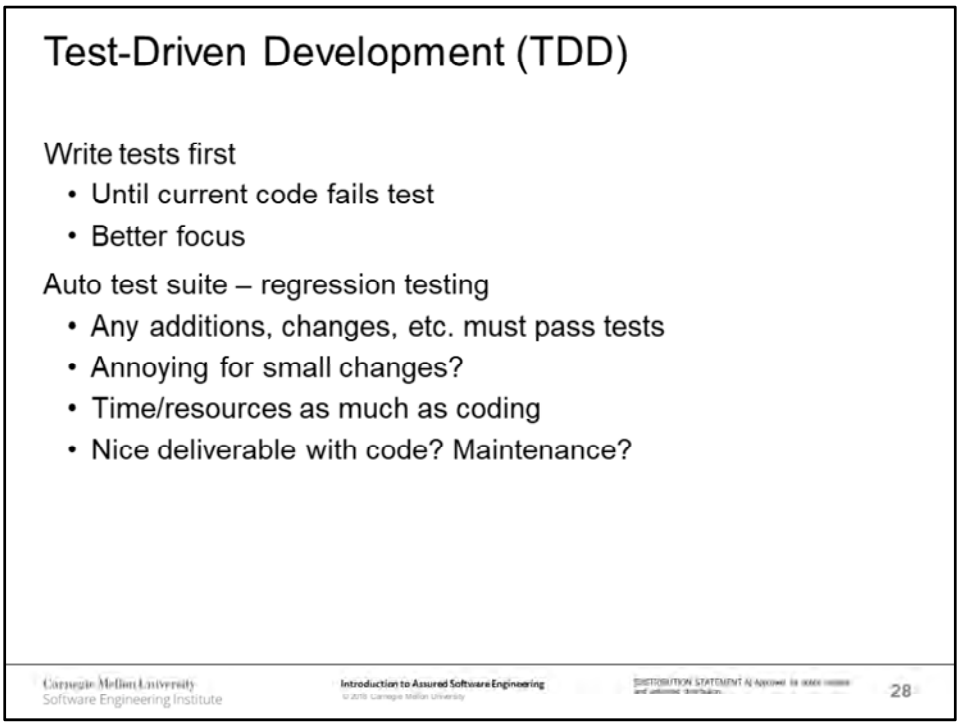
The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The planning process is divided into two parts:
Release Planning: This is focused on determining what requirements are included in which near-term releases, and when they should be delivered. The customers and developers are both part of this. Release Planning consists of three phases:

- Exploration Phase: In this phase the customer will provide a shortlist of high-value requirements for the system. These will be written down on user story cards.
- Commitment Phase: Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release.
- Steering Phase: In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed.

Iteration Planning: This plans the activities and tasks of the developers. In this process the customer is not involved. Iteration Planning also consists of three phases:

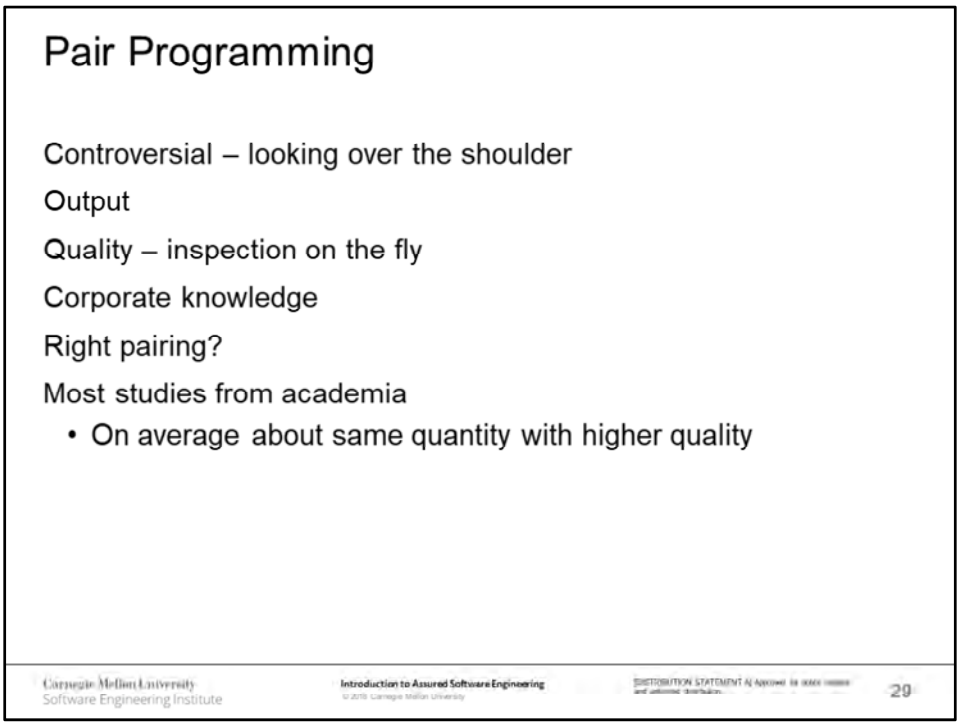
- Exploration Phase: Within this phase the requirement will be translated to different tasks. The tasks are recorded on task cards.
- Commitment Phase: The tasks will be assigned to the programmers and the time it takes to complete will be estimated.
- Steering Phase: The tasks are performed and the end result is matched with the original user story.

The purpose of the Planning Game is to guide the product into delivery. Instead of predicting the exact dates of when deliverables will be needed and produced, which is difficult to do, it aims to "steer the project" into delivery using a straightforward approach. Wikipedia

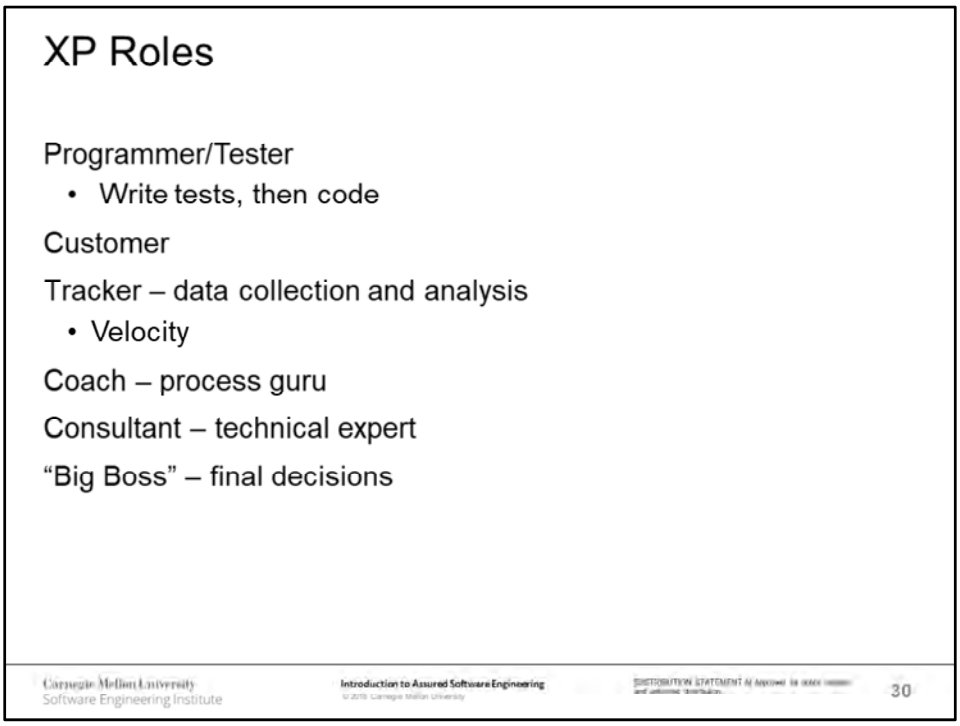


Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right. *Wikipedia*



Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the driver, writes code while the other, the observer, reviews each line of code as it is typed in. The two programmers switch roles frequently. While reviewing, the observer also considers the "strategic" direction of the work, coming up with ideas for improvements and likely future problems to address. This frees the driver to focus all of his or her attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide. *Wikipedia*



The XP Programmer is responsible for implementing the code to support the user stories.

The XP Tester role helps the customer define and write acceptance tests for user stories.

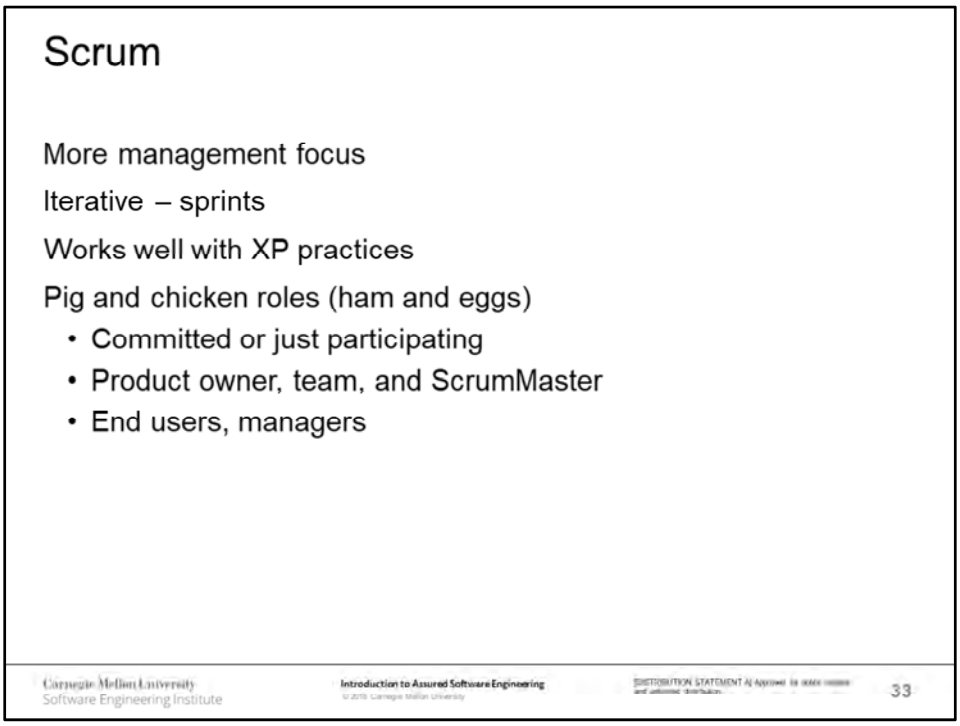
The XP Customer role has the responsibility of defining what is the right product to build, determining the order in which features will be built, and making sure the product actually works.

The XP Tracker role measures and communicates the team's progress.

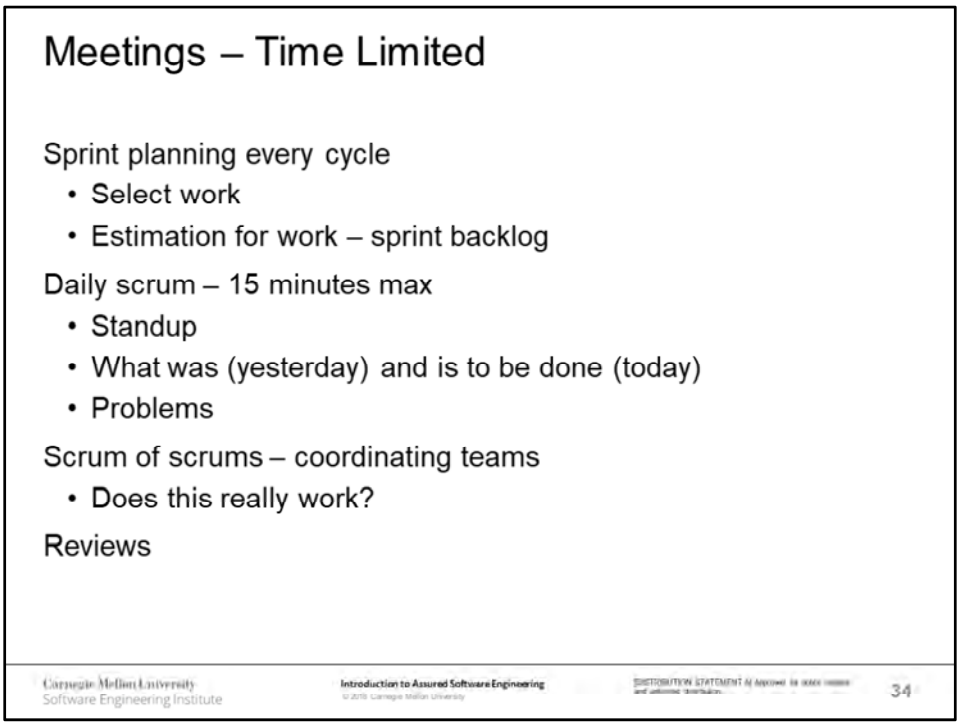
The XP Coach is a supporting role which helps a team stay on process and help the team learn.

The XP Consultant is hired when the team needs additional special knowledge.

The XP Big Boss is the manager of the project and provides the resources.



Scrum is an iterative and incremental Agile software development framework for managing software projects and product or application development. It defines "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal." It challenges assumptions of the "traditional, sequential approach" to product development. Scrum enables teams to self-organize by encouraging physical co-location of all team members and daily face to face communication among all team members and disciplines in the project. A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements. *Wikipedia*



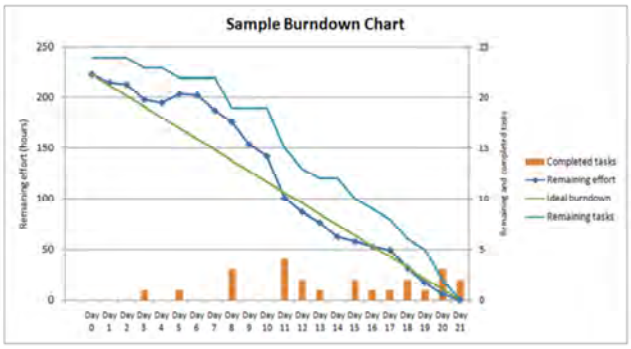
Meeting time is limited but there are frequent meetings.

Artifacts

Product backlog – entire project tasks

Sprint backlog – that sprint's tasks

Burndown chart

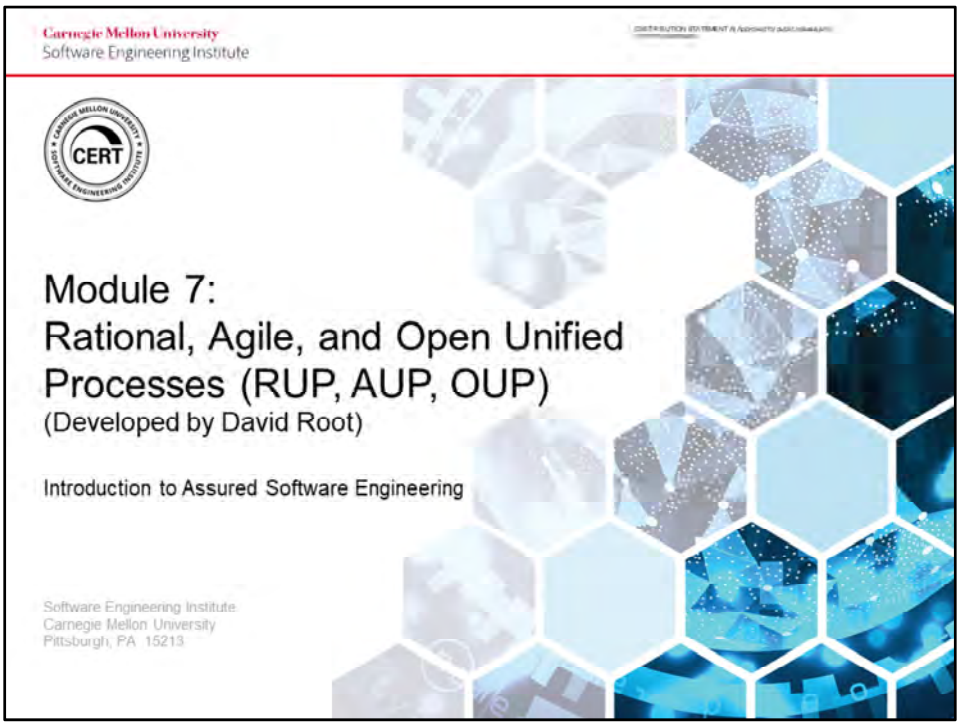


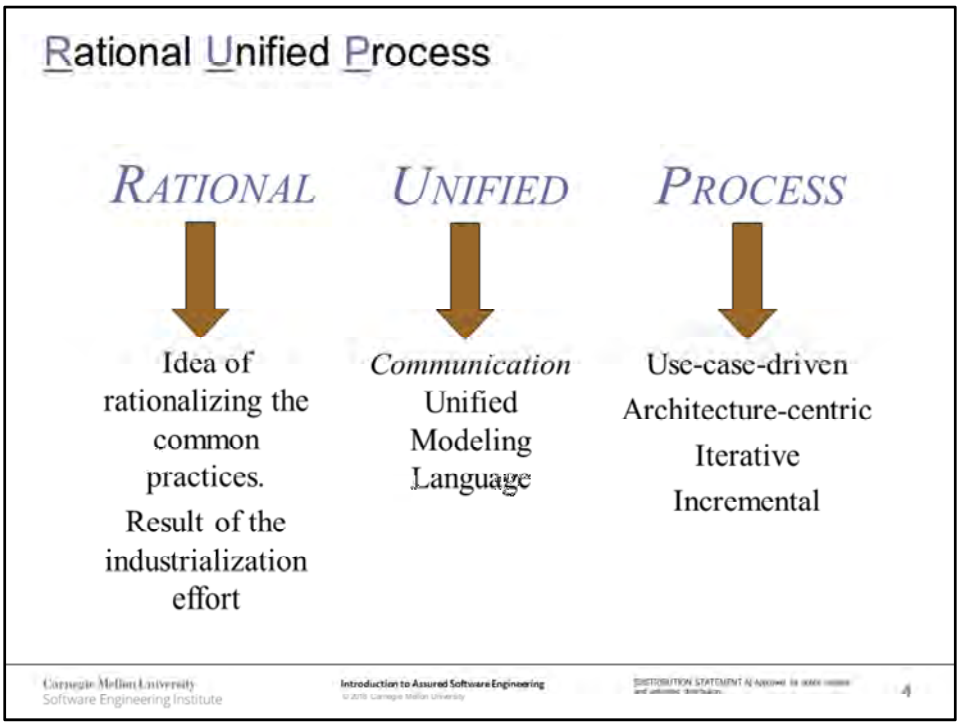
http://en.wikipedia.org/wiki/File:SampleBurndownChart.png

The *product backlog* is an ordered list of *requirements* that is maintained for a product. It consists of features, bug fixes, non-functional requirements, etc.—whatever needs to be done in order to successfully deliver a viable product. The *product backlog items* (PBIs) are ordered by the Product Owner based on considerations like risk, business value, dependencies, date needed, etc.

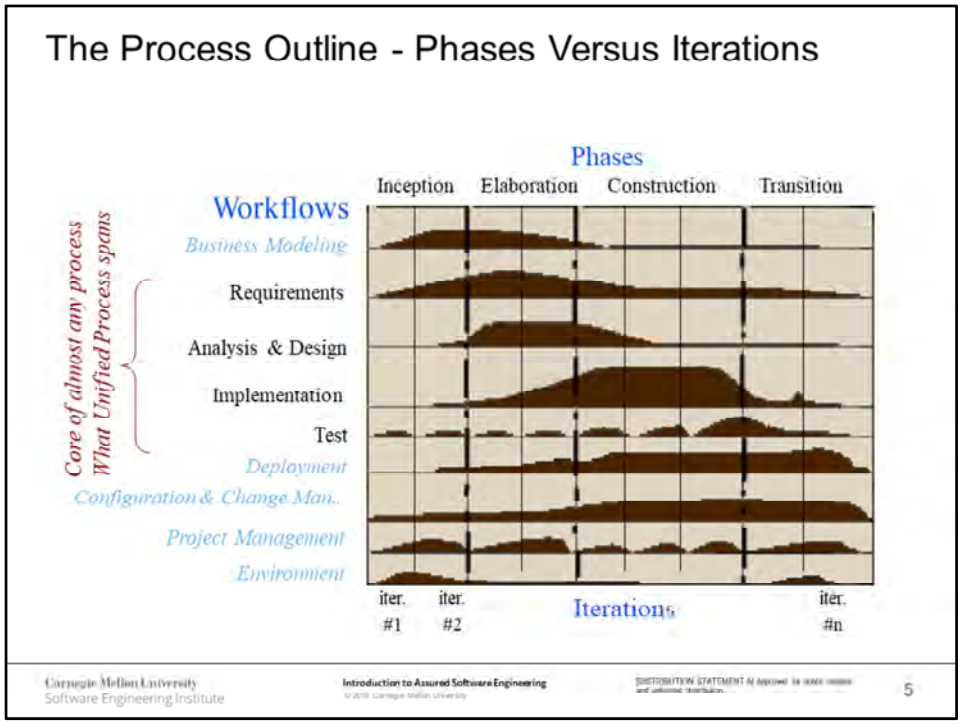
The *sprint backlog* is the list of work the Development Team must address during the next sprint. The list is derived by selecting product backlog items from the top of the product backlog until the Development Team feels it has enough work to fill the sprint.

The *sprint burn down chart* is a publicly displayed chart showing remaining work in the sprint backlog.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 7:
Rational, Agile, and Open Unified Processes (RUP, AUP, OUP)
(Developed by David Root)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



The Rational Unified Process (RUP) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is not a single concrete prescriptive process, but rather an adaptable process framework, intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs. RUP is a specific implementation of the Unified Process.



RUP is based on a set of building blocks and content elements, describing what is to be produced, the necessary skills required and the step-by-step explanation describing how specific development goals are to be achieved. The main building blocks, or content elements, are the following:

- Roles (who) – A Role defines a set of related skills, competencies and responsibilities.
- Work Products (what) – A Work Product represents something resulting from a task, including all the documents and models produced while working through the process.
- Tasks (how) – A Task describes a unit of work assigned to a Role that provides a meaningful result.

Within each iteration, the tasks are categorized into nine disciplines:

Six "engineering disciplines"

- Business Modeling
- Requirements
- Analysis and Design
- Implementation
- Test
- Deployment

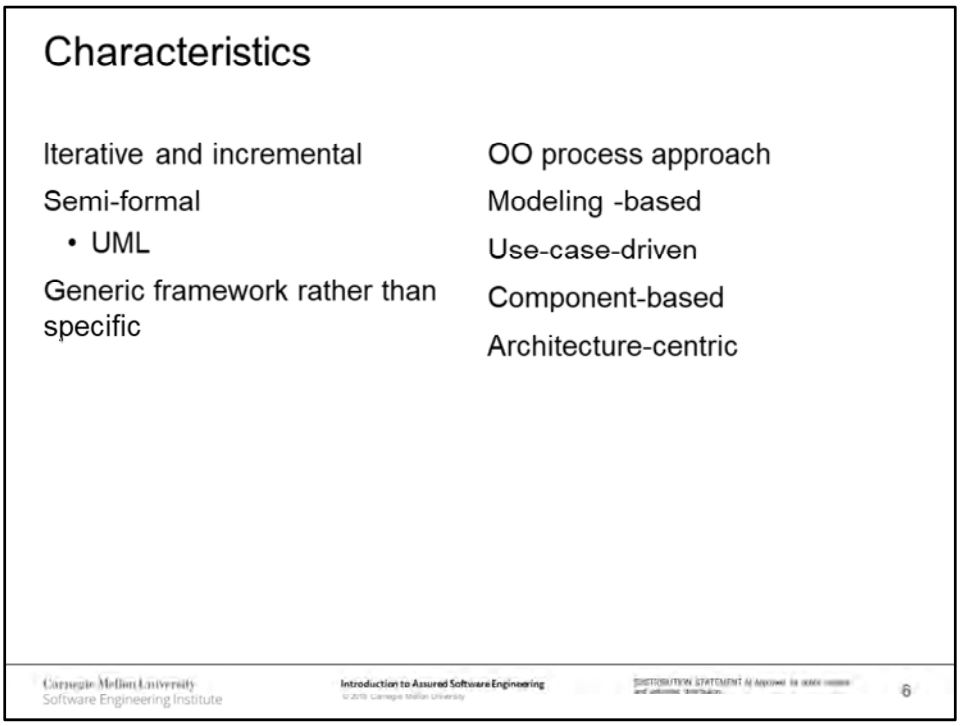
Three supporting disciplines

- Configuration and Change Management
- Project Management
- Environment

The RUP has determined a project life cycle consisting of four phases. These phases allow the process to be presented at a high level in a similar way to how a 'waterfall'-styled project might be presented, although in essence the key to the process lies in the iterations of development that lie within all of the phases. Also, each phase has one key objective and milestone at the end that denotes the objective being accomplished. The visualization of RUP phases and disciplines over time is referred to as the RUP hump chart.

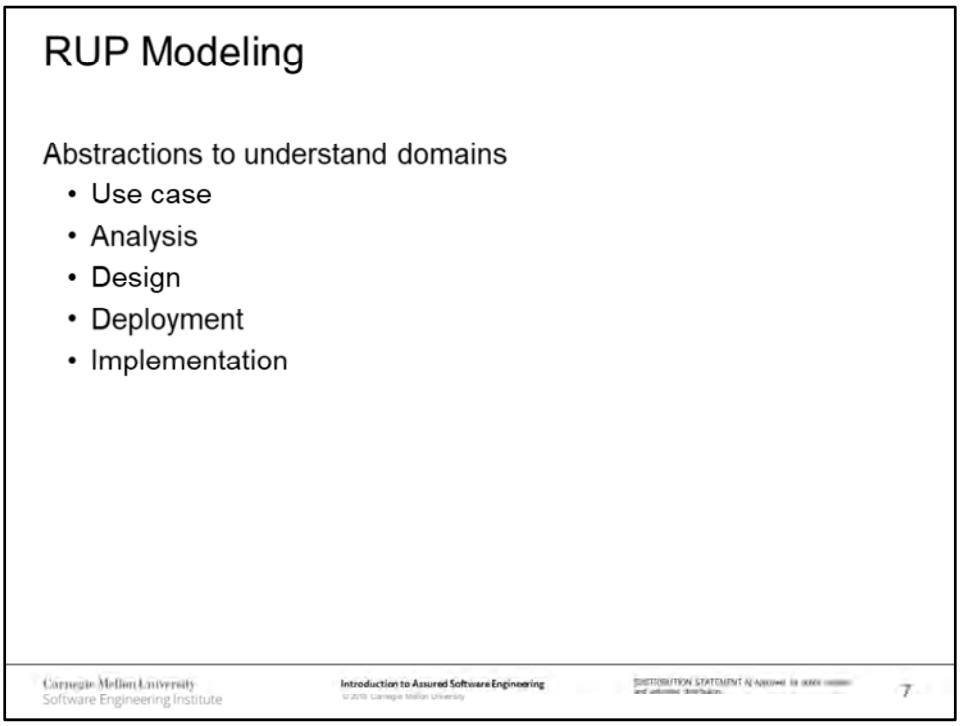
Four project lifecycle phases

1. Inception Phase: Stakeholders, Requirements understanding, cost/schedule estimates, architectural prototype, compare actual expenditures versus planned expenditures.
2. Elaboration Phase: Most of the use-case descriptions are developed (80%), software architecture, development plan for the overall project, Business case and risk list.
3. Construction Phase: Build the software system, Software Release.
4. Transition Phase: Move from development to production, training, testing



The Rational Unified Process describes how to effectively deploy commercially proven approaches to software development for software development teams. These are called “best practices” not so much because you can precisely quantify their value, but rather, because they are observed to be commonly used in industry by successful organizations. The Rational Unified Process provides each team member with the guidelines, templates and tool mentors necessary for the entire team to take full advantage of among others the following best practices:

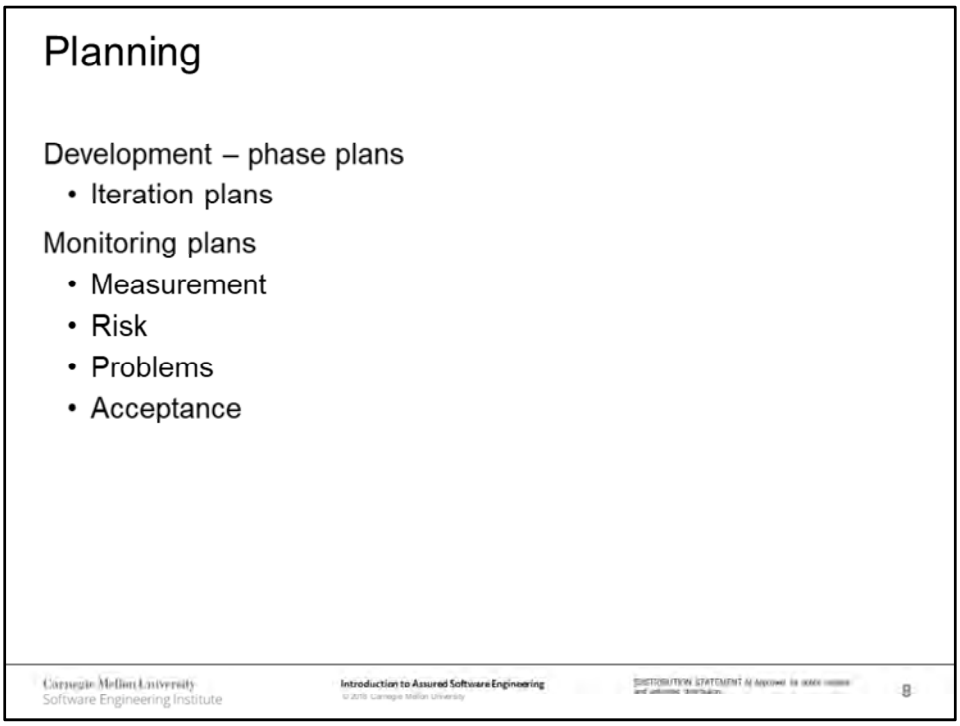
1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software



RUP activities create and maintain models. Rather than focusing on the production of large amount of paper documents, the Unified Process emphasizes the development and maintenance of models—semantically rich representations of the software system under development.

One of the major problems with most business engineering efforts, is that the software engineering and the business engineering community do not communicate properly with each other. This leads to the output from business engineering is not being used properly as input to the software development effort, and vice-versa. The Rational Unified Process addresses this by providing a common language and process for both communities, as well as showing how to create and maintain direct traceability between business and software models.

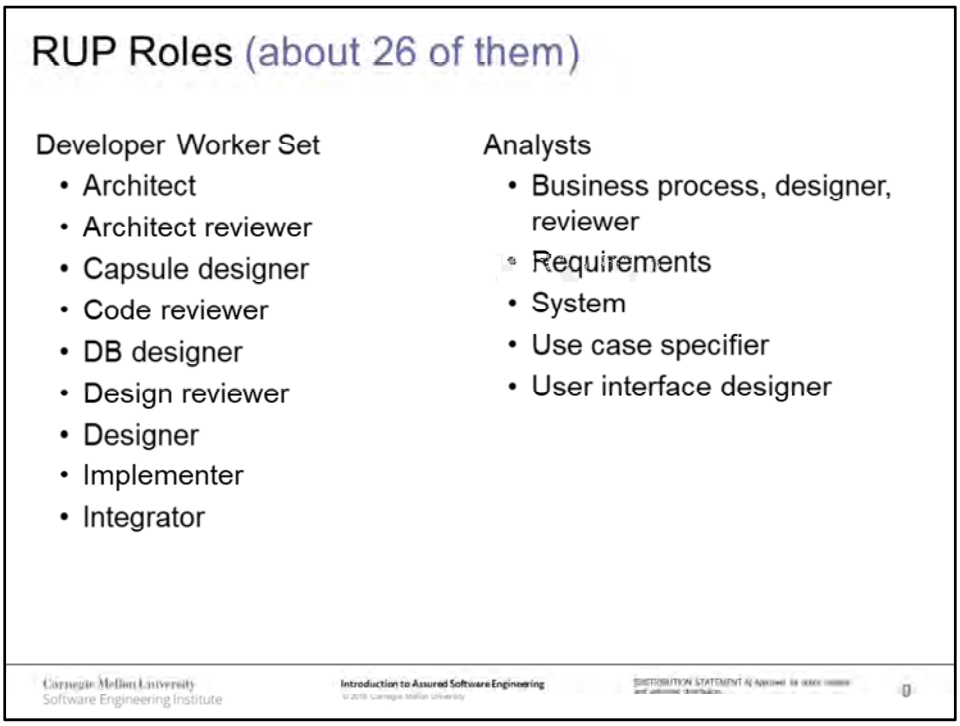
In Business Modeling we document business processes using so called business use cases. This assures a common understanding among all stakeholders of what business process needs to be supported in the organization. The business use cases are analyzed to understand how the business should support the business processes. This is documented in a business object-model. Many projects may choose not to do business modeling.



Software Project Management is the art of balancing competing objectives, managing risk, and overcoming constraints to deliver, successfully, a product in which meets the needs of both customers (the payers of bills) and the users. The fact that so few projects are unarguably successful is comment enough on the difficulty of the task.

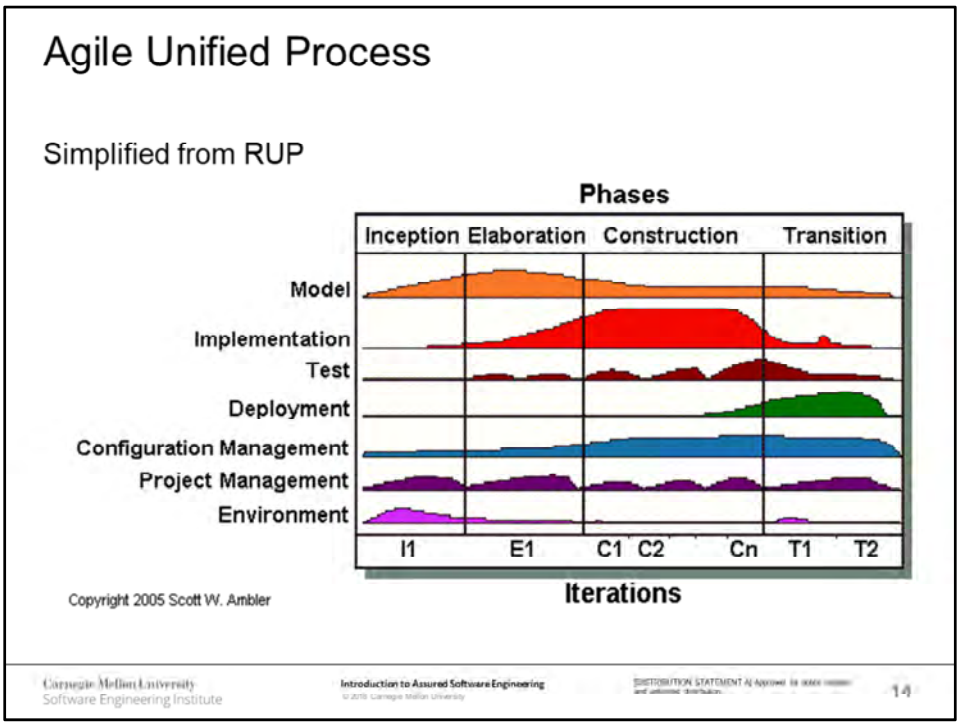
RUP provides:
• A framework for managing software-intensive projects.
• Practical guidelines for planning, staffing, executing, and monitoring projects.
• A framework for managing risk.

It is not a recipe for success, but it presents an approach to managing the project that will markedly improve the odds of delivering successful software.

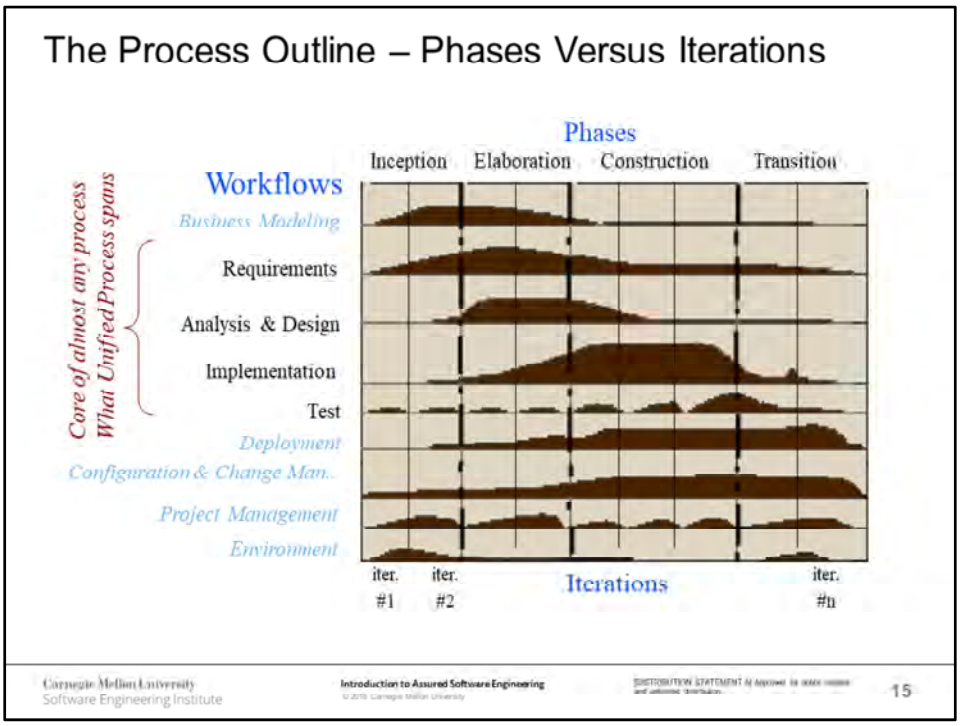


RUP role definitions are consistent with the notion of separating breadth and depth. Personality types for breadth workers and depth workers are very different. Breadth work is quick, inexact, and resilient. Depth work takes much more time, requires attention to detail, and must be of significantly better quality.

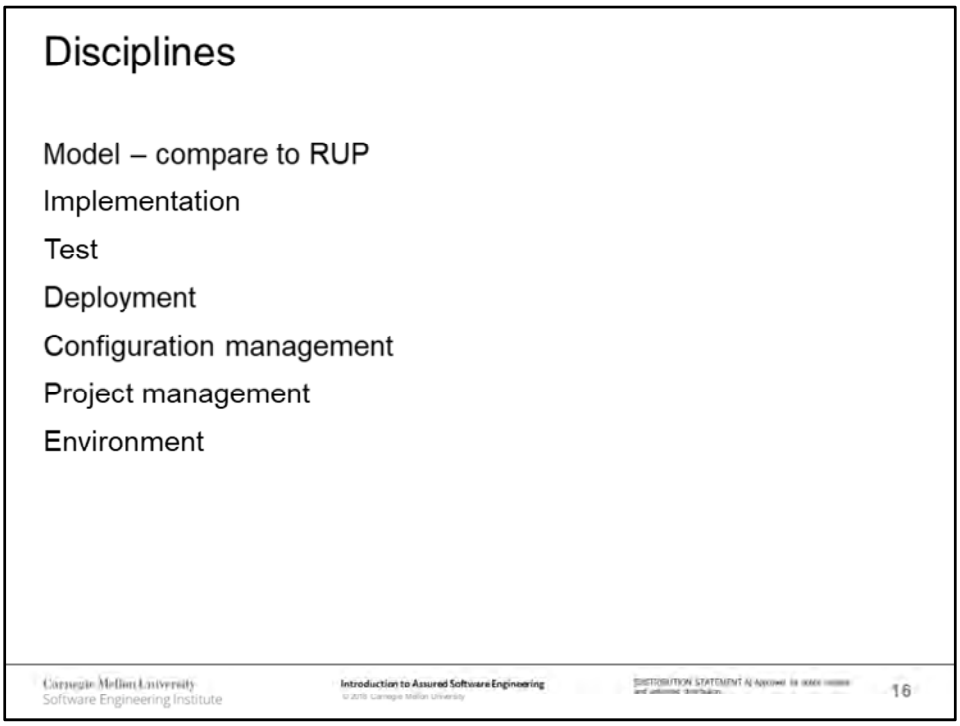
For example for requirements:
Breadth role is the Requirements Systems Analyst who discovers all requirement use cases.
Depth role is the Requirements Engineer who details a single set of requirement use cases.



Agile Unified Process (AUP) is a simplified version of the IBM Rational Unified Process (RUP) developed by Scott Ambler. It describes a simple, easy to understand approach to developing business application software using agile techniques and concepts yet still remaining true to the RUP. The AUP applies agile techniques including test driven development (TDD), Agile Modeling, Agile change management, and database refactoring to improve productivity. *Wikipedia*



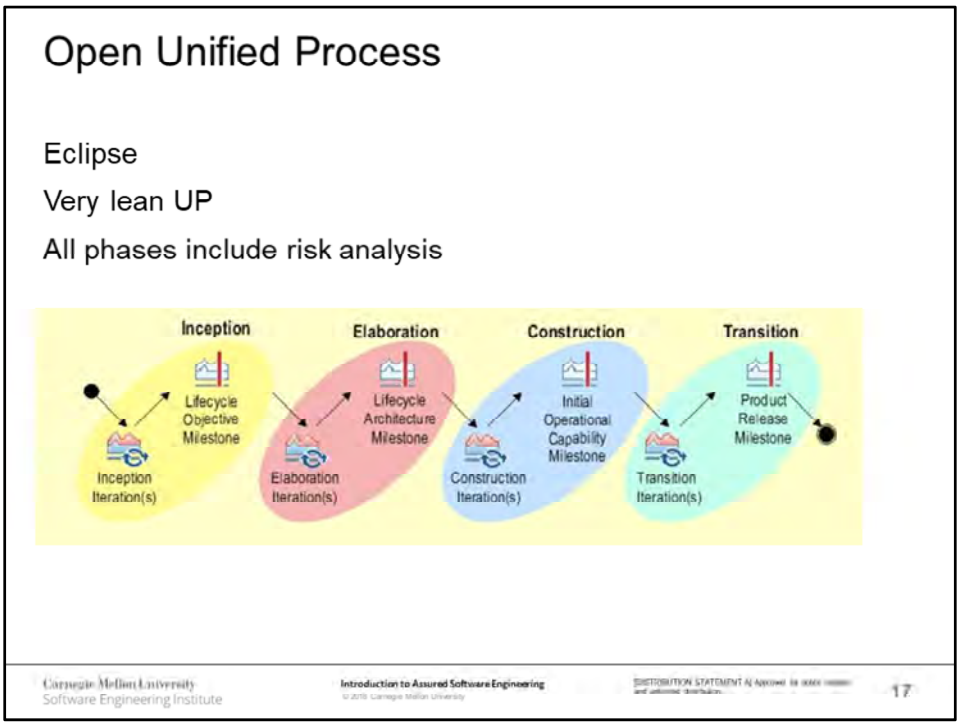
The Agile Unified Process distinguishes between two types of iterations. A development release iteration results in a deployment to the quality-assurance and/or demo area. A production release iteration results in a deployment to the production area. This is a significant refinement to RUP.



AUP has seven disciplines:

1. Model. Understand the business of the organization, the problem domain being addressed by the project, and identify a viable solution to address the problem domain.
2. Implementation. Transform model(s) into executable code and perform a basic level of testing, in particular unit testing.
3. Test. Perform an objective evaluation to ensure quality. This includes finding defects, validating that the system works as designed, and verifying that the requirements are met.
4. Deployment. Plan for the delivery of the system and to execute the plan to make the system available to end users.
5. Configuration Management. Manage access to project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them.
6. Project Management. Direct the activities that take place within the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.), and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget.
7. Environment. Support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.

Wikipedia

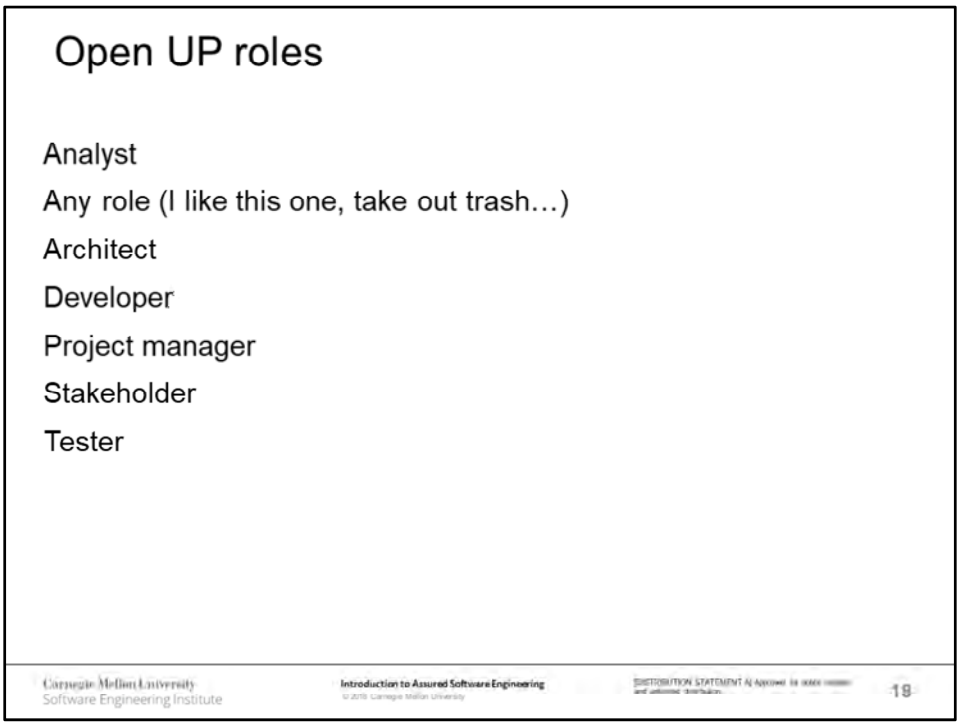


The Open Unified Process (OpenUP) is a part of the Eclipse Process Framework (EPF), an open source process framework developed within the Eclipse Foundation. Its goals are to make it easy to adopt the core of RUP. OpenUP preserves the essential characteristics of RUP, which include iterative development, use cases and scenarios driving development, risk management, and architecture-centric approach. Most optional parts of RUP have been excluded, and many elements have been merged. The result is a much simpler process that is still true to RUP principles.

OpenUP targets small and collocated teams interested in agile and iterative development. Small projects constitute teams of 3 to 6 people and involve 3 to 6 months of development effort. Wikipedia

OpenUP is driven by the four core principles listed below that support a statement in the Agile Manifesto (slides 72-73). Principles capture the general intentions behind a process and create the foundation for interpreting roles and work
products, and for performing tasks:

- Collaborate to align interests and share understanding. This principle promotes practices that foster a healthy team environment, enable collaboration and develop a shared understanding of the project.
- Balance competing priorities to maximize stakeholder value. This principle promotes practices that allow project participants and stakeholders to develop a solution that maximizes stakeholder benefits, and is compliant with constraints placed on the project.
- Focus on the architecture early to minimize risks and organize development. This principle promotes practices that allow the team to focus on architecture to minimize risks and organize development.
- Evolve to continuously obtain feedback and improve. This principle promotes practices that allow the team to get early and continuous feedback from stakeholders, and demonstrate incremental value to them.



Analyst represents customer and end-user concerns by gathering input from stakeholders to understand the problem to be solved and by capturing and setting priorities for requirements.

Any Role represents anyone on the team that can perform general tasks.

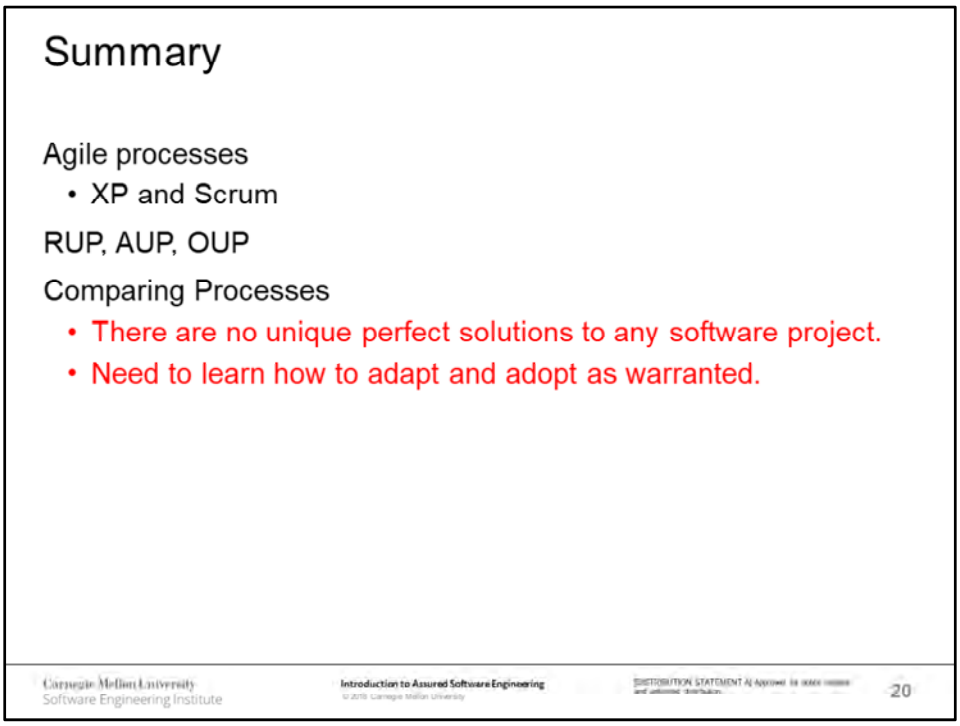
Architect is responsible for designing the software architecture, which includes making the key technical decisions that constrain the overall design and implementation of the project.

Developer is responsible for developing a part of the system, including designing it to fit into the architecture, and then implementing, unit-testing, and integrating the components that are part of the solution.

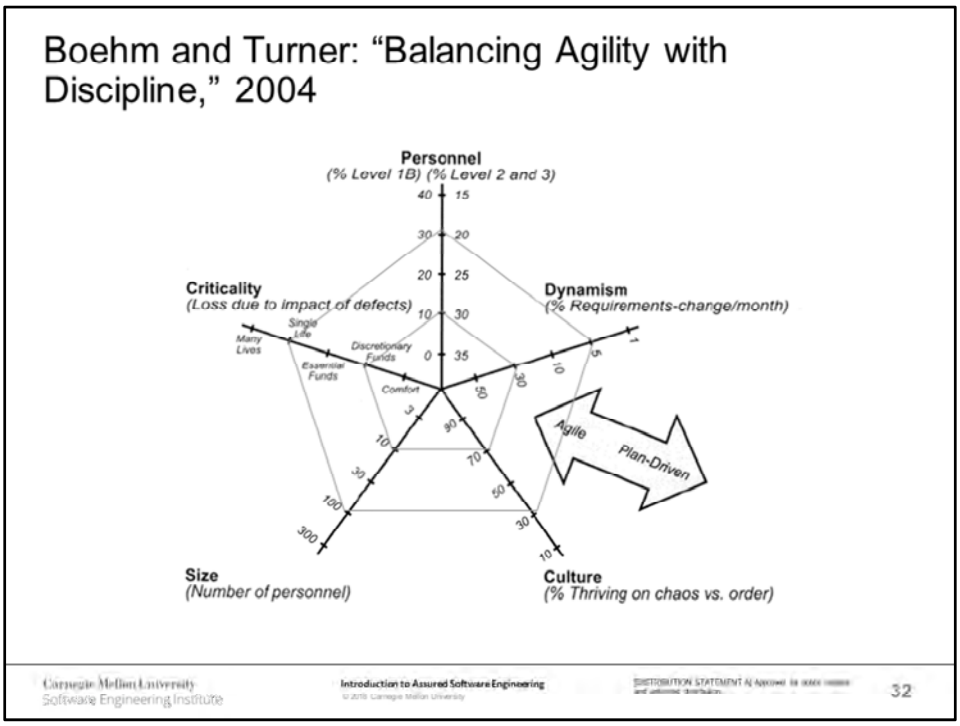
Project Manager leads the planning of the project in collaboration with stakeholders and team, coordinates interactions with the stakeholders, and keeps the project team focused on meeting the project objectives.

Stakeholder represents interest groups whose needs must be satisfied by the project. It is a role that may be played by anyone who is (or potentially will be) materially affected by the outcome of the project.

Tester is responsible for the core activities of the test effort, such as identifying, defining, implementing, and conducting the necessary tests, as well as logging the outcomes of the testing and analyzing the results.



Summarize the main points of the talk.
Make sure that the summary is aligned with the lesson objectives.

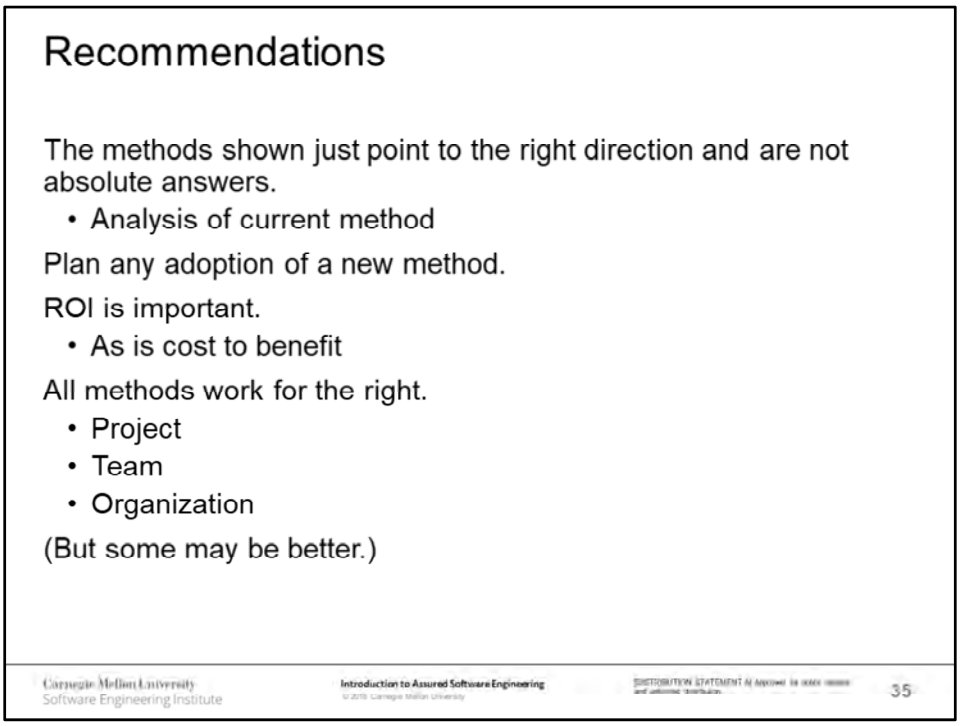


Boehm and Turner concluded that there are five critical factors involved in determining the relative suitability of agile or plan-drive methods in a particular project situation. They are

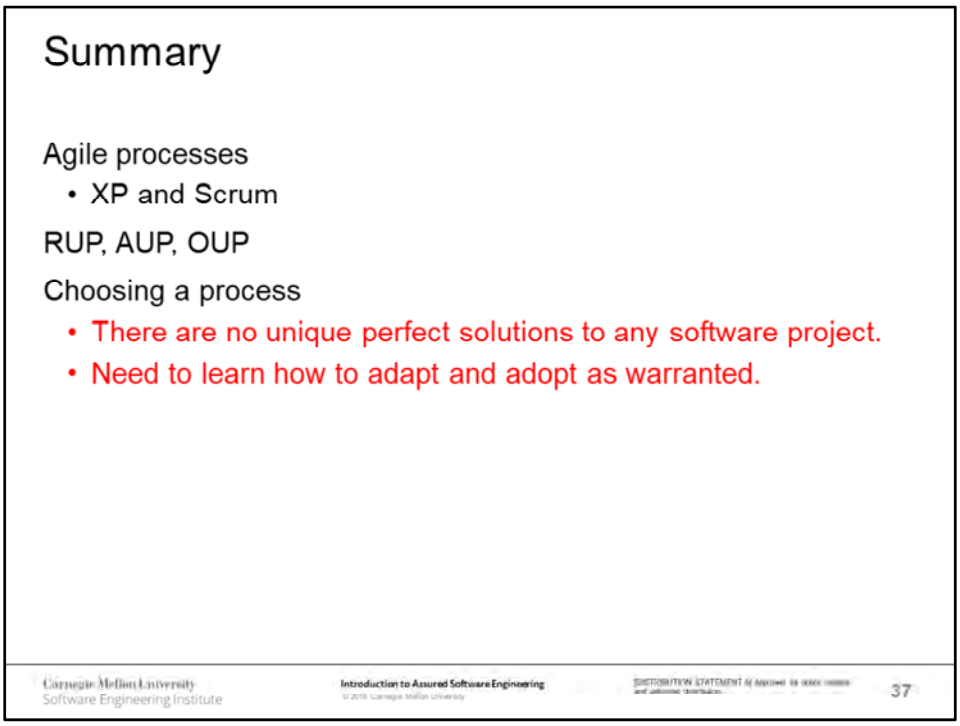
1. size
2. criticality
3. dynamism
4. personnel
5. culture

Size and criticality distinguish between the lighter-weight (towards center) and heavier-weight (towards periphery) methods. The culture axis reflects the reality that agile methods will succeed better in a culture that "thrives on chaos" than one that "thrives on order" and vice versa. The other two axes are asymmetrical in that both agile and plan-driven methods are likely to succeed at one end, and only one of them is likely to succeed at the other. For dynamism, agile methods are comfortable with both high and low rates of change, but plan-driven methods prefer low rates of change. For the personnel scale, plan-driven methods can work well with both high and low skill levels, agile methods require a richer mix of higher level skills.

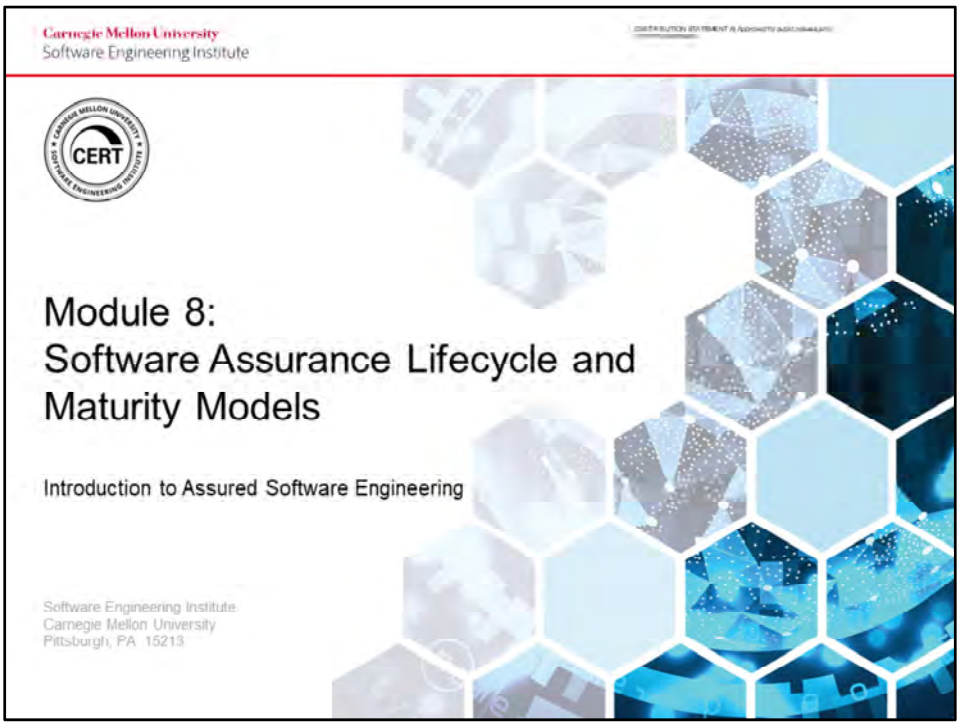
By rating a project along each of the five axes, you can visibly evaluate its home ground relationships. If all the ratings are near the center, agile may be best. If they are near the periphery, you may best succeed with a plan-driven method.



There is no “one” right method for an organization. In fact, most organizations support several methods and it will depend on the projects characteristics. When using a method, it is important that people are trained on the method and they understand how it will be implemented on that project.



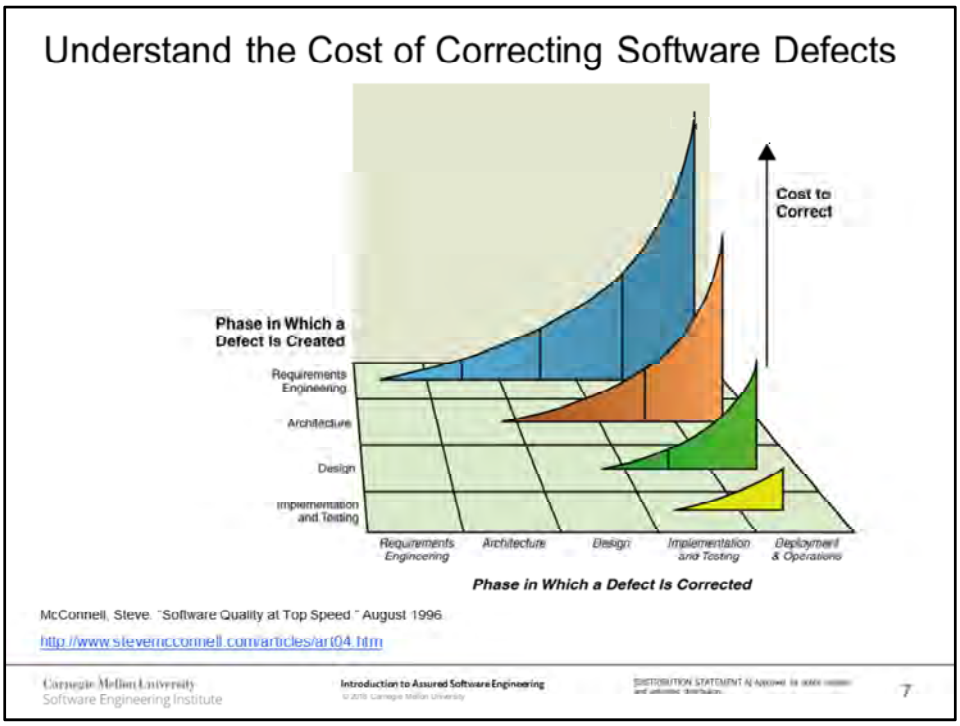
Summarize the main points of the talk.
Make sure that the summary is aligned with the lesson objectives.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 8:
Software Assurance Lifecycle and Maturity Models
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Richard Bejtlich has a humorous description of the security most organizations are practicing. He refers to it as "Soccer Goal Security."

"I see the goalie as representing most preventative security countermeasures. Player 9 is the threat. The soccer ball is an exploit. They are attacking an enterprise, represented by the soccer net. The goalie is addressing the threat he expects, namely someone trying to score from the side of the net he is defending. In many cases the goalie is fighting the last war; perhaps the last time he was scored upon came from the side he now defends?

The threat is smart and unpredictable, attacking a different part of the net. The net itself (the enterprise) is huge. Not only is the front of the net open, the net itself is riddled with holes. A particularly clever attacker might see his objective as getting the ball in the net using any means necessary. That might include cutting the ball into smaller pieces and sending the fragments through holes in the net. Another attacker might dig his way under the goal and send the ball up through a tunnel. Yet another attacker might wait for the goalie to get tired, or drop his guard, or lose his vision at night. A really vicious threat would attack the goalie himself."

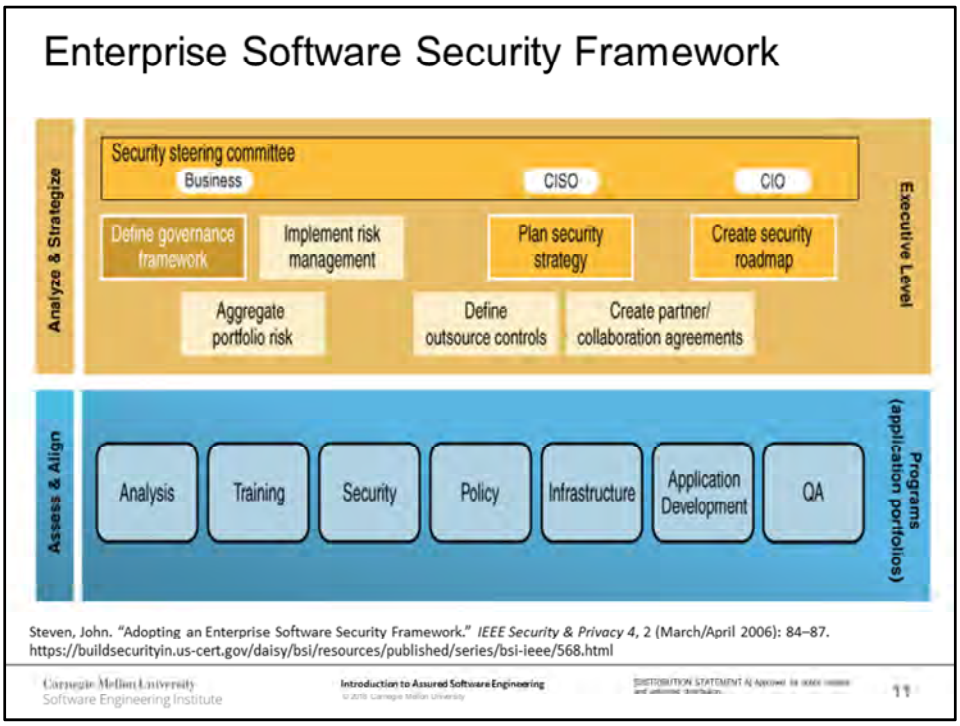


[Soo Hoo 01. "Tangible ROI through Secure Software Engineering]

"Findings indicate that significant cost savings and other advantages are achieved when security analysis and secure engineering practices are introduced early in the development cycle. The return on investment ranges from 12 percent to 21 percent, with the highest rate of return occurring when analysis is performed during application design." [security analysis costs, defects found, vuls fixed, cost to fix by phase]

"According to software quality assurance empirical research, one dollar required to resolve an issue during the design phase grows into 60 to 100 dollars to resolve the same issue after the application has shipped."

"Since nearly three-quarters of security-related defects are design issues that could be resolved inexpensively during the early stages, a significant opportunity for cost savings exists when secure software engineering principles are applied during design."

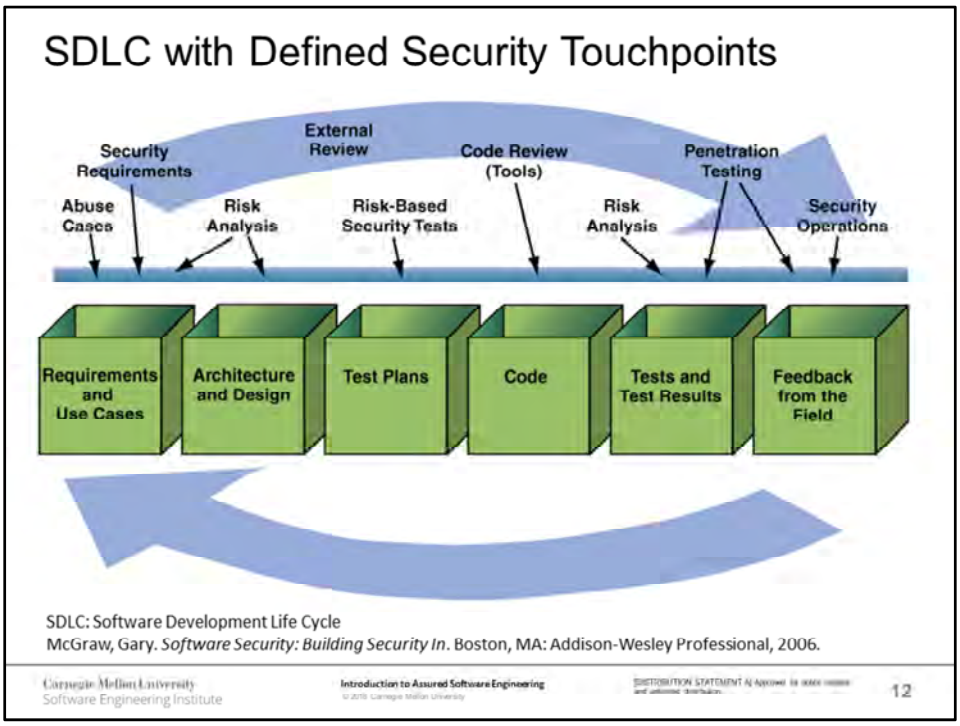


Even the best efforts have met considerable resistance because the problem is mostly organizational and cultural, not technical.

The purpose of the ESSF is to organize everybody's responsibility for achieving secure software into a "who, what, when" structure – describe what activities each role is responsible for and at what point the activity should be conducted. Focus on who needs to do what and when.

1. Assess the organization's current software development and security strengths and weaknesses (s/w built in-house, outsourced, COTS, vendor solution)
2. Don't start off with low hanging fruit (like a vul analysis tool) or expect developers to add practices that they haven't been budgeted for
3. Like IS, need executive sponsorship, clear roles, responsibilities, and clear objectives that tie to business requirements.
4. Don't use just network security folks. Need to integrate security experts into software development teams.
5. Document technology-specific prescriptive guidance for developers.
6. Don't require teams to begin conducting every activity on day one. Slowly introduce the simplest activities first, then iterate.
7. Establish a goal of improving attack resistance in each activity from its inception.

In the context of ESSF, governance is competency in measuring software-induced risk and supporting an objective decision-making process for remediation and software release.

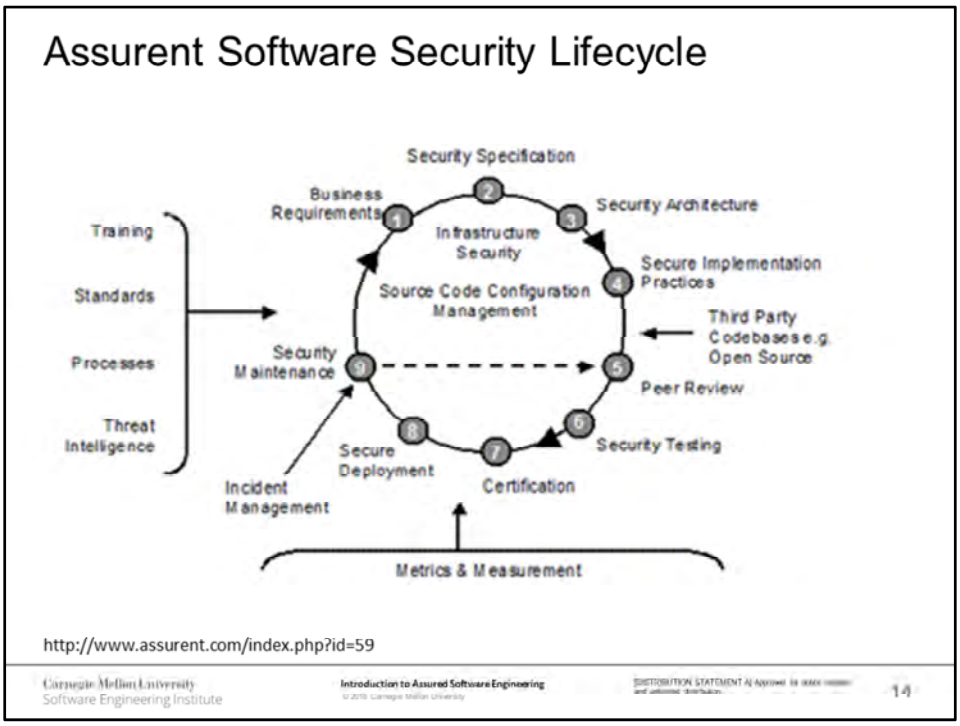


Touchpoints are tasks and activities that augment existing development processes. They are also used for outsource assurance and COTS validation.

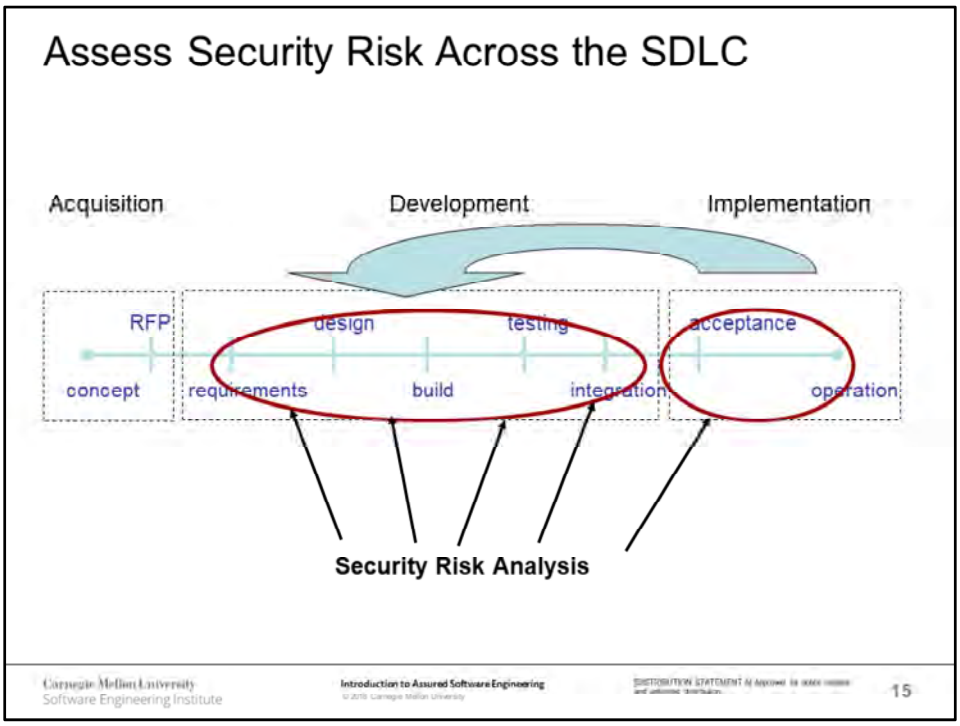
Microsoft's Security Development Lifecycle (SDL)

OWASP (Open Web Application Testing Security Project) Testing Framework [http://www.owasp.org/index.php/The_OWASP_Testing_Framework]

- Phase 1: Before Development Begins
- Phase 2: During Definition and Design
- Phase 3: During Development
- Phase 4: During Deployment
- Phase 5: Maintenance and Operations



Referenced in [Fedchak 07] “Software Project Management for Software Assurance”



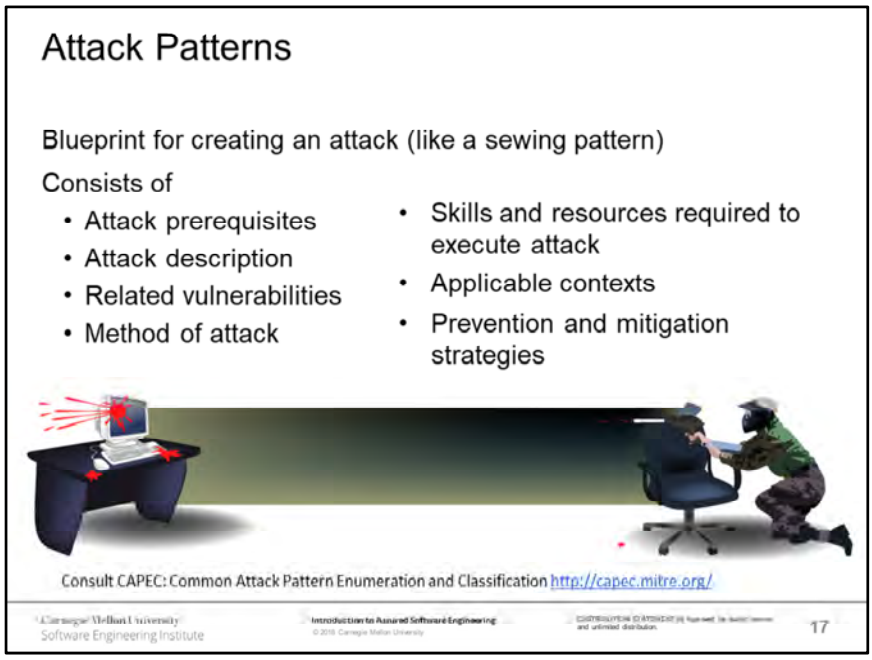
Need to consider both risks to the software and risks to the information that the software processes.

Drive unacceptable risk issues back into the development life cycle.

Systems and software change as the development process progresses – important to revisit the risk assessment to verify that unacceptable risks have not been introduced.

This said, “rarely does an organization use a risk mgmt framework to consistently calculate a risk’s impact at the project mgmt or portfolio level.” so the s/w org needs to make this translation and connection to gain business owner understanding and respect – to choose mitigating security risk over time-to-market, for example [Steven 06]

A candidate checklist to aid in software security risk identification is available in [Fedchak 07] – “Software Project Management for Software Assurance”



[http://capec.mitre.org/] [http://capec.mitre.org/data/index.html]

To be effective, developers need to think outside of the box and to have a firm grasp of the attacker's perspective and the approaches used to exploit software.

Attack patterns are a powerful mechanism to capture and communicate the attacker's perspective. They are descriptions of common methods for exploiting software. They derive from the concept of design patterns applied in a destructive rather than constructive context and are generated from in-depth analysis of specific real-world exploit examples.

One example: Buffer overflow: improper or missing bounds checking. Attacker is able to write past the boundaries of allocated buffer regions in memory; able to crash system or redirect execution

Man in the middle

Motivation data modification, privilege escalation, information leakage

Description This type of attack targets the communication between two components (typically client and server). The attacker places himself in the communication channel between the two components. Whenever one component attempts to communicate with the other (data flow, authentication challenges, etc.), the data first goes to the attacker, who has the opportunity to observe or alter it, and it is then passed on to the other component as if it was never intercepted. This interposition is transparent leaving the two compromised components unaware of the potential corruption or leakage of their communications. The potential for Man-in-the-Middle attacks yields an implicit lack of trust in communication or identify between two components.

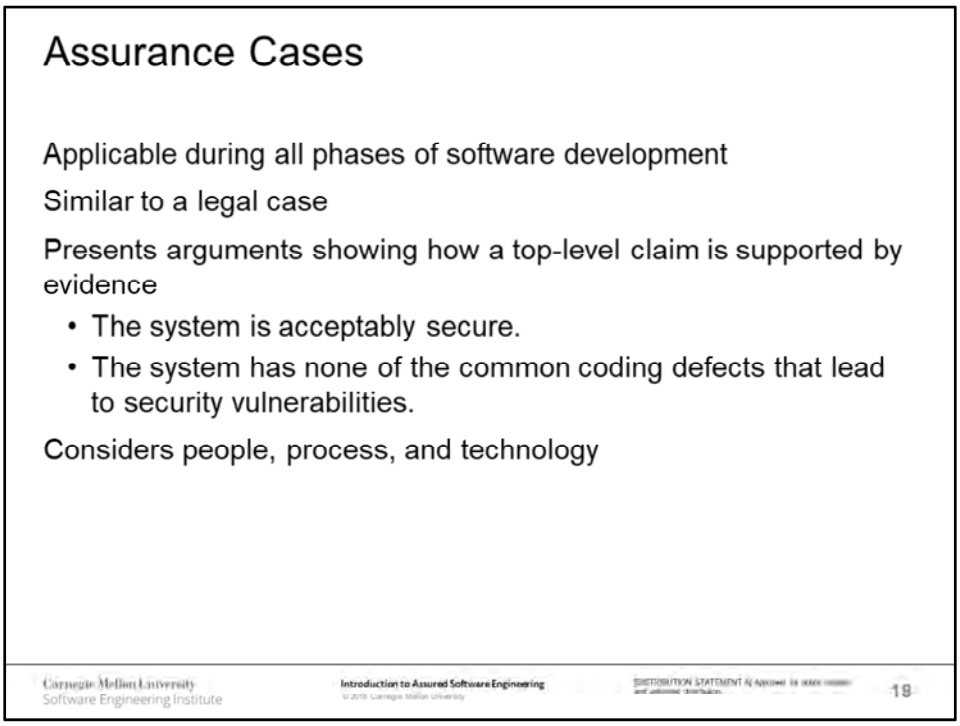
Attack Execution Flow The attacker probes to determine the nature and mechanism of communication between two components looking for opportunities to exploit.

The attacker inserts himself into the communication channel initially acting as a routing proxy between the two targeted components.

The attacker observes, filters or alters passed data of its choosing to gain access to sensitive information or to manipulate the actions of the two target components for his own purposes.

Attack Prerequisites There are two components communicating with each other.An attacker is able to identify the nature and mechanism of communication between the two target components. An attacker can eavesdrop on the communication between the target components. Strong mutual authentication is not used between the two target components yielding opportunity for attacker interposition. The communication occurs in clear (not encrypted) or with insufficient and spoofable encryption.

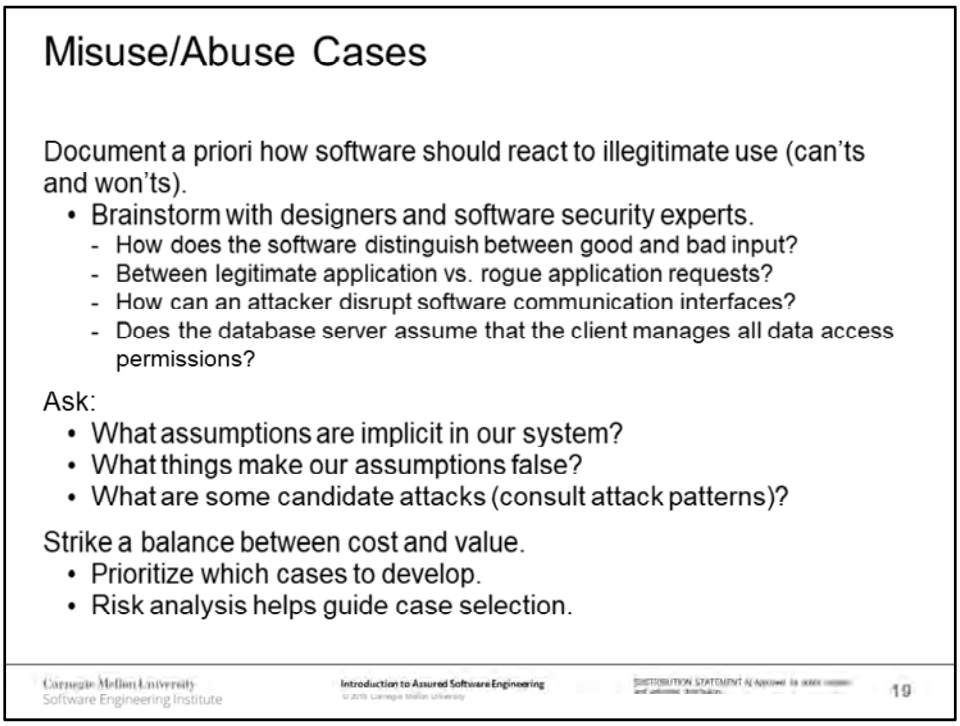
Solutions: encryption, strong mutual authentication



[Fedchak 07]

An assurance case can be a justification for confidence that a software or software-intensive system is secure. One definition of an assurance case is that it is:

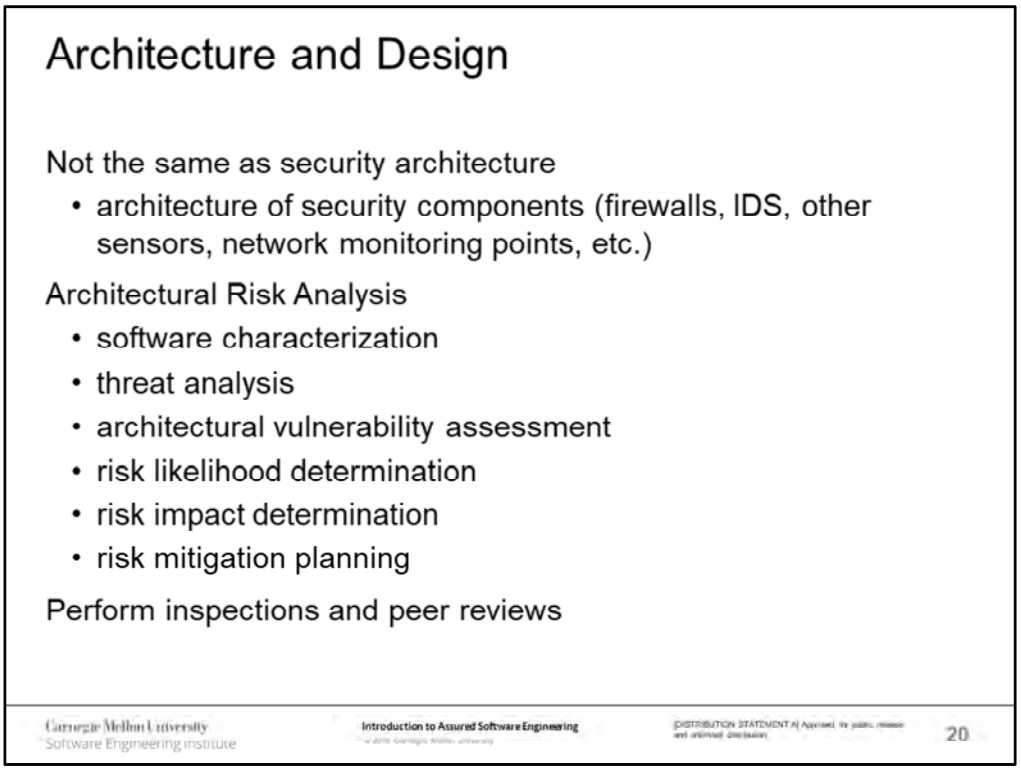
"a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's properties are adequately justified for a given application in a given environment" [Ankrum 2006].



Inclusion of a new requirement or feature includes determining how it might be unintentionally misused or intentionally abused.

For example, assuming that the connection between a Web server and a database server can always be trusted. Attacker will try to make the Web server send inappropriate requests to access valuable data.

Users can't enter more than 50 characters.



Parallels info security risk assessment. An architectural risk assessment validates that security requirements were translated into aspects of the s/w's design and that the design resists attack.

While much of the fanfare of software security today is focused on buffer overflows, SQL injection, and other implementation bugs, the reality is that approximately half of the defects leading to security vulnerabilities found in today's software are actually due to flaws in architecture and design [McGraw 2006].

The goal of building security into the architecture and design phase of the SDLC is to reduce significantly the number of flaws as early as possible while also reducing ambiguities and other weaknesses.

Requires experience, expertise, knowledge; often human/expert intensive. Architectural-level flaws can currently only be found through human analysis.

While architectural risk analysis is not focused primarily on assets, it does depend on the accurate identification of the software's ultimate purpose and how that purpose ties into the business's activities in order to qualify and quantify the impact of risks identified during the process. Because of this, a **solid understanding of the assets that the software guards** or uses should be considered a prerequisite to performing architectural risks analysis.

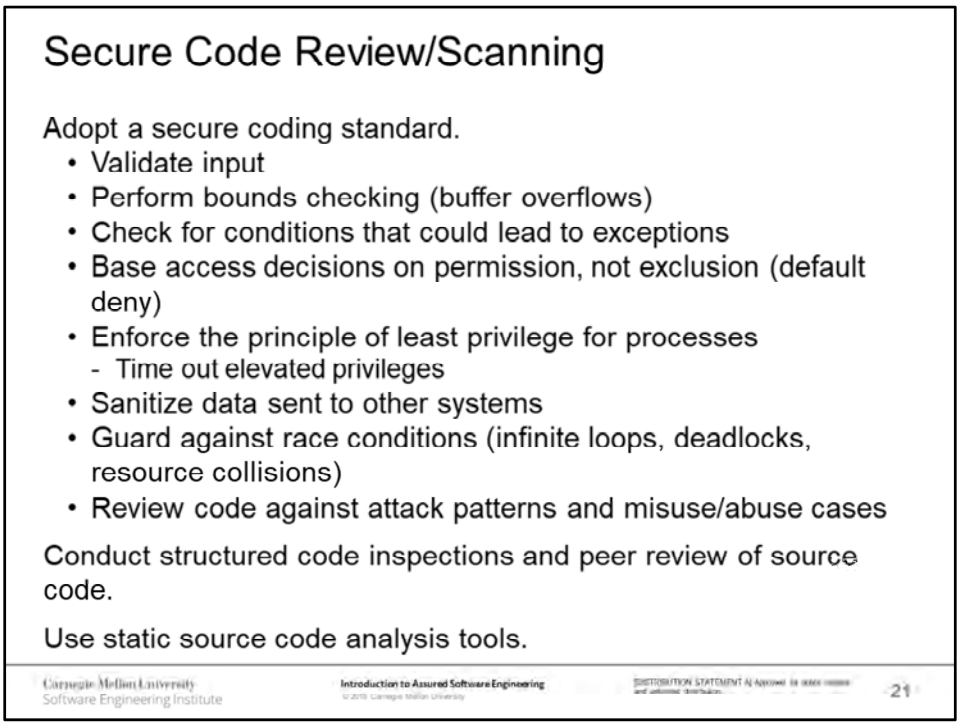
Software characterization: what the software is, how it works, components, interfaces, zones of trust, one page

Threat analysis: identify relevant threats, attacker access/skill level, map to specific vulnerabilities, identify mitigations

Arch vul assessment: attack resistance (uses attack patterns), ambiguity, and dependency analysis. Ambiguity example: ramifications of a user login that persists after the account is locked. Dependency analysis: vuls associated with the s/w's execution environment (OS, network, application platform).

Risk likelihood: threat motivation/capability, vul impact (attractiveness to attacker), effectiveness of current controls. Qualitative high, medium, low

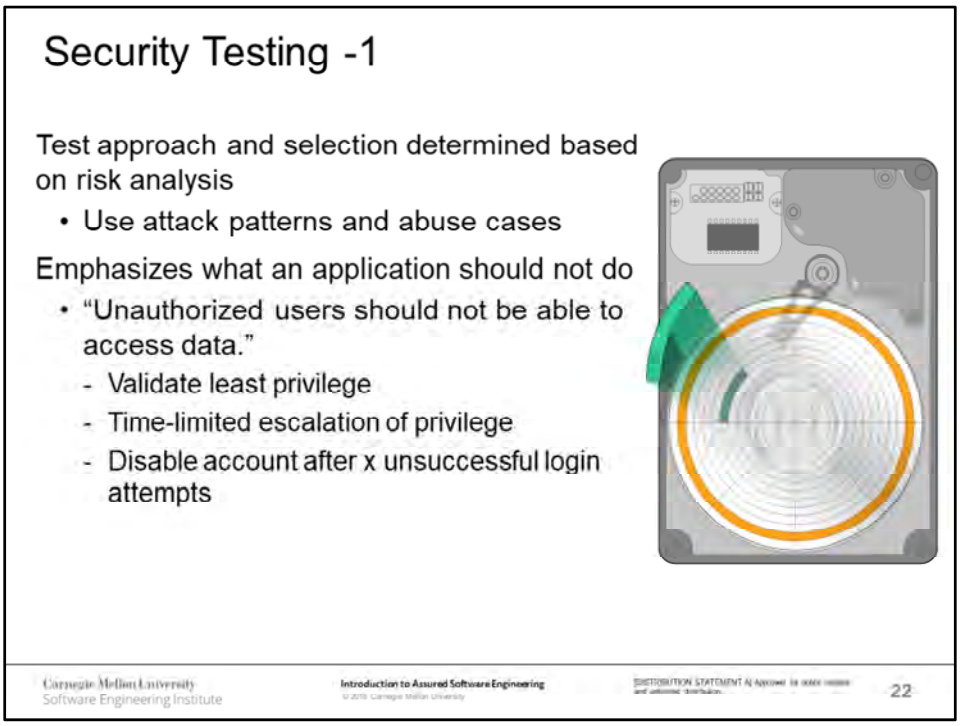
Risk impact: 3x3 matrix of likelihood (high, med, low) vs. impact (high, med, low)



Static code checkers, runtime code checkers, profiling tools, penetration testing tools, stress test tools, and application scanning tools can find some security bugs in code.

[cert.org/secure-coding]

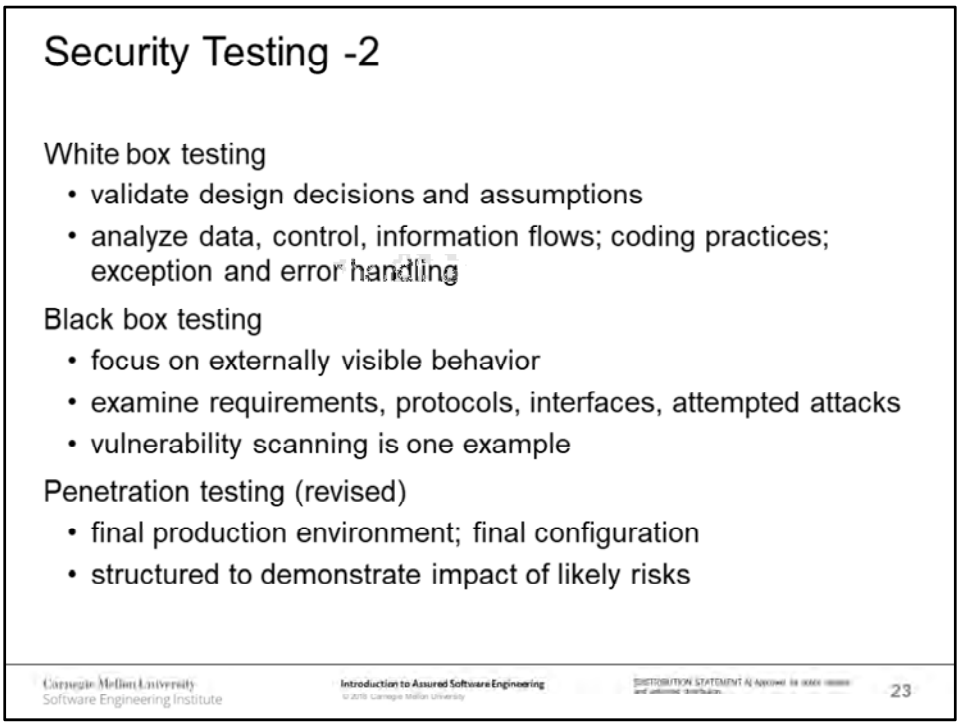
Most vulnerabilities stem from a relatively small number of common programming errors. By identifying insecure coding practices and developing secure alternatives, software developers can take practical steps to reduce or eliminate vulnerabilities before deployment.



One important difference between security testing and other testing activities is that the security test engineer needs to emulate an intelligent attacker. An adversary might do things that no ordinary user would do, such as entering a thousand-character surname or repeatedly trying to corrupt a temporary file. Test engineers must consider actions that are far outside the range of normal activity and might not even be regarded as legitimate tests under other circumstances. A security test engineer must think like the attacker and find the weak spots first.

Security testing is different from traditional software testing in that it emphasizes what an application should not do rather than what it should do. While it sometimes tests conformance to positive requirements such as "user accounts are disabled after three unsuccessful login attempts" and "network traffic must be encrypted," more often it tests negative requirements [Fink 1997] such as "outside attackers should not be able to modify the contents of the Web page" and "unauthorized users should not be able to access data."

[Use of mitigations for negative requirements] For example, the risk of password-cracking attacks can be mitigated by disabling an account after three unsuccessful login attempts, and the risk of SQL injection attacks from a Web interface can be mitigated by using an input validation whitelist (a list of all known good inputs which a system is permitted to accept) that excludes characters used to perform this type of attack.



Feed penetration testing results back into design and development.

The Product Security Office drives the product security programs across EMC that focuses on preventing our products from introducing new risks into our customers' IT infrastructures. We are rolling out these programs across EMC product teams by providing standards, practices and enabling resources that addresses product security at every stage of the product lifecycle, from when the product is being defined to when it is in product and supported for our customers.

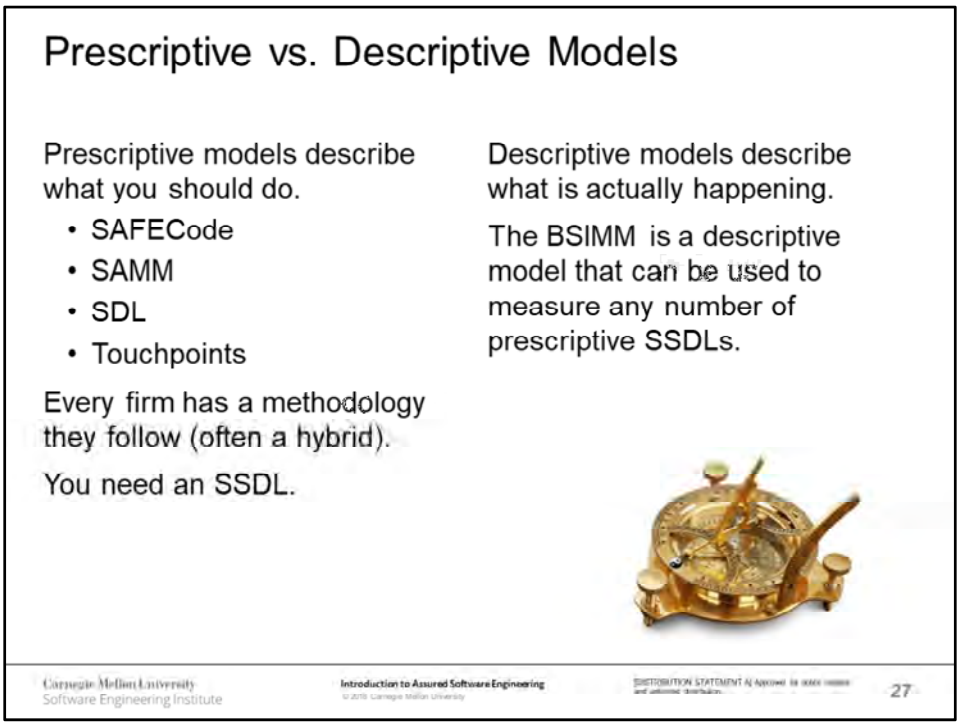
Our **Security Development Lifecycle** focuses on security controls required as part of the product development before the product is released for General Availability.

Our **Security Certification** program focus on assisting and coordinating security certifications such as FIPS 140-2 or Common Criteria that our products are required to support on specific markets

Our **Vulnerability Response** program deals with vulnerabilities impacting our products after they have shipped. It provides a central point for our customer to report vulnerabilities and for EMC products to coordinate the issuance or remedies or security patches that addresses these vulnerabilities

Finally, our **Software Supply Chain Risk Management** Program defines the standards and requirements for product code integrity that govern the inclusion of external components and open source software into our products, the protection of our source code systems and the delivery and support of our products to our customers.

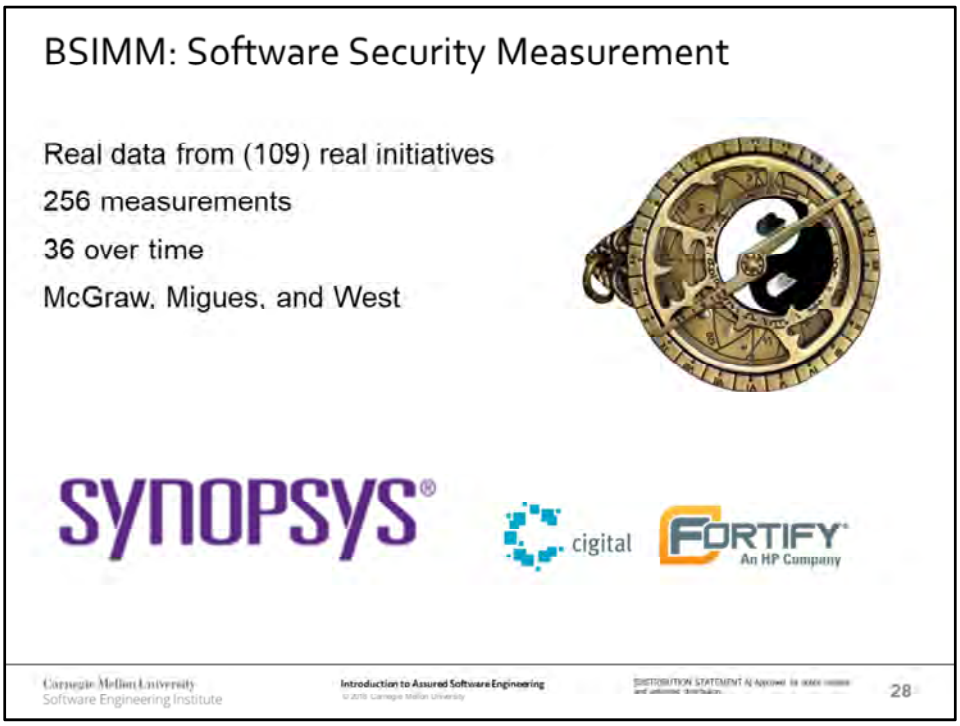
.



There is plenty of confusion (especially in the press) about methodologies and measurement tools. The BSIMM is not a methodology. It is a measurement tool.

The BSIMM8 is used to measure and describe (in common terms) each of the 109 distinct SSDL methodologies in use in the current BSIMM8 Community.

See the InformIT article BSIMM versus SAFECode and Other Kaiju Cinema (Dec 26, 2011) http://bit.ly/tLIOnJ

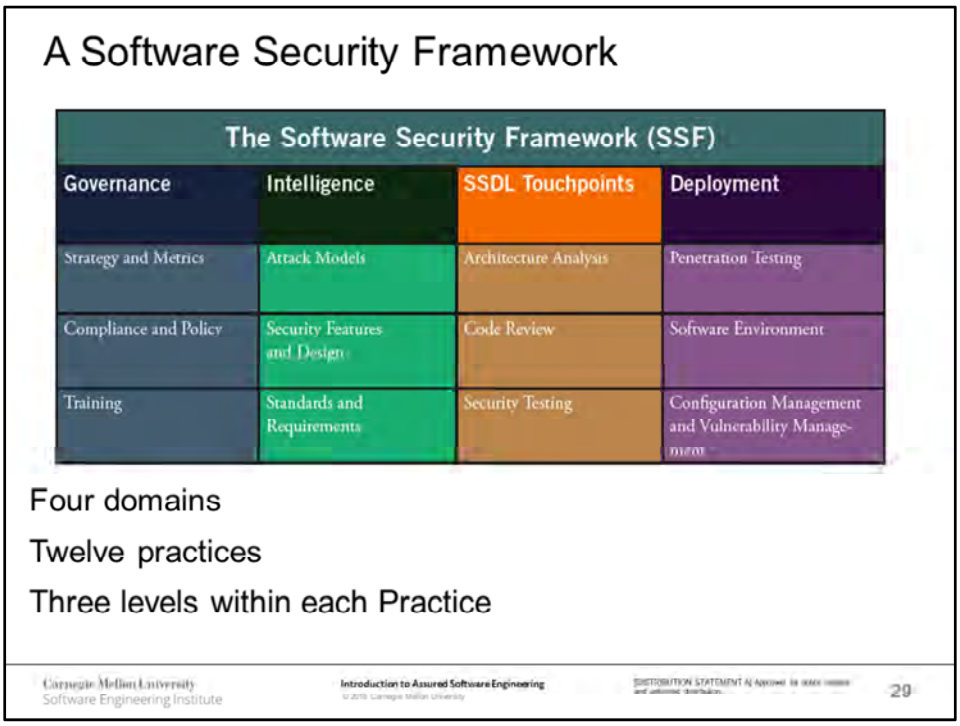


Originally conceived in 2006 by Gary McGraw and Sammy Migues (currently with Synopsis Software Integrity Group, then of Cigital) and Brian Chess (then of Fortify). The first BSIMM was published in 2009 by Synopsis, Inc..

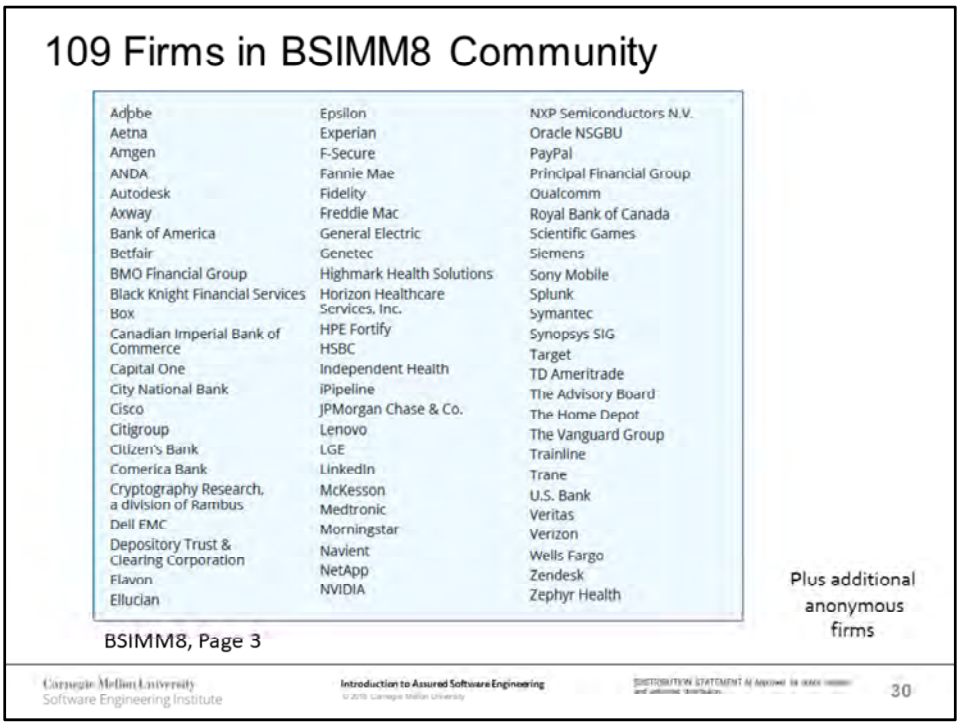
BSIMM8 was released on 9/20/2017

- BSIMM8 includes data from 109 firms in six vertical markets and a longitudinal study. Data gathered from firms during previous 42 months.
- Thirty-six firms were measured twice or more (giving us Longitudinal Study data) and the data show measurable improvement
- BSIMM8 describes 113 activities divided into 12 practices.
- The BSIMM8 data set has 256 distinct measurements (some firms use BSIMM to measure distinct business units and some have been measured more than once).
- BSIMM8 describes the work of 1,268 SSG members working with a satellite of 3051 people to secure the software developed by 290,852 developers as part of a combined portfolio of 94,802 applications.

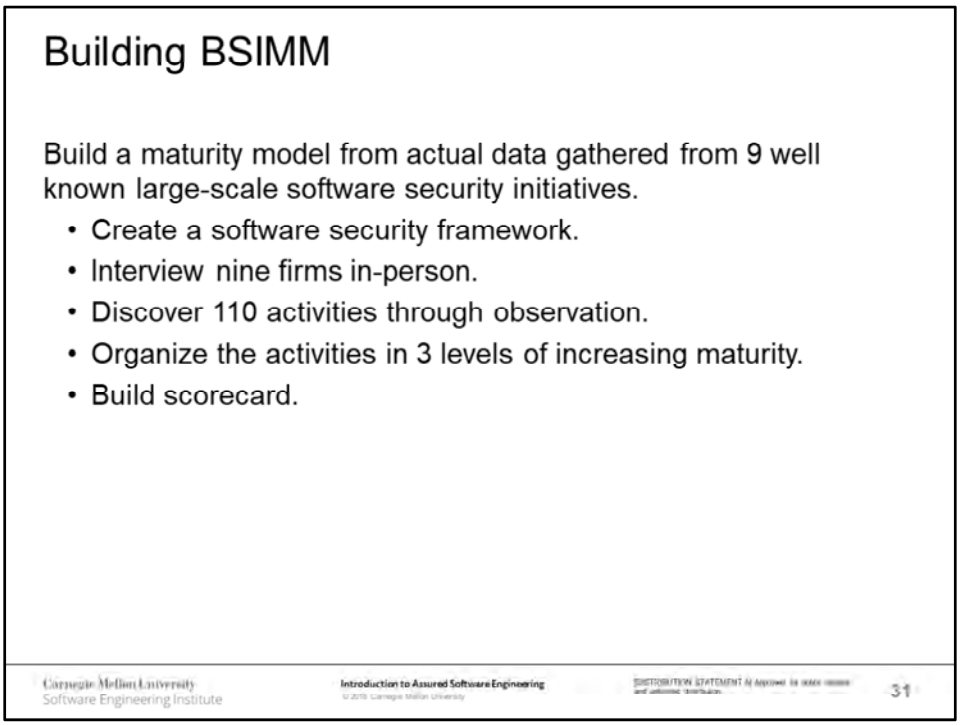
The BSIMM remains the only measuring stick for software security initiatives based on science. It is extremely useful for comparing the initiative of any given firm to a large group of similar firms. The BSIMM has been used by multiple firms to strategize and plan their software security initiatives and measure the results.



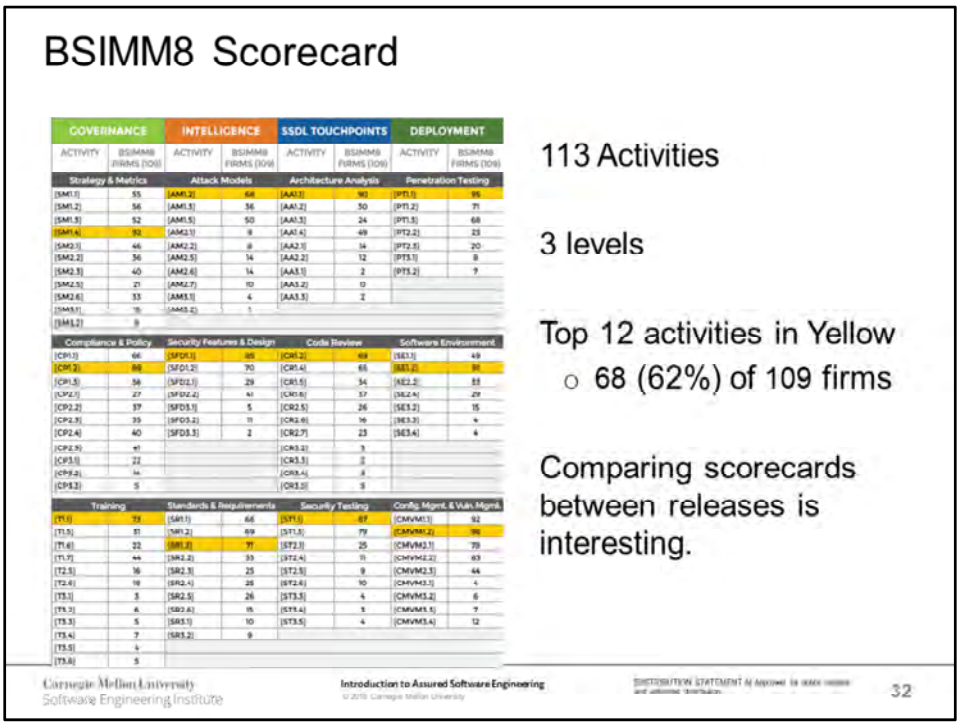
We captured data through interviews, but we needed a place -- sort of an archaeological grid -- to sort the data and make it available to everyone. In the beginning as it were, we simply had the Software Security Framework. Literally, we had just one sheet of paper with that diagram on it and a general understanding of what might go in each bucket.



72 of 109 firms

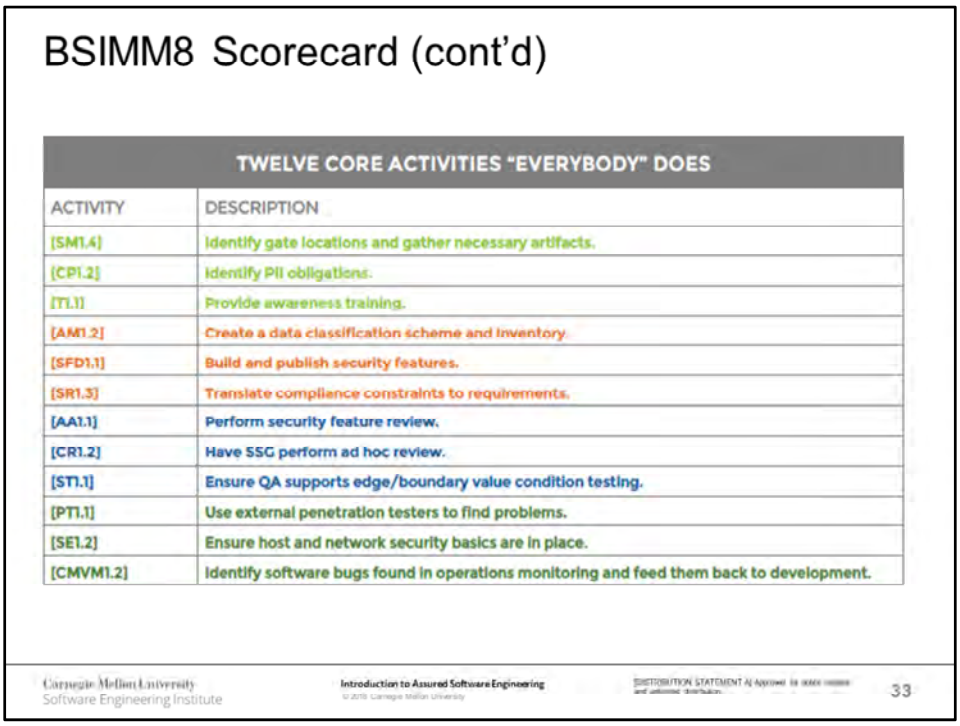


In 2008, we started with interviews of the Software Security Group (SSG) owners at nine firms. We effectively asked each to "Tell me about everything you do to make software security happen" and then gently guided the conversation from there. We didn't ask any yes/no questions and were very diligent about being observers rather than guiding the conversation. We sorted through all the interview data, removed all the duplicates and wishful thinking, and organized the remaining individual activities into the practices of the SSF. There were 110 activities in BSIMM1. These activities are organized into three levels of increasing maturity.



This is the 109 firm raw data about activities.

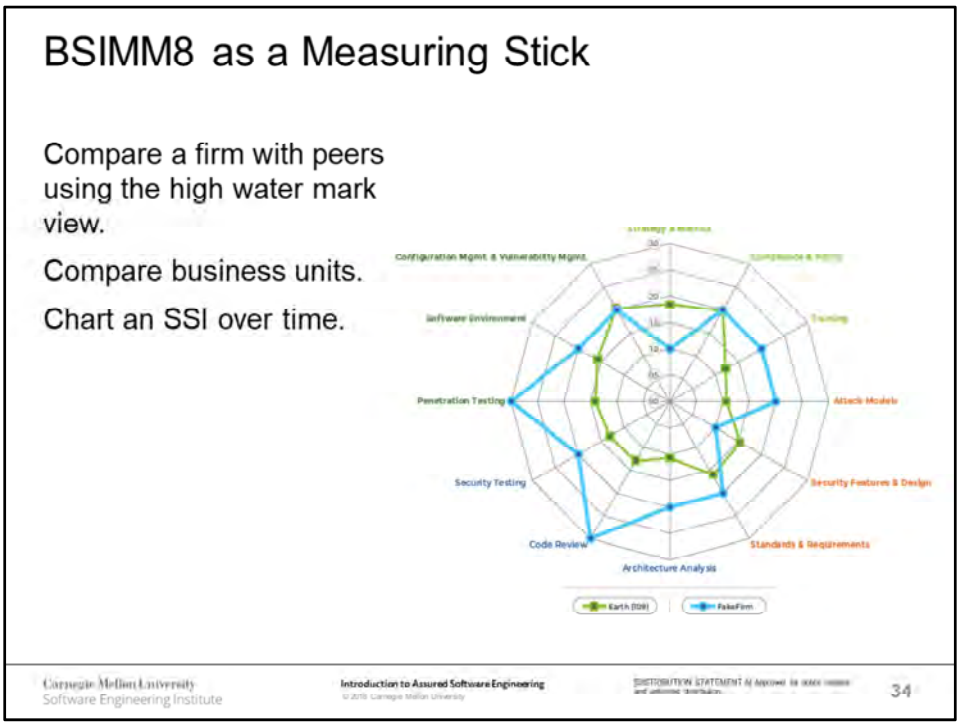
Highlighted in yellow are the top 12 activities performed by 68 (62%) of the 109 firms



BSIMM8 Scorecard (cont'd)

TWELVE CORE ACTIVITIES "EVERYBODY" DOES	
ACTIVITY	DESCRIPTION
[SM1.4]	Identify gate locations and gather necessary artifacts.
[CP1.2]	Identify PII obligations.
[T1.1]	Provide awareness training.
[AM1.2]	Create a data classification scheme and inventory.
[SFD1.1]	Build and publish security features.
[SR1.3]	Translate compliance constraints to requirements.
[AA1.1]	Perform security feature review.
[CR1.2]	Have SSG perform ad hoc review.
[ST1.1]	Ensure QA supports edge/boundary value condition testing.
[PT1.1]	Use external penetration testers to find problems.
[SE1.2]	Ensure host and network security basics are in place.
[CMVM1.2]	Identify software bugs found in operations monitoring and feed them back to development.

More specifics about the top 12 activities performed by 68 (62%) of the 109 firms



This is a comparison of a FAKE firm's highest level of activity observed for each of the 12 practices (a "high water mark" in blue) against an average of high-water marks over a group of firms.

Note where the blue is INSIDE the green. These are practices where the firm is substantially behind what we have observed elsewhere. In general, firms with a "round" curve have a more balanced program than firms with a "prickly" shape or worse yet a "butterfly" shape. Remember, this is not a value judgment, it is simply a comparison to what other firms are doing.

BSIMM8 as a Longitudinal Study

36 firms measured at least twice

Raw score increased in 29 of 36 firms

Observation count increased by 33.4%

"SSI's mature over time"

BSIMM8 includes a longitudinal study of 36 firms which were measured at least twice. The average time between measurements was 25.8 months. According to the study "remeasurement over time shows a clear trend of increased maturity among the 36 firms remeasured thus far. The raw score went up in 29 of the 36 firms. Across all 36 firms, the observation count increased by an average of 10.3 (33.4%). Software security initiatives mature over time."

BSIMM8

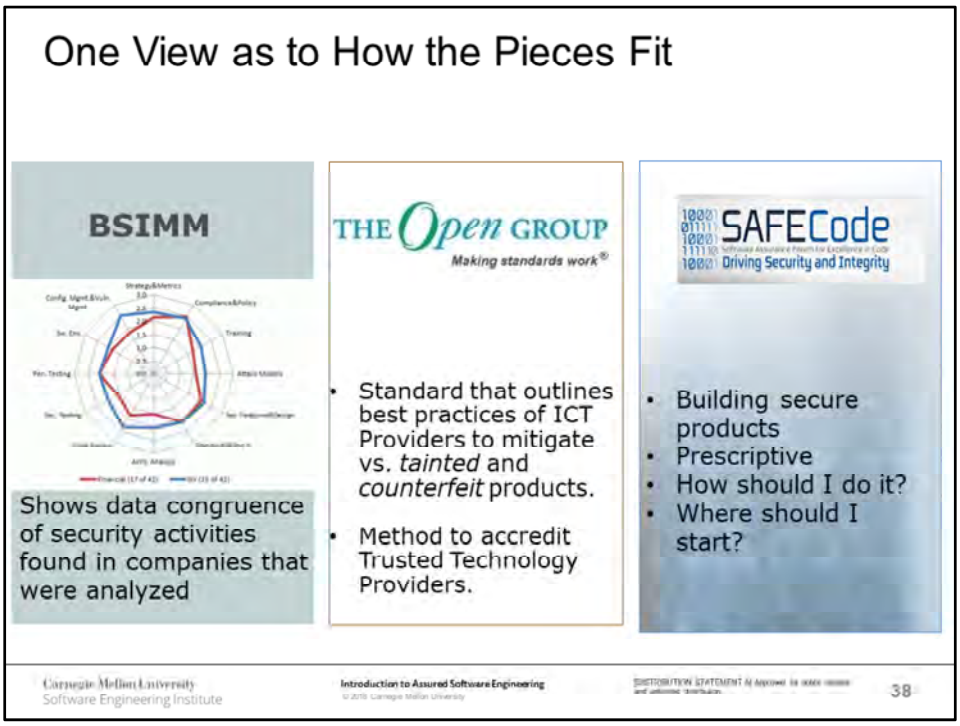
BSIMM8 released September 2017 under creative commons

http://bsimm.com

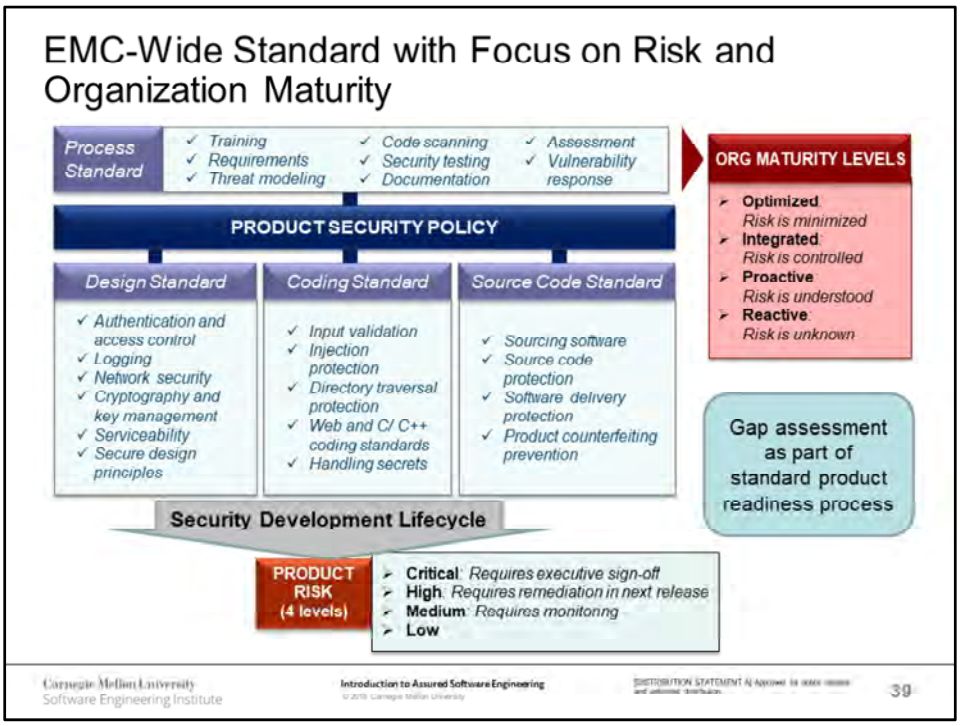
BSIMM is a yardstick.

- Use it to see where you stand.
- Use it to figure out what your peers do.

The state of the BSIMM model as of BSIMM8.



ICT - Information and Communication Technology



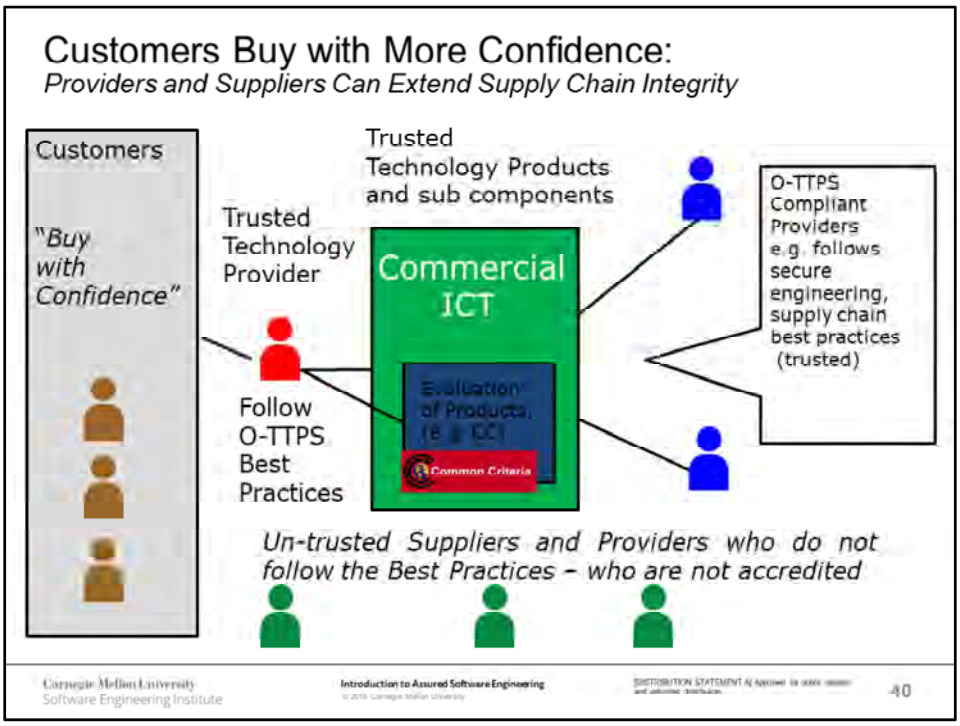
EMC's approach to measuring product security is captured in the Product Security Policy, which is our internal standard that we have mapped to customer requirements and the various regulations such as SOX, HIPAA, PCI DSS, FISMA etc.

3 standards:

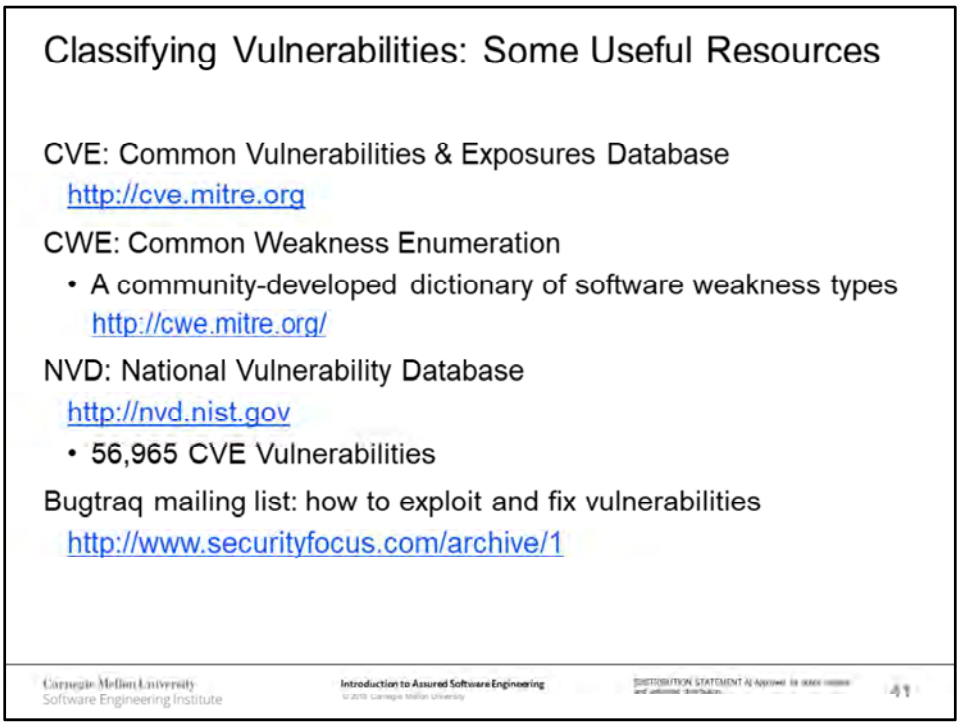
1. Architecture and design stds capturing Authentication, Authz, Accountability, crypto, design principles type of requirements
2. Coding stds. cover basic coding issues that can lead to various vulnerabilities such as validation issues, various injection vulns, web and C/C++ coding stds etc.
3. Process stds. cover things/ activities that product teams should do in order to bake security in their dev lifecycle

Assessment gives us 2 things: Product gaps and Process gaps that give us the level of risk in the product due to non-compliance to the policy and the org maturity level which is primarily based on the process gaps.

Talk about using health risk assessment as a way to do security.



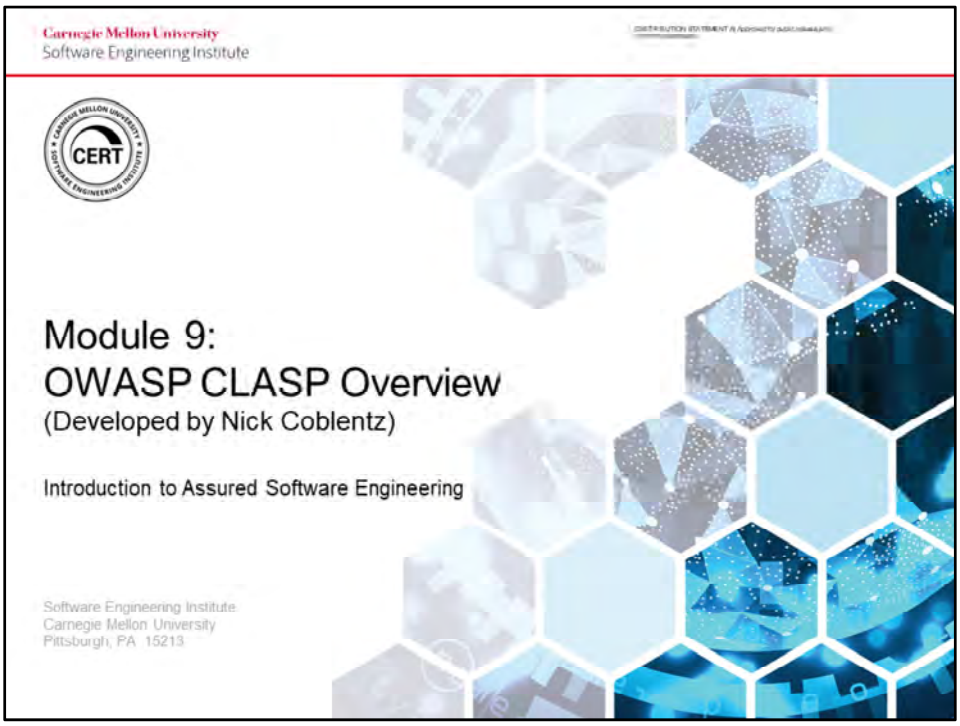
Open – Trusted Technology Provider Standard

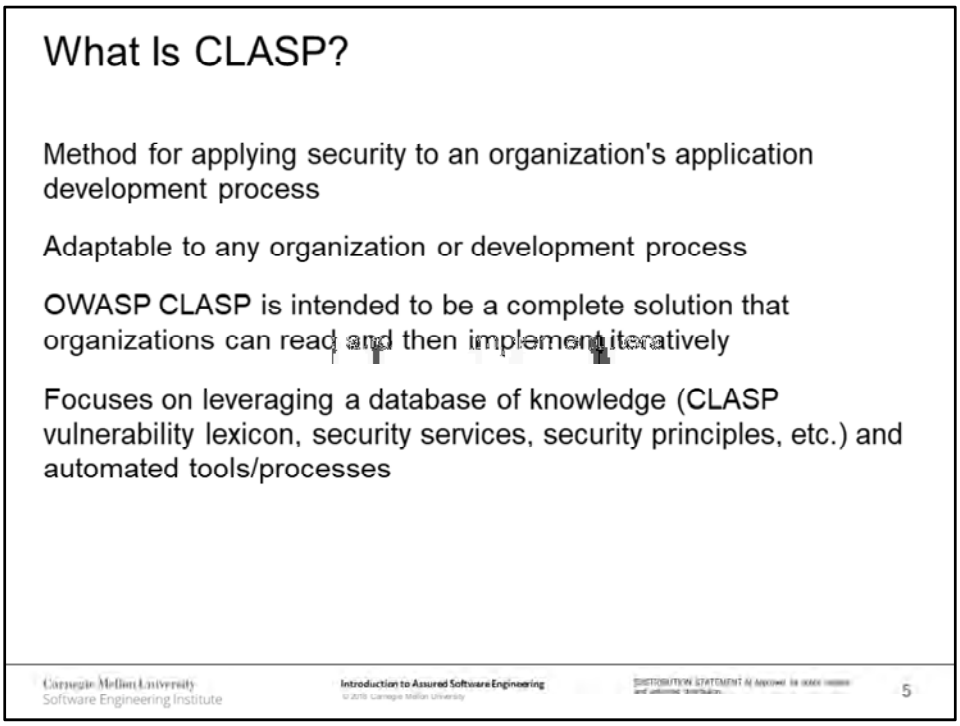


Some top categories

- Data handling
- API abuse
- Security features
- Time & state
- Error handling
- Code quality
- Encapsulation

Source: Common Weaknesses Enumeration (CWE)


Carnegie Mellon University
Software Engineering Institute
CERT
Module 9:
OWASP CLASP Overview
(Developed by Nick Coblentz)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Method for applying security to an organization (not just SDLC)
Organization -

- awareness programs
- metrics
- global security policies

Application -

- threat modeling
- arch review
- pen test
- code review

Adaptable to any organization or development process

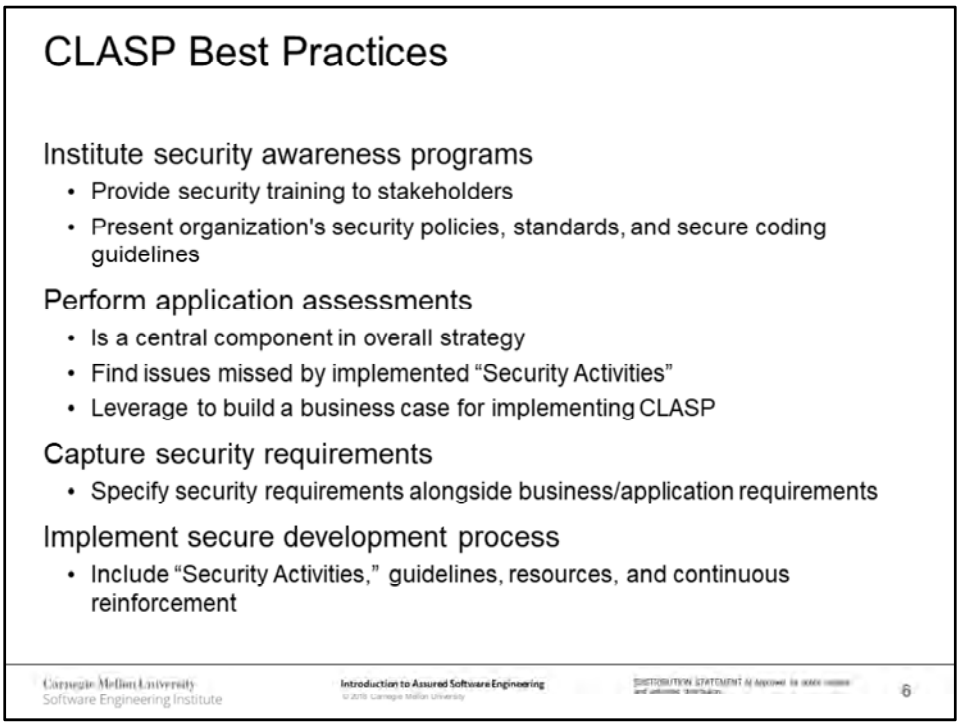
- Fits closest to Rational Unified Process (iterative process with heavy focus on roles)
- also works with Waterfall and to some extent Agile

Intended to be a complete solution that organizations can read and then implement iteratively

- The idea is for this documentation can be handed to a responsible person and they will have enough information to implement one or more parts in their organization

Focuses on leveraging a database of knowledge (much provided directly by CLASP) and automated tools/processes

- CLASP provides a vulnerability lexicon, security principles and other information as a resource to perform security activities
- Also recommends utilizing automated tools whenever possible (code review, assessments, etc.)



BP1: Institute security awareness program

BP2: Perform application assessments

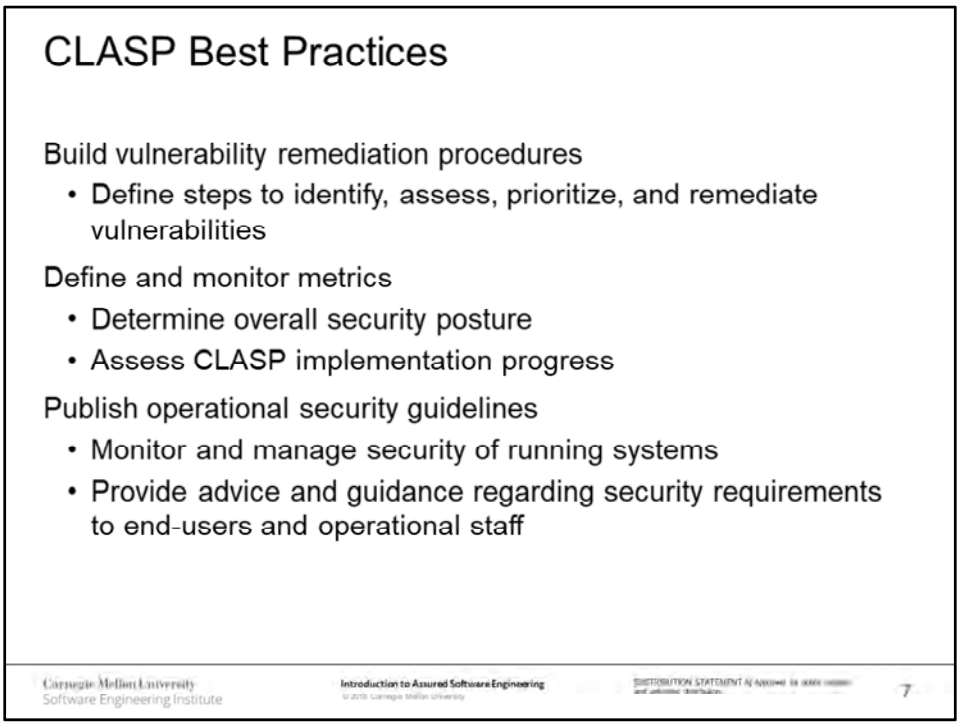
- Perform security analysis of system requirements and design (threat modeling)
- Research and assess security posture of technology solutions
- Perform source-level security review
- Identify, implement, and perform security tests
- Verify security attributes of resources

BP3: Capture Security Requirements

- Detail misuse cases
- Document security-relevant requirements
- Identify attack surface
- Identify global security policy
- Identify resources and trust boundaries
- Identify user roles and resource capabilities
- Specify operational environment

BP3: Implement secure development process

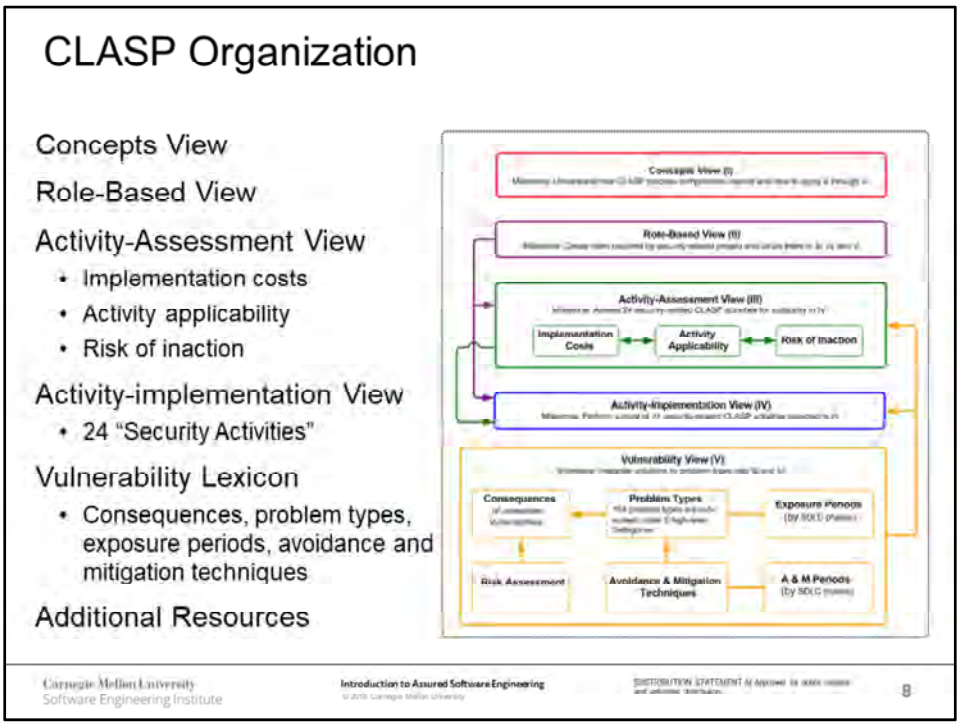
- Annotate class designs with security properties
- Apply security principles to design
- Implement and elaborate resource policies and security technologies
- Implement interface contracts
- Integrate security analysis into source management process
- Perform code signing



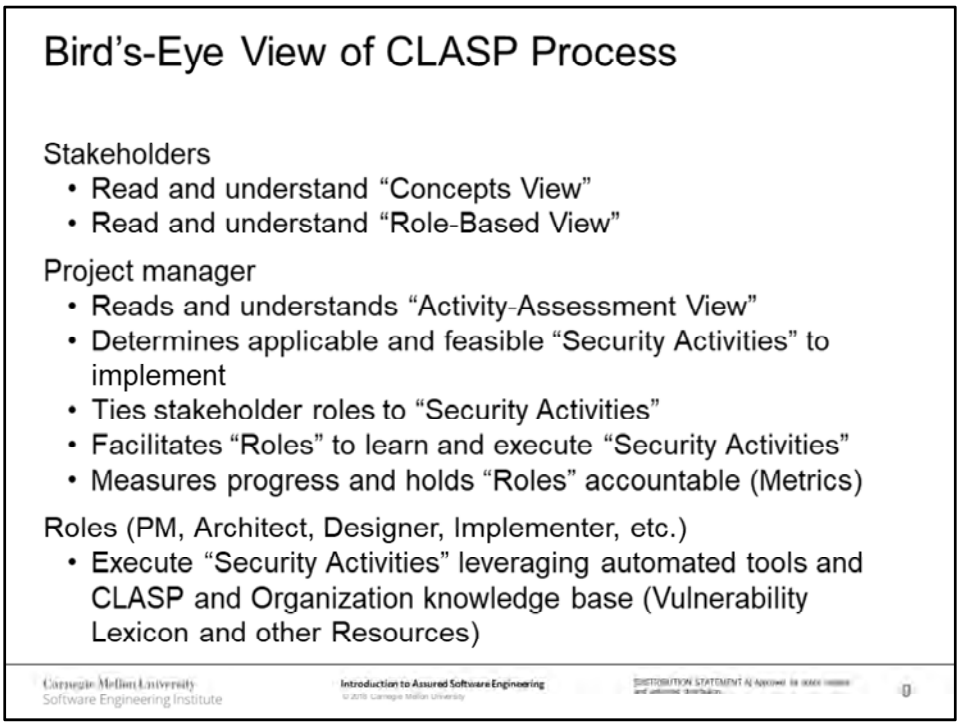
BP5: Build vulnerability remediation procedures
- Address reported security issues
- Manage security issue disclosure process

BP6: Define and monitor metrics

BP7: Publish operational security guidelines
- Build operational security guide
- Specify database security configuration



Organization used in learning and presenting CLASP material
Documentation corresponds to these topics
Detail later
Frame conversation

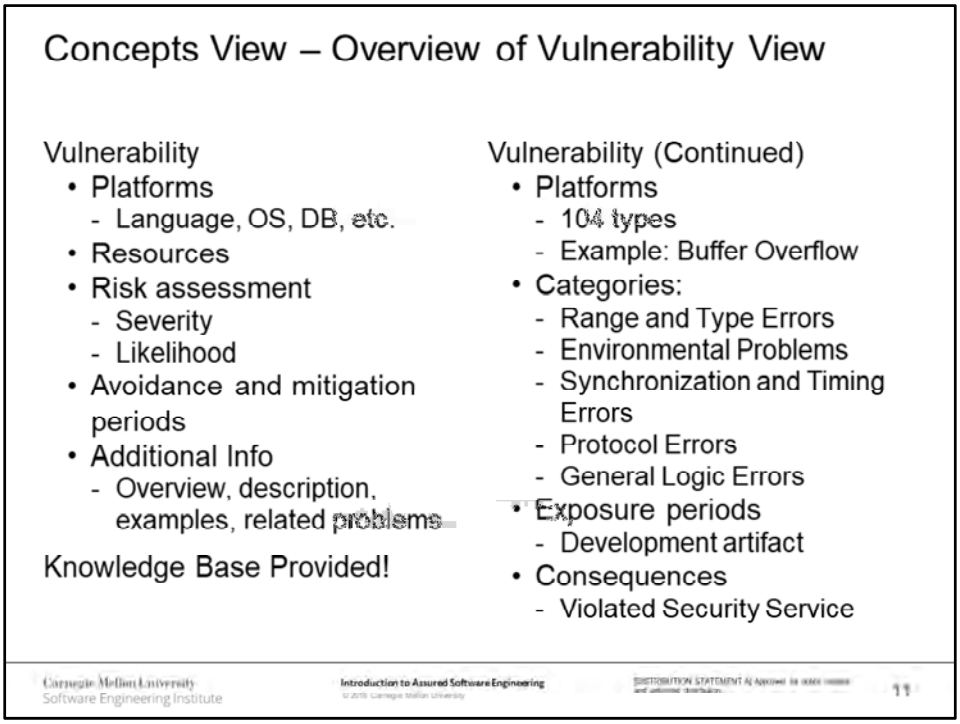


Process for implementing security

Stakeholders are all the people involved in the planning, design, implementation, and delivery of the project.

PM is the driving force behind CLASP implementation.

Security Activities are tied to Roles such as Architect, PM, Designer, Requirements Specifier, Implementer, etc. instead of steps of the development process.



Problem types “basic causes”

Categories

Exposure Periods

- Development lifecycle artifact where vulnerability is introduced
- Examples: requirements specification; architecture and design; implementation; and deployment

Consequences

Violated CLASP “Security Service”

- Authorization (resource access control)
- Confidentiality (of data or other resources)
- Authentication (identity establishment and integrity)
- Availability (denial of service)
- Accountability
- Non-repudiation

Platforms

- Programming languages, OS's, Database Products, etc

Resources

- resources needed to exploit the issue
- Ex: local account, physical access to network, etc

Risk Assessment

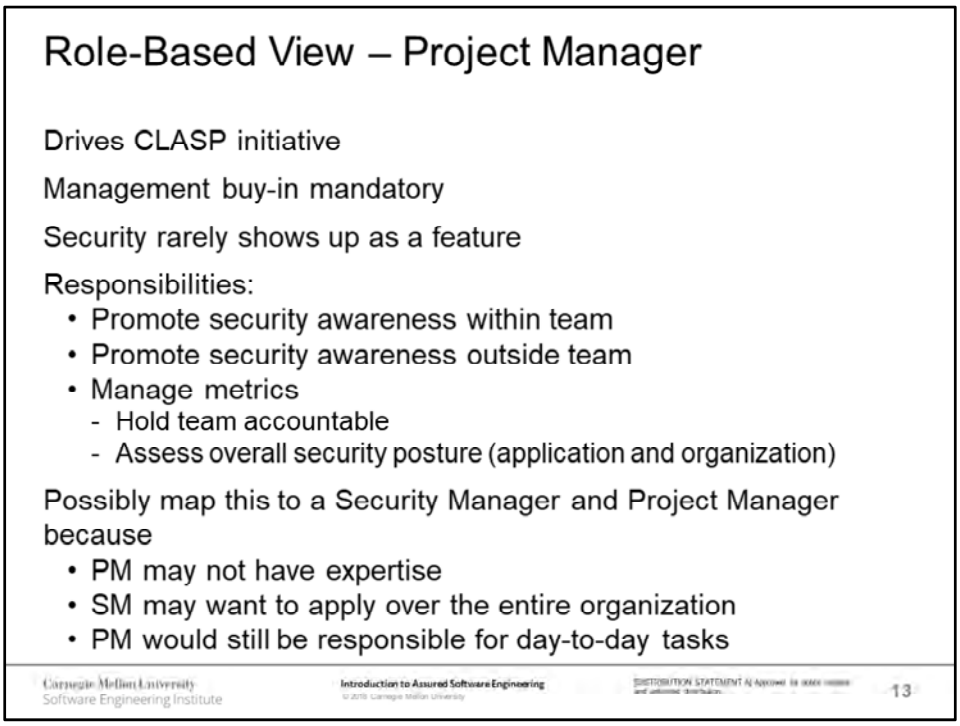
- Severity and Likelihood

Avoidance and Mitigation

- Development lifecycle artifact and activity that can be used to eliminate the issue in the future

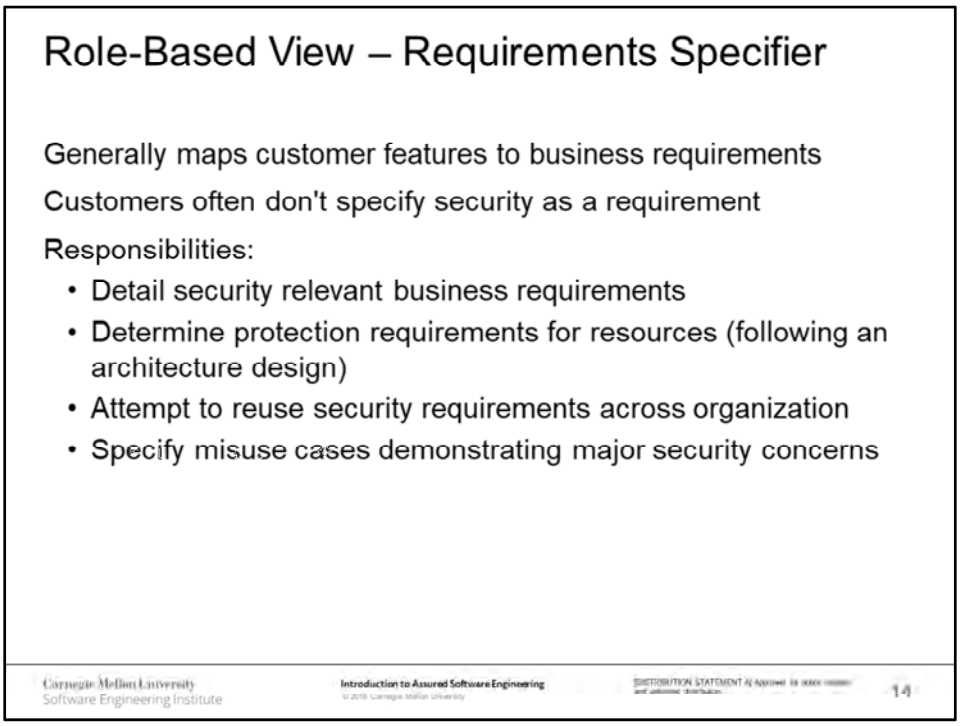
Knowledge Base Provided!!!!

- Provides the majority of this information already



Responsibilities

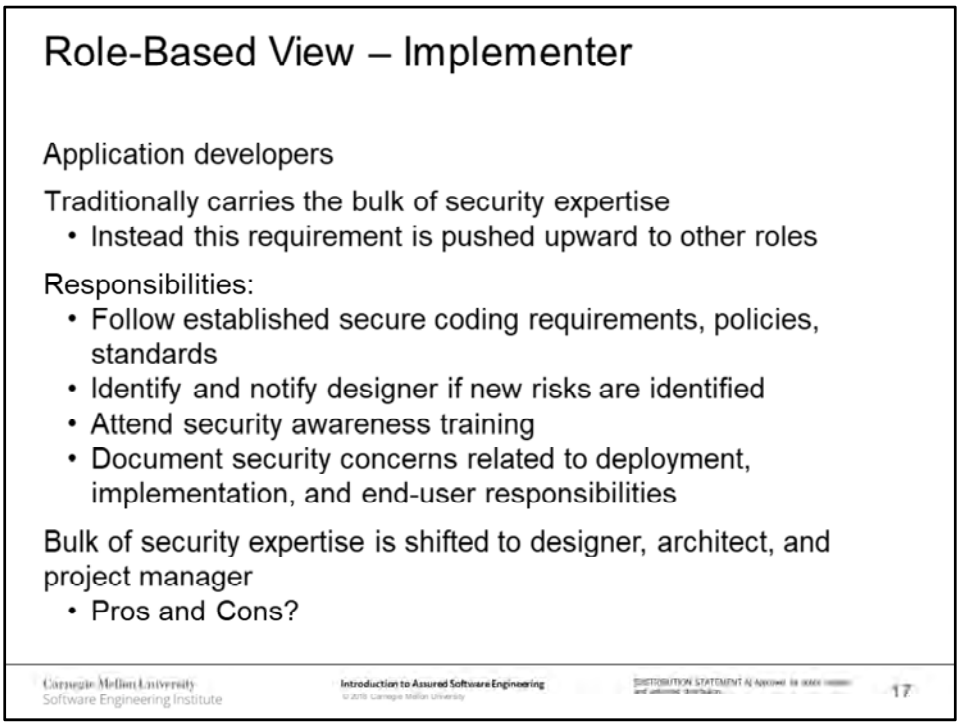
- awareness within team
  - Training classes
  - hold meetings to encourage discussion about security documentation (policies, requirements, threat models, etc)
- outside of team
  - demonstrate impact of application security on the business to organization (CSO, CIO, etc)
- Manage Metrics
  - May not be collecting metrics but needs to analyze them
    - o determine organization and application security posture
    - o How to improve process
    - o how to improve individuals



Requirements specifier often does not have the security expertise to provide detailed and complete security requirements.

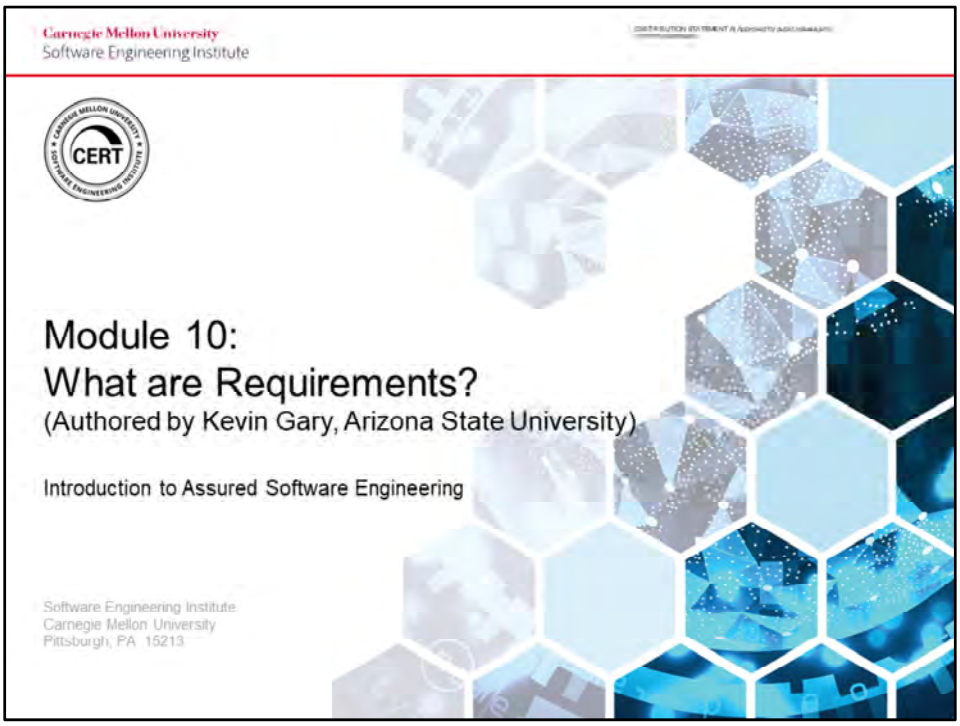
can provide initial high-level goals

Other roles will often contribute security requirements when carrying out other tasks such as threat modelling.

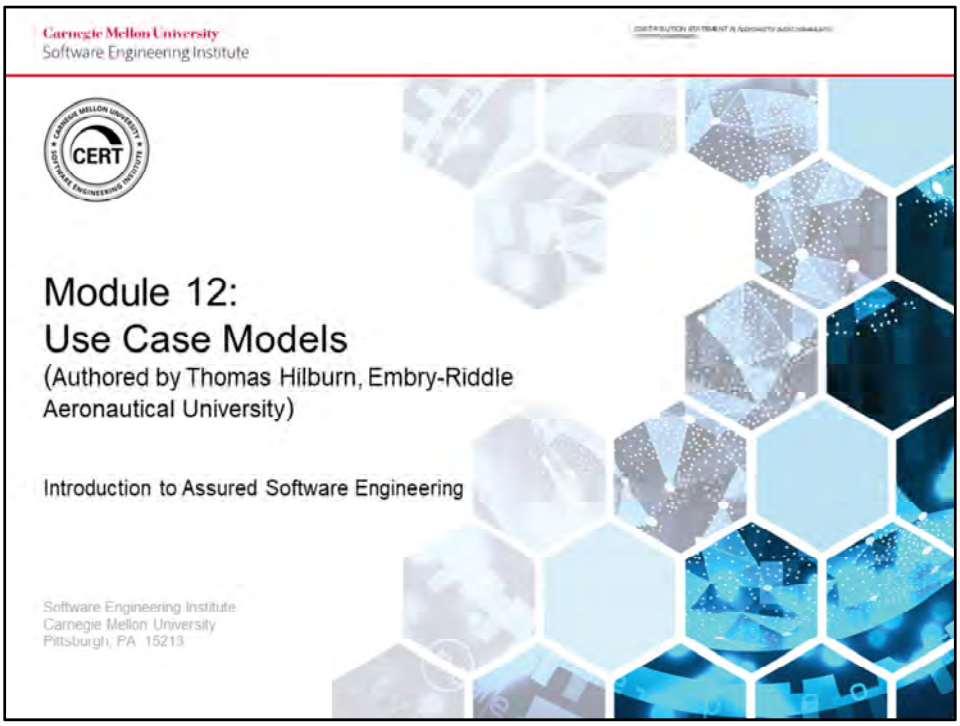


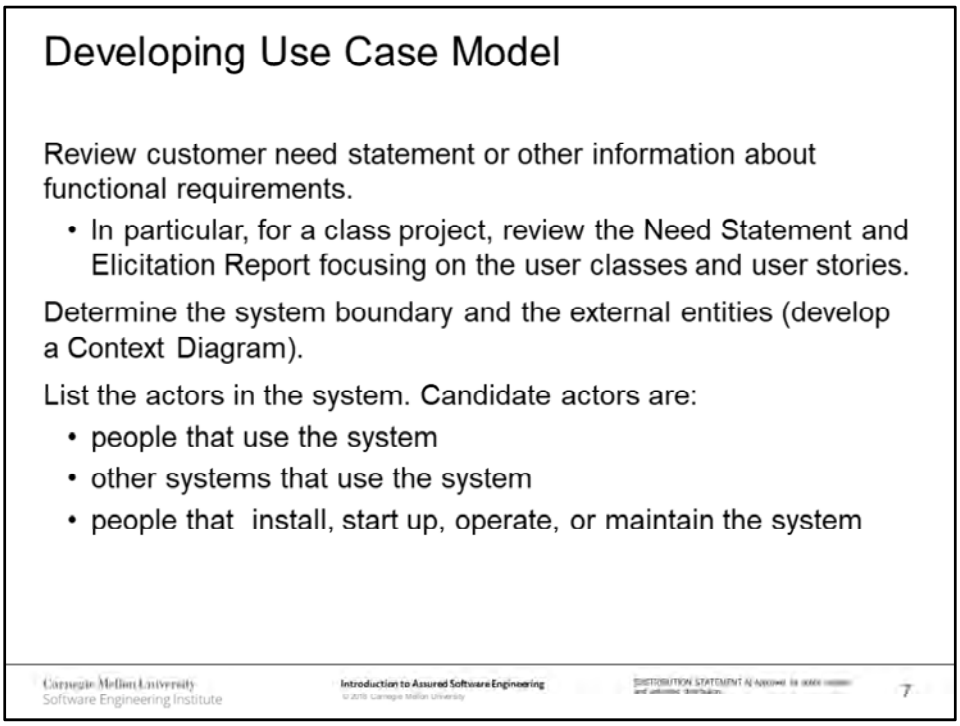
Pros and Cons

- Reduced cost in high priced extremely detail oriented security training
- Those three roles can be trained in CLASP
- Developers simply follow organization's policies, standards, and guidelines and only have to worry about implementing bugs
- Still need awareness training to contribute to misuse cases, threat modeling, architecture/design review, etc.

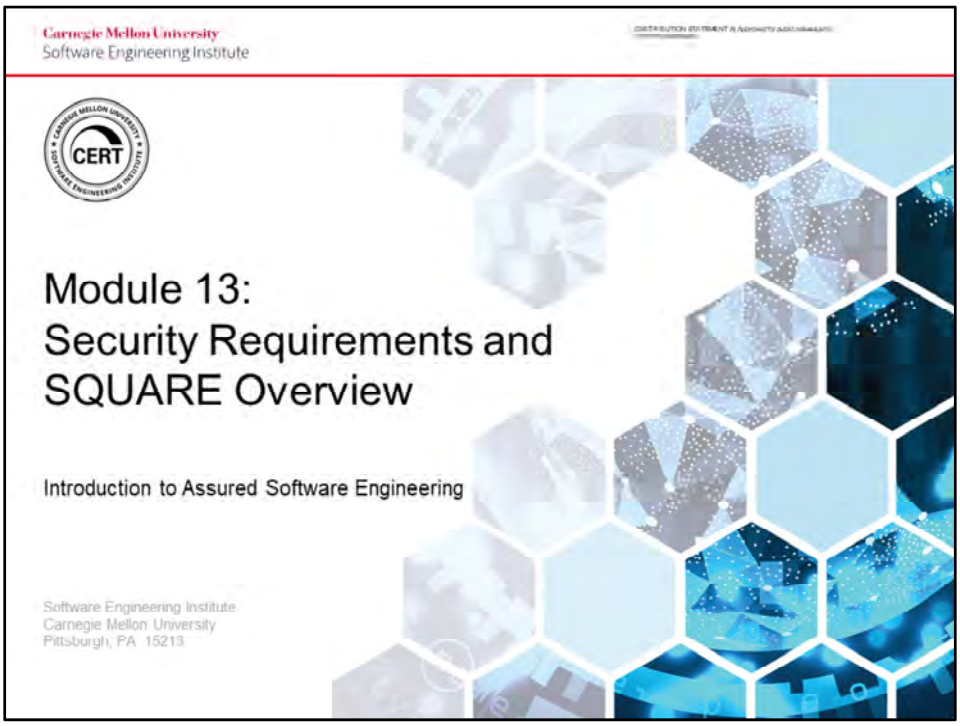


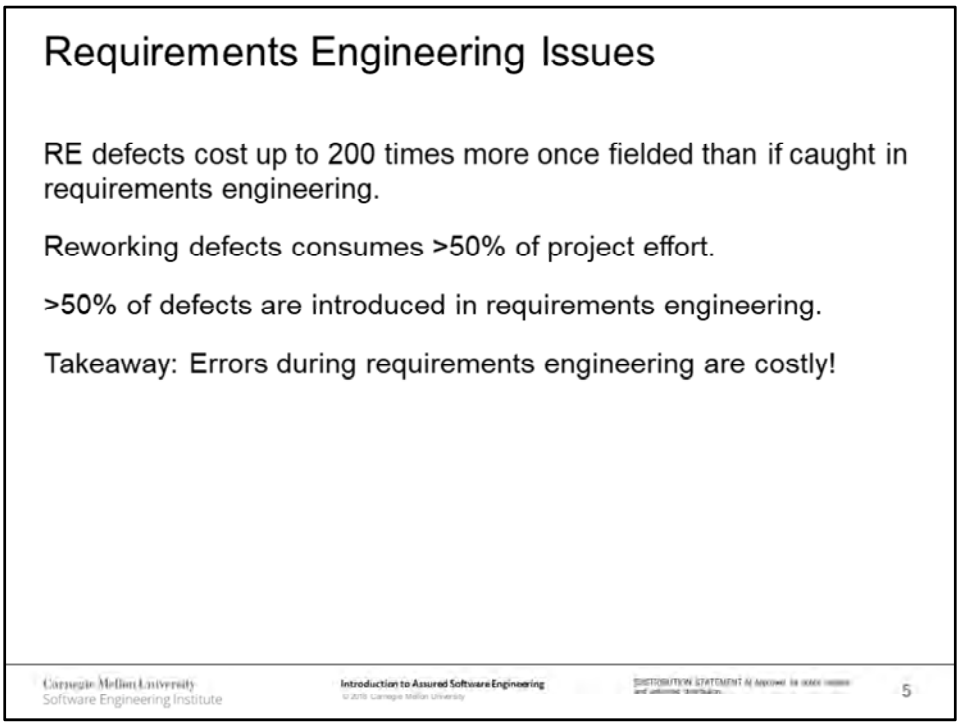
ASU site has resources, quizzes and exams that could be useful.
For more resources: https://softwareenterprise.asu.edu/curricular-modules


Carnegie Mellon University
Software Engineering Institute
CERT
Module 12:
Use Case Models
(Authored by Thomas Hilburn, Embry-Riddle Aeronautical University)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

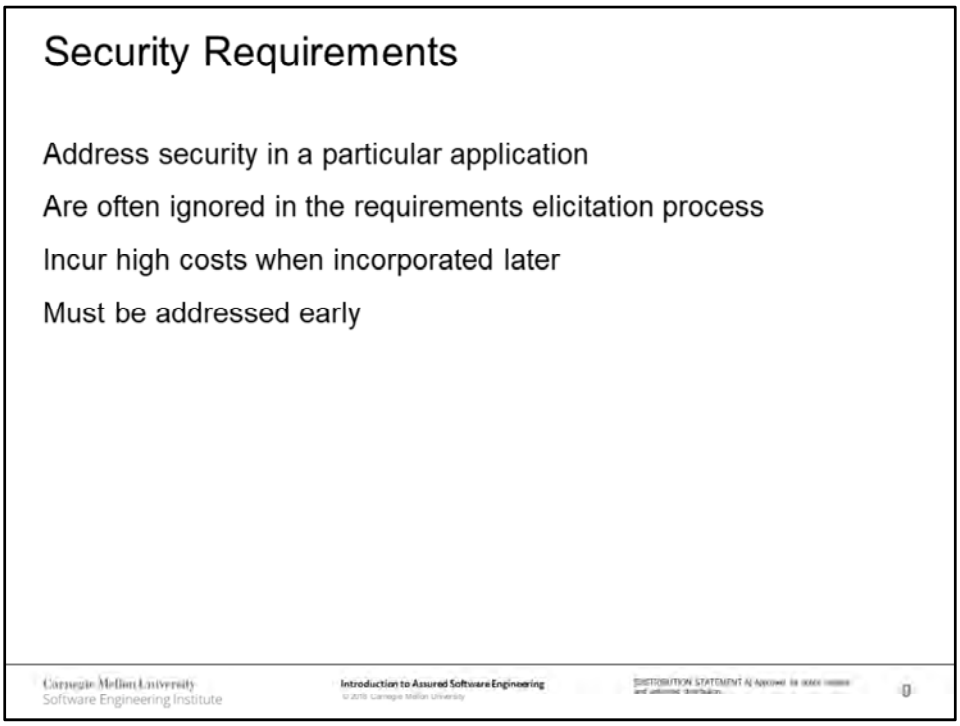


Instructor Note: For the review mentioned here, the instructor will need to provide a project for the students to work on. Usually we would do a project that runs the length of the course, with a mix of individual and team assignments.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 13:
Security Requirements and
SQUARE Overview
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



It is well recognized in industry that requirements engineering is critical to the success of any major development project. Several authoritative studies have shown that requirements engineering defects cost 10 to 200 times as much to correct once fielded than if they were detected during requirements development.



Often the security requirements are developed independently of the rest of the requirements engineering activity and hence are not integrated into the mainstream of the requirements activities. As a result, security requirements that are specific to the system and that provide for protection of essential services and assets are often neglected.

The requirements elicitation and analysis that is needed to get a better set of security requirements seldom takes place.

Typically we focus on what the system should do and not what it should *not* do. Often security related requirements need to be framed in the light of what the system should not do.

Security Requirements Engineering Issues – Example

Cost of Fixing Vulnerabilities Later

Stage	Critical Bugs Identified	Cost of Fixing One Bug	Cost of Fixing All Bugs
Requirements		$130	
Design		$455	
Coding		$977	
Testing	50	$7,136	$356,800
Maintenance	150	$14,102	$2,115,300
Total	**200**		**$2,472,100**

Cost of Fixing Vulnerabilities Early

Stage	Critical Bugs Identified	Cost of Fixing One Bug	Cost of Fixing All Bugs
Requirements		$130	
Design		$455	
Coding	150	$977	$146,550
Testing	50	$7,136	$356,800
Maintenance		$14,102	
Total	**200**		**$503,350**

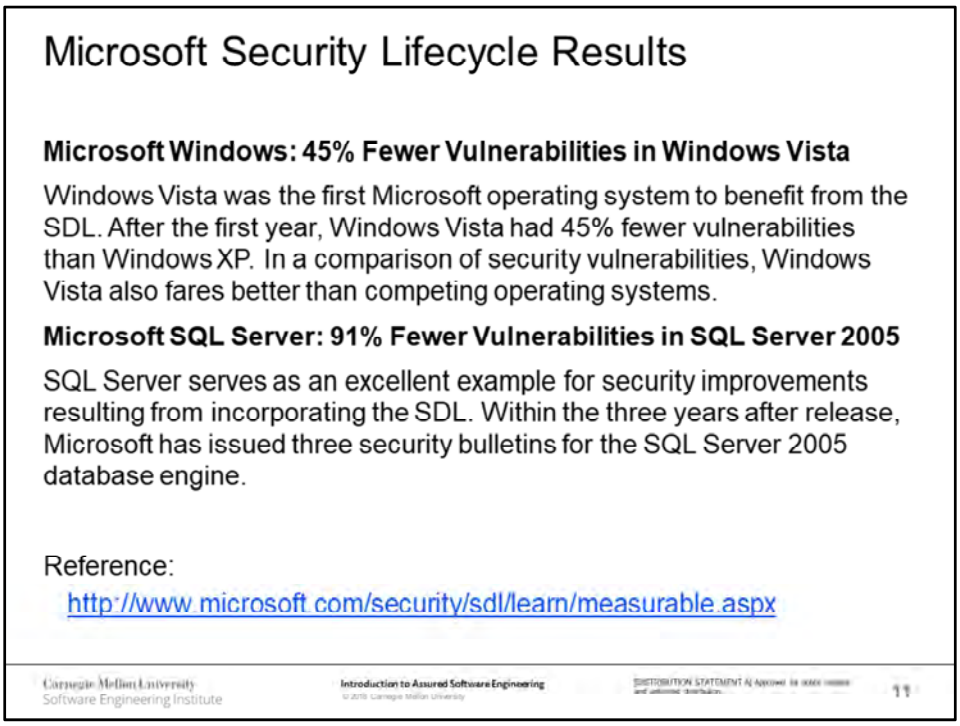
As can be seen, identifying defects early in the lifecycle reduced costs by nearly $2 million.

Carnegie Mellon University Software Engineering Institute — Introduction to Assured Software Engineering — 10

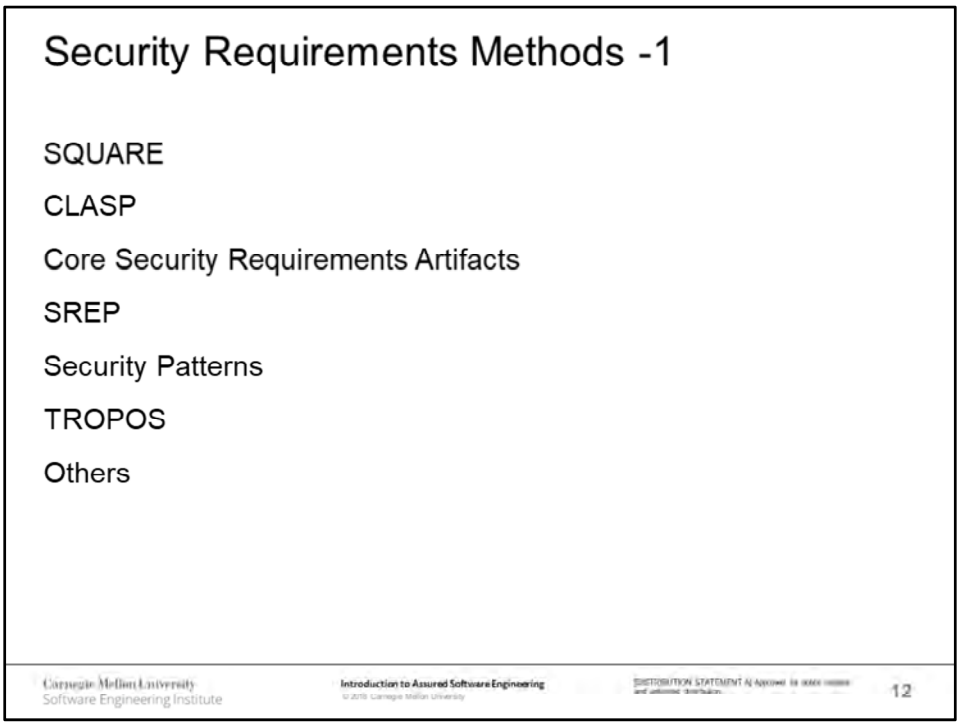
This is an example from a presentation made on Identifying and Reducing Software Risk in the Enterprise by Fortify Software.

This example highlights the enormity of the situation with respect to fixing bugs later in the life cycle. As can be seen from the tables, on the left, most defects were identified and fixed in the testing and maintenance phases of the project. However, the table on the right shows that most of the defects were identified and fixed earlier, i.e. in the coding and testing phases of the project. As a result, it resulted in saving up to $2 million.

Thus, it is of paramount importance to detect and fix defects as early in the project life cycle as possible.



Microsoft SDL process includes more attention to requirements, especially with the use of STRIDE for threat modeling and attack surface. (They have their own method and do not use SQUARE.)



These are some of the methods and processes that can be used in identifying security requirements:

Security Quality Requirements Engineering (SQUARE) is a process aimed specifically at security requirements engineering.

The Comprehensive, Lightweight Application Security Process (CLASP) approach to security requirements engineering [OWASP 2007] is a life-cycle process that suggests a number of different activities across the development life cycle to improve security. Among these is a specific approach for security requirements.

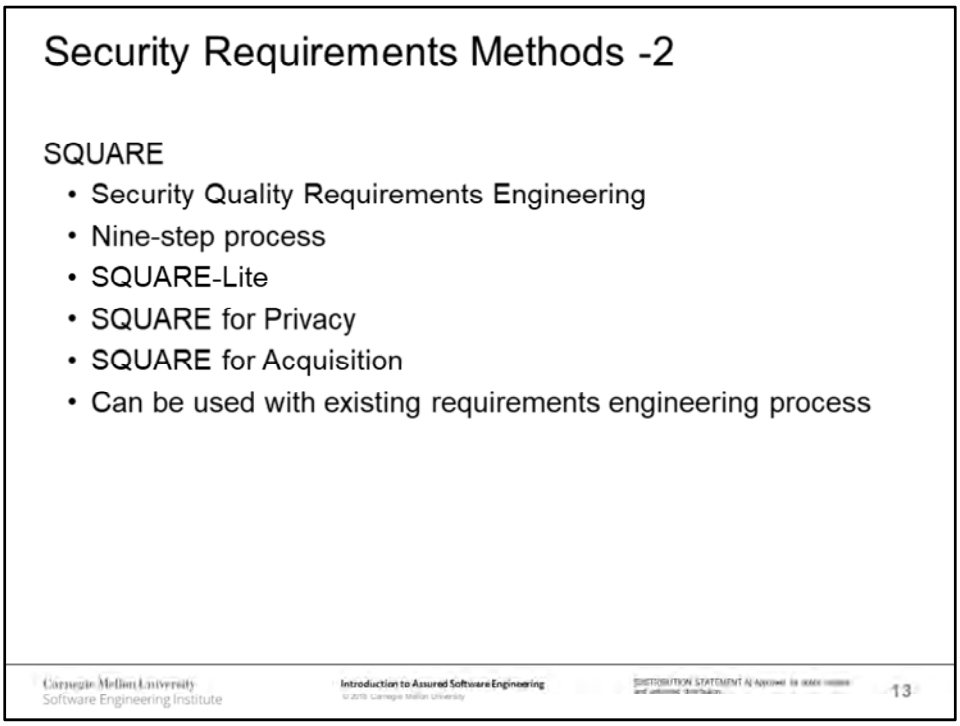
Core security requirements artifacts [Moffett 2004] takes an artifact view and starts with the artifacts that are needed to achieve better security requirements.

The Security Requirements Engineering Process (SREP) [Mellado 2007] is a nine-step process that is based partially on SQUARE but incorporates consideration of the Common Criteria and notions of reuse.

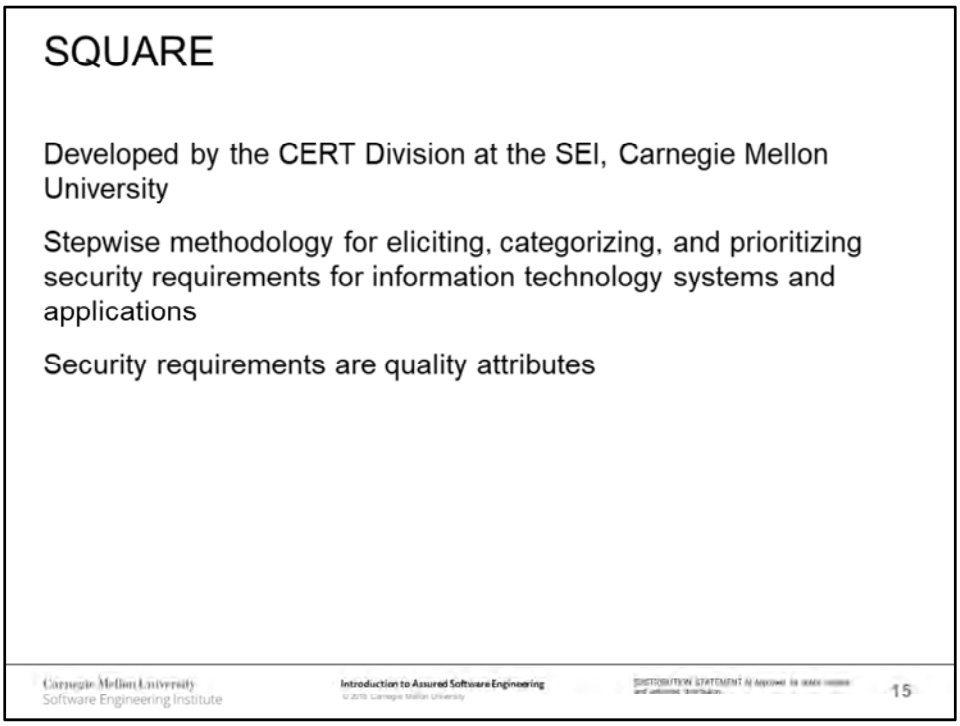
Security patterns are useful in going from requirements to architectures and then designs [Haley 2007, Rosado 2006, Weiss 2007].

Tropos is a self-contained life-cycle approach [Giorgini 2007]. It is very specific in terms of how to go about requirements specification.

Other useful techniques are the use of attack trees in security requirements engineering [Ellison 2003] and misuse and abuse cases [Alexander 2003, Fernandez 2007, Sindre 2000]. Formal specification approaches to security requirements, such as Software Cost Reduction (SCR) [Heitmeyer 2002] have also been useful. The higher levels of the Common Criteria [CCEVS 2007] provide similar results.

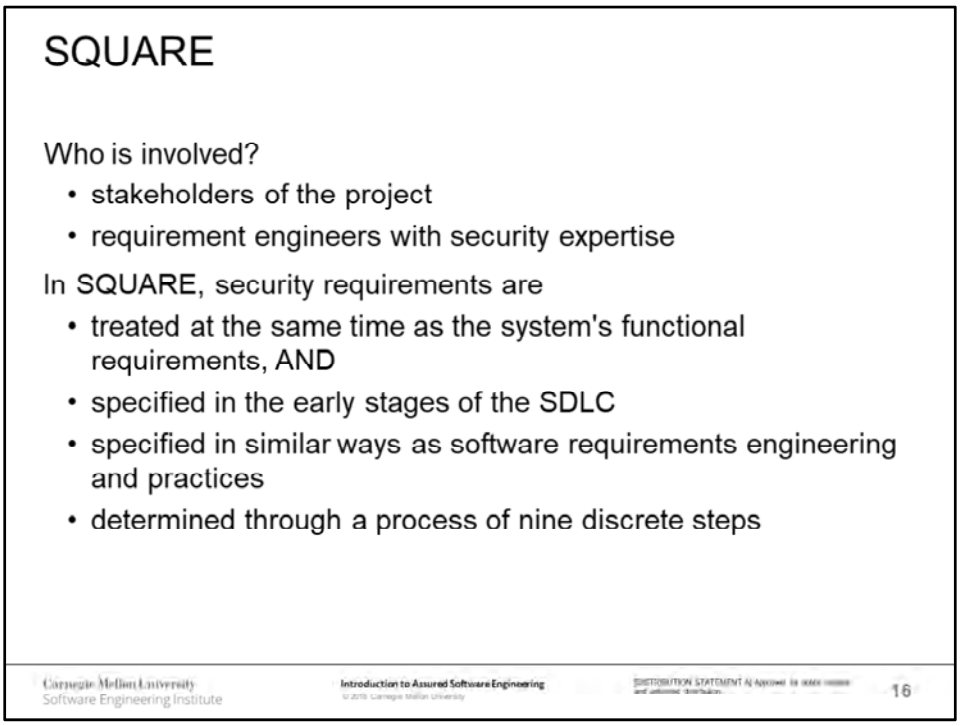


Briefly introduce SQUARE



SQUARE: Mention that this lecture will be a brief overview of SQUARE. All nine steps will be introduced and skimmed over. The two lectures following will describe it in detail.

SQUARE history: The Software Engineering Institute's Networked Systems Survivability (NSS) Program at Carnegie Mellon University developed SQUARE as a methodology to help organizations build security into the early stages of the production life cycle. Development of the SQUARE methodology started in 2003. It was initially baselined in 2005. Since then it has been published in papers and book chapters. There is a robust tool to support SQUARE.



Describe the two main parties involved. Define what stakeholder means in this situation (it represents only the clients).

Reiterate that SQUARE is not a new requirements engineering process but meant to be used with an existing RE process.

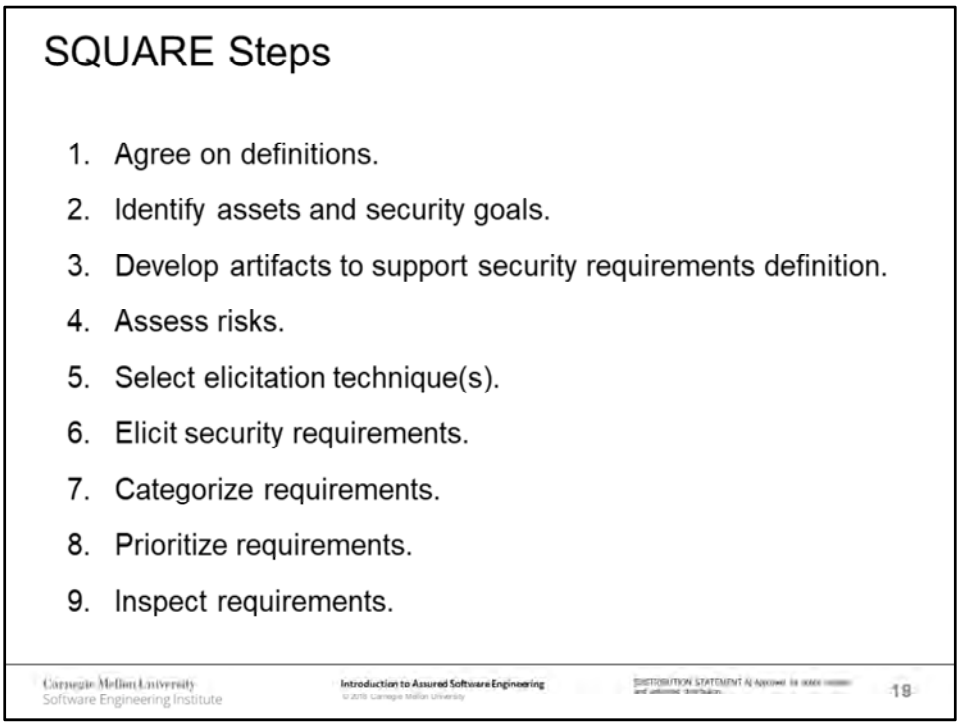
Objective is to elicit security requirements artifact as a documented artifact. Security requirements are seen as quality attributes.

It is a well documented process.

The time needed to execute SQUARE depends on the size of the project.

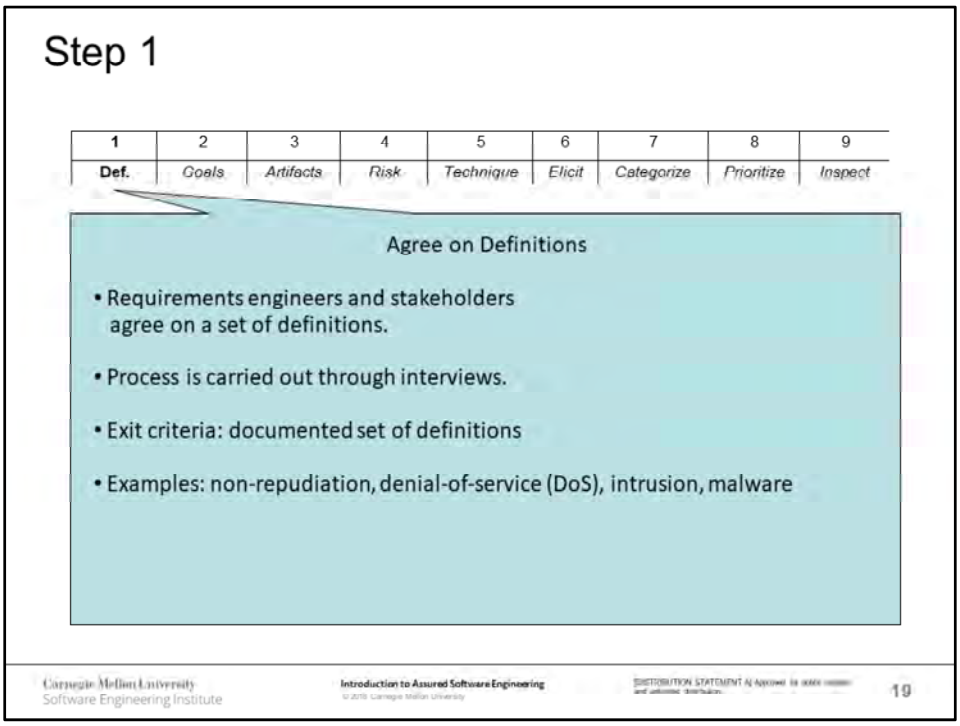
A lighter version of SQUARE called SQUARE-Lite is also being developed. Using SQUARE-Lite, one can complete the process in a shorter time than the full SQUARE.

Actors involved: A team of stakeholders (in this context, stakeholders would only mean the clients) and a requirements engineering team (should ideally have expertise in security)



Run through these steps, preferably with an example. Use this example throughout this lecture.

Could use the example from the previous lecture and build on one aspect of it.



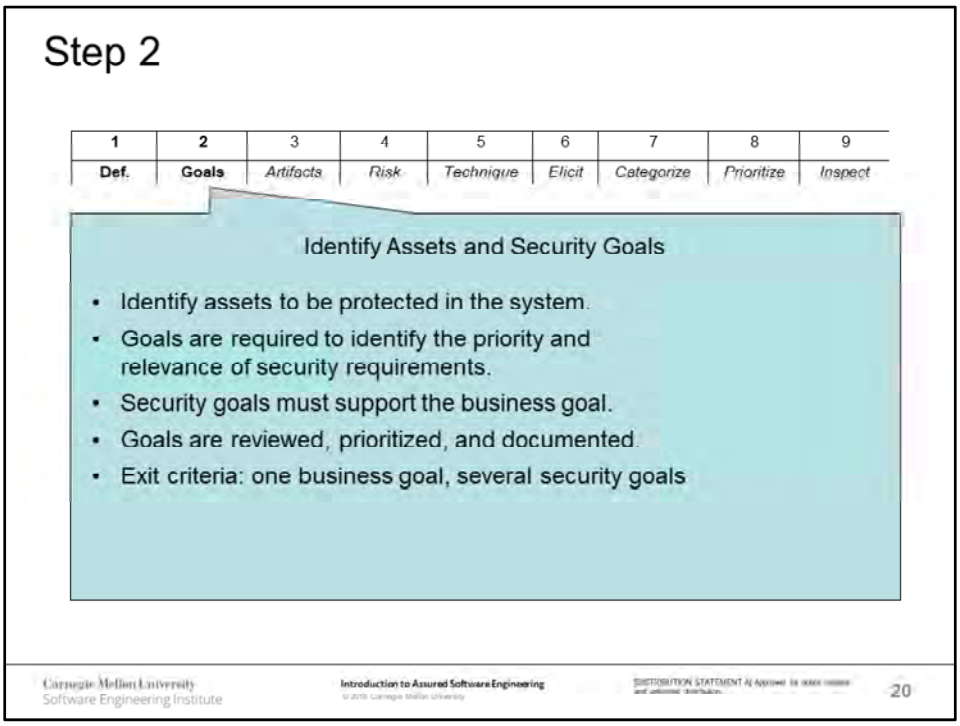
- The requirements engineering team and stakeholders must first agree on a common set of terminology and definitions.

- Guarantees effective and clear communication throughout the requirements engineering process

- Resolves ambiguity and differences in perspective

- SQUARE. Using public resources, such as the Software Engineering Body of Knowledge, (SWEBOK) [IEEE 05], IEEE 610.12 Standard Glossary of Software Engineering Terminology, [IEEE 90], and Wikipedia.

Examples: access control (ACL), antivirus software, artifact, asset, attack, audit, authentication, availability, back door, breach, brute force, buffer overflow, cache cramming, cache poisoning, confidentiality, control.



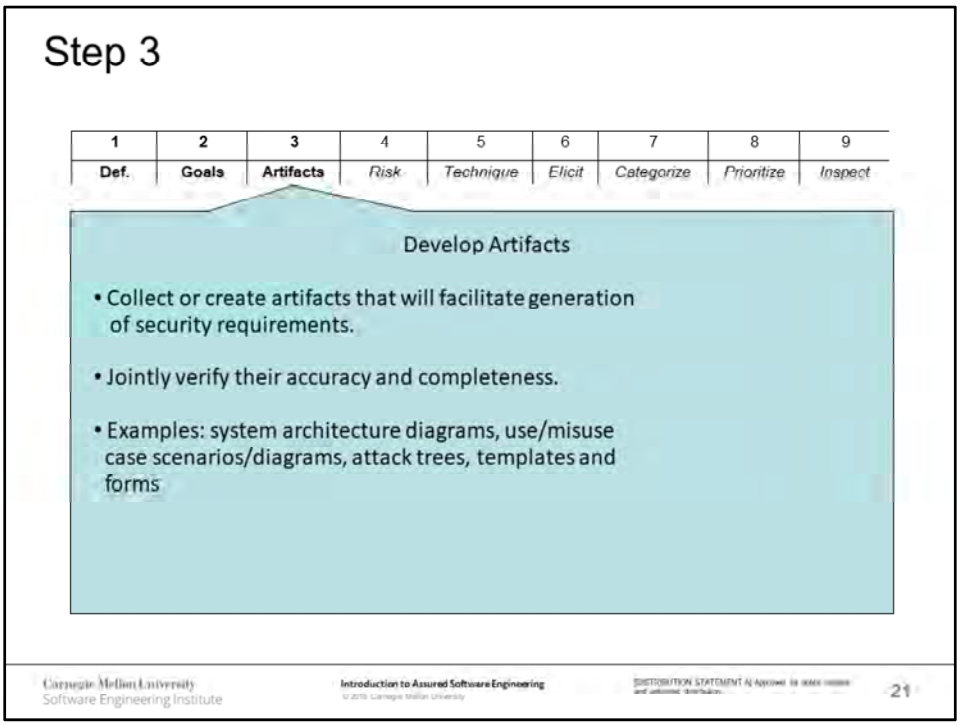
Initially, different stakeholders will likely have different security goals. Therefore they should formally agree on a set of prioritized security goals for the project. Without overall security goals for the project, it is impossible to identify the priority and relevance of any security requirements that are generated.

The security goals of the project must be in clear support of the project's overall business goal, which also must be identified and enumerated in this step.

Once the goals of the various stakeholders have been identified, they must be prioritized. In the absence of consensus, an executive decision may be needed to prioritize the goals.

Generate security goals as opposed to requirements or recommendations.

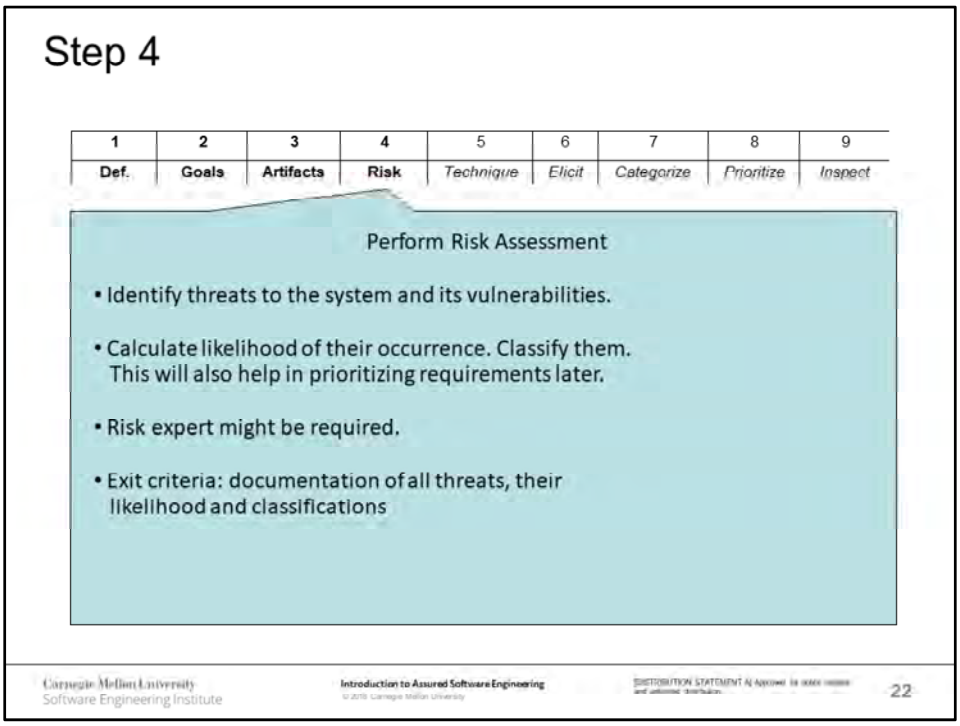
Exit criteria: A single business goal for the project and several prioritized security goals



Complete set of artifacts of the system. The following are the types of artifacts that should be collected:
- system architecture diagrams
- use case scenarios/diagrams
- misuse case scenarios/diagrams
- attack trees
- standardized templates and forms

In developing such artifacts, it is important to enlist the assistance of knowledgeable engineers from the organization.

Verify the accuracy and completeness of all artifacts.

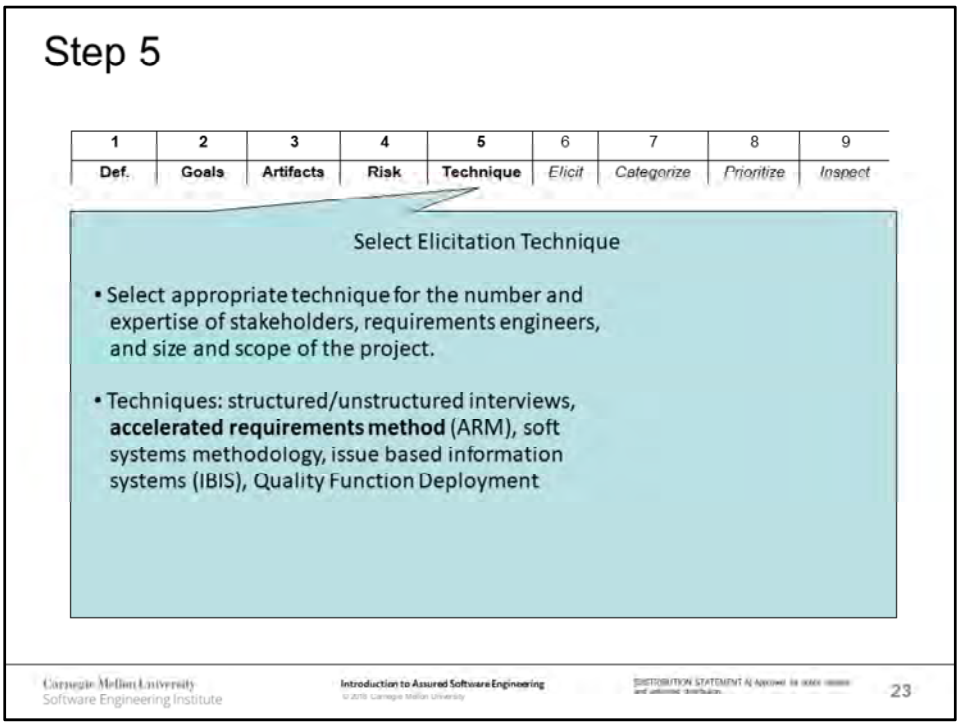


Identify the vulnerabilities and threats that face the system, the likelihood that the threats will materialize as real attacks, and any potential consequences of an attack. Without a risk assessment, organizations can be tempted to implement security requirements or countermeasures without a logical rationale.

After the threats have been identified by the risk assessment method, they must be classified according to likelihoods. Again, this will aid in prioritizing the security requirements that are generated at a later stage. For each threat identified, a corresponding security requirement can identify a quantifiable, verifiable response. For instance, a requirement may describe speed of containment, cost of recovery, or limit to the damage that can be done to the system's functionality.

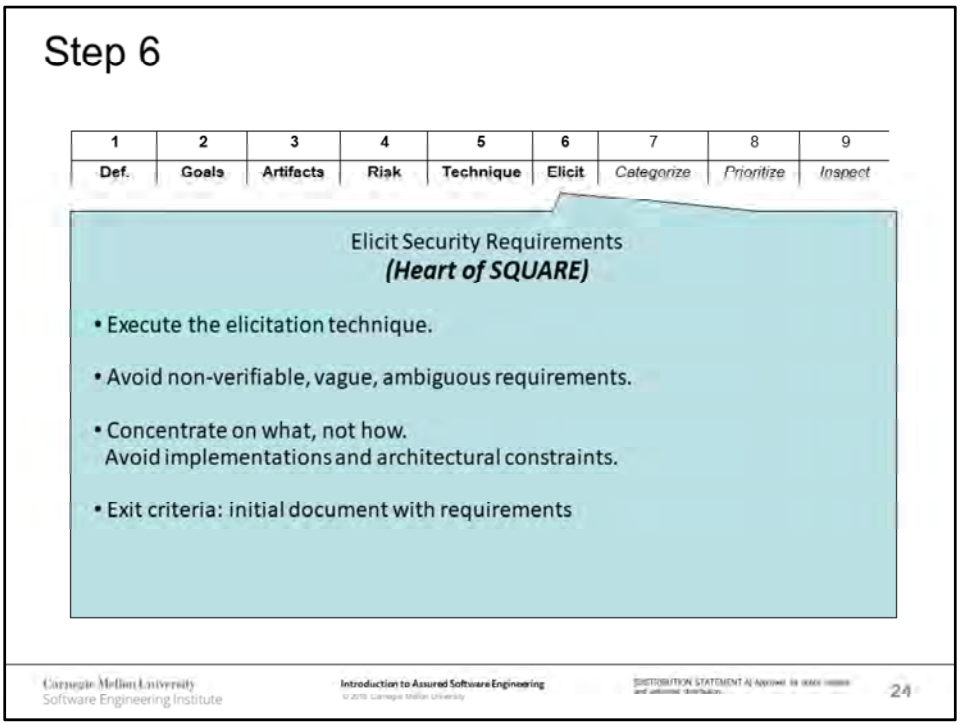
Requirements engineering team should facilitate the completion of a structured risk assessment, likely performed by an external risk expert. Review the results of the risk assessment and share them with stakeholders.

The results must be well documented and shared with the stakeholders.



The requirements engineering team must select an elicitation technique that is suitable for the client organization and project. Multiple techniques will likely work for the same project. The difficulty is in choosing a technique that can adapt to the number and expertise of stakeholders, size and scope of the client project, and expertise of the requirements engineering team. Previous experience has shown that the Accelerated Requirements Method (ARM) has been successful in eliciting security requirements. The organization can also use its existing requirements elicitation technique, although some techniques are not ideal for eliciting security requirements.

Examples
- Structured/unstructured interviews
- Use/misuse cases [Jacobson 92]
- Accelerated Requirements Method [Wood 89, Hubbard 99]
- Soft Systems Methodology [Checkland 89]
- Issue-Based Information Systems [Kunz 70]
- Quality Function Deployment [QFD 05]
- Feature-Oriented Domain Analysis [Kang 90]
- Controlled Requirements Expression [Mullery 79]
- Critical Discourse Analysis [Schiffrin 94]

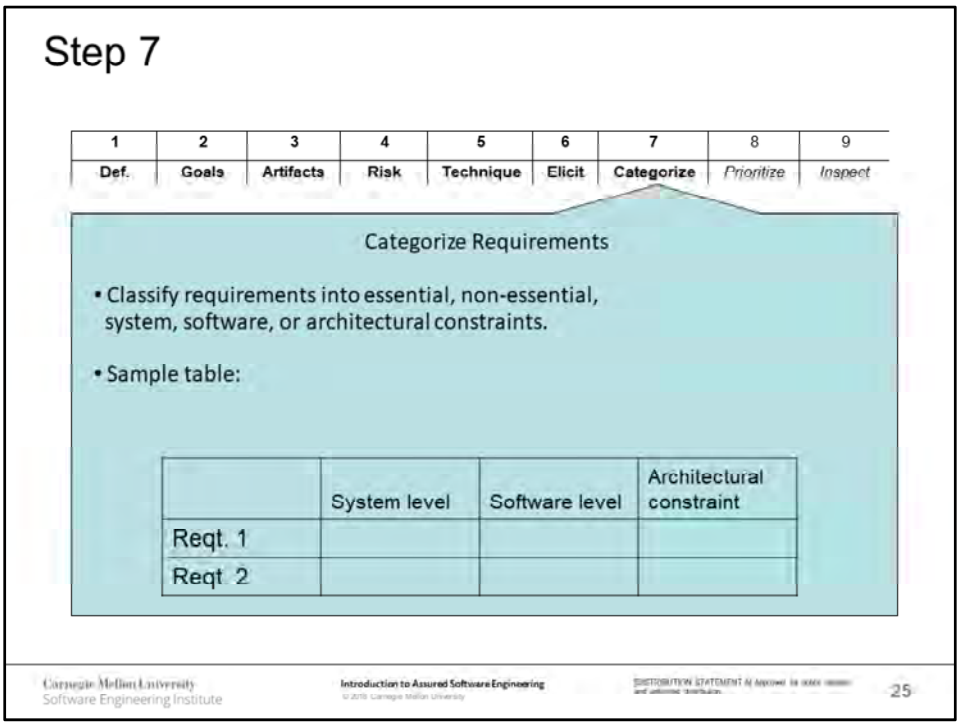


This is the heart of the SQUARE process. This step is simply a matter of executing the technique that was previously selected.

Perhaps the largest mistake that the requirements engineering team can make in this step is to elicit non-verifiable or vague, ambiguous requirements. Each requirement must be stated in a manner that will allow relatively easy verification once the project has been implemented. For instance, the requirement "The system shall improve the availability of the existing customer service center" is impossible to measure objectively. Instead, the requirements engineering team should encourage the production of requirements that are clearly verifiable and, where appropriate, quantifiable. A better version of the previously stated requirement would thus be "The system shall handle at least 300 simultaneous connections to the customer service center." A second mistake that the requirements engineering team can make in this step is to elicit *implementations* or *architectural constraints* instead of requirements. Requirements are concerned with *what* the system should do, not *how* it should be done.

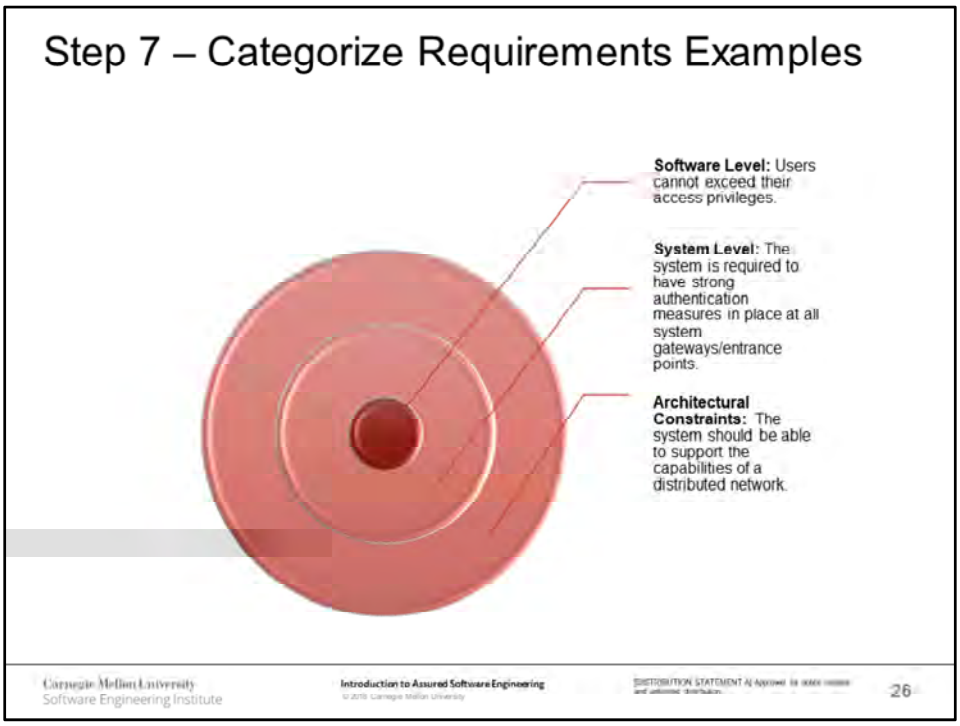
Face-to-face interaction with the stakeholders

Exit Criteria: An initial set of security requirements for the system has been elicited and documented. It is not necessary that the set be considered final or completely correct.



The purpose of this step is to allow the requirements engineer and stakeholders to classify the requirements as essential, non-essential, system level, software level, or as architectural constraints.

Since the goal of SQUARE is to produce security requirements, the requirements engineering team and stakeholders should avoid producing architectural constraints. Architectural constraints are provided as a category here to serve as an outlet for "requirements" that, upon categorization, are considered to be constraints. Ideally, such anomalies would be identified and corrected in the previous steps of the process.

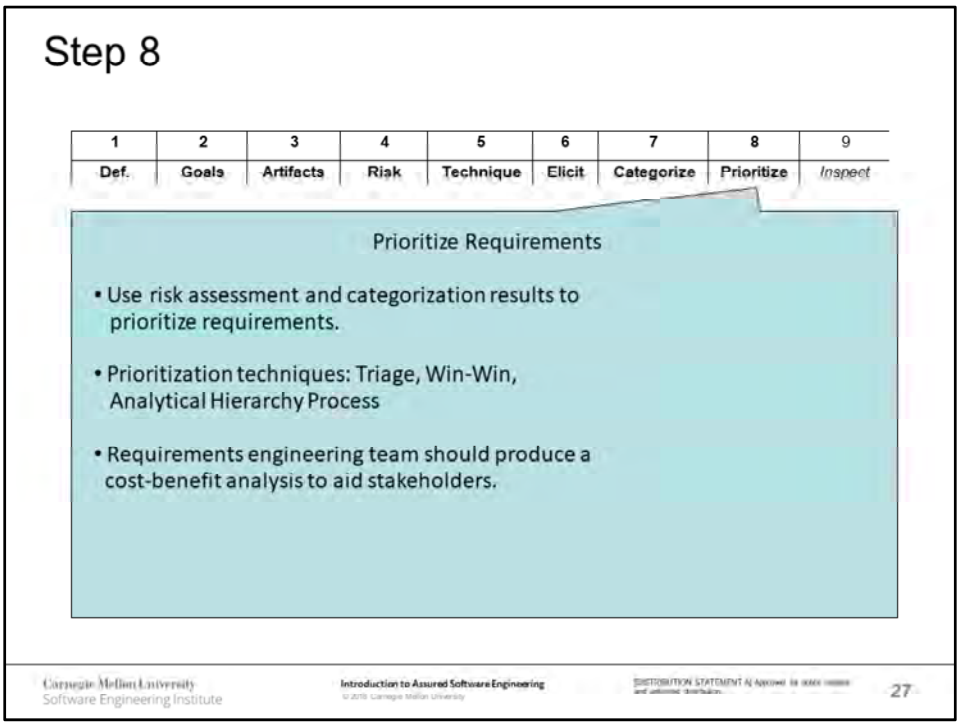


This slide shows an example of each of the different types of requirements that could emerge during the implementation of the SQUARE process.

Software Level: Users cannot exceed their access privileges. Access control would typically be handled in software.

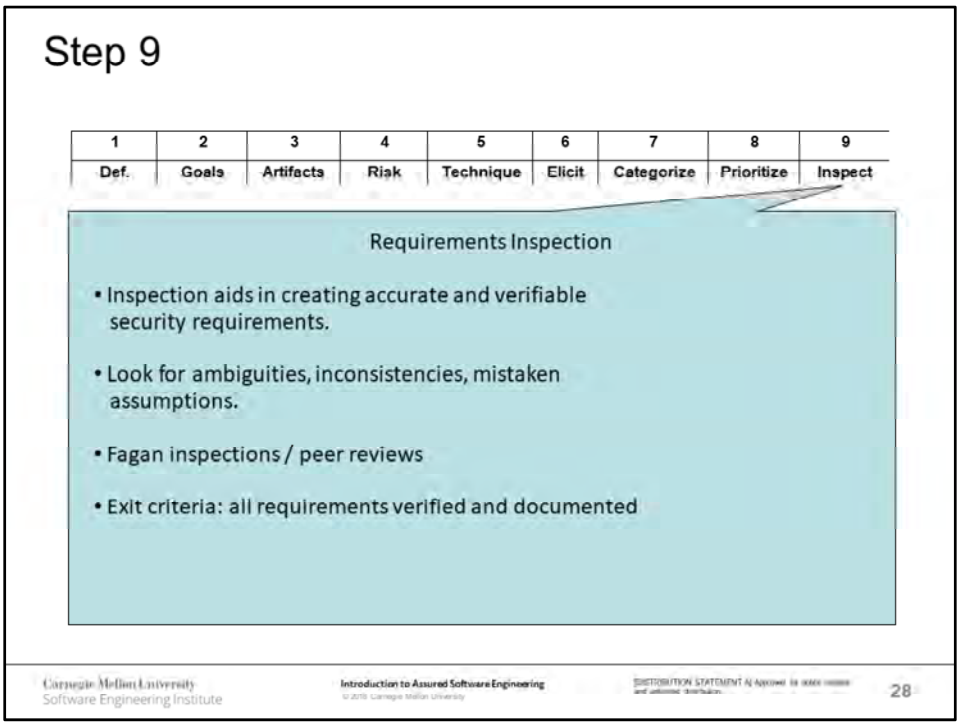
System Level: System level (hardware and software) requirements are handled by the system team to determine how the system will behave. An example for a system level requirement as shown here is to have system level authentication measures in place so as to prevent intrusion from potential hackers and other security threats from taking place.

Architectural Constraints: These are generally the restrictions determined by the customer based on the business needs of the product. Architectural constraints consist of business constraints and technical constraints that can influence the system. An example of an architectural constraint is that the system should support a distributed network. This would have a bearing on how the software would be designed and which technologies would be used to develop and support the distributed environment.



In most cases, the client organization will be unable to implement *all* of the security requirements due to lack of time, resources, or developing changes in the goals of the project. Thus, the purpose of this step in the SQUARE process is to prioritize the security requirements so that the stakeholders can choose which requirements to implement and in what order. The results of Step 4, the risk assessment, and Step 7, categorization, are crucial inputs to this step.

During prioritization, some of the requirements may be deemed to be entirely unfeasible to implement. In such cases, the requirements engineering team has a choice: completely dismiss the requirement from further consideration, or document the requirement as "future work" and remove it from the working set of project requirements. This decision should be made after consulting with the stakeholders.

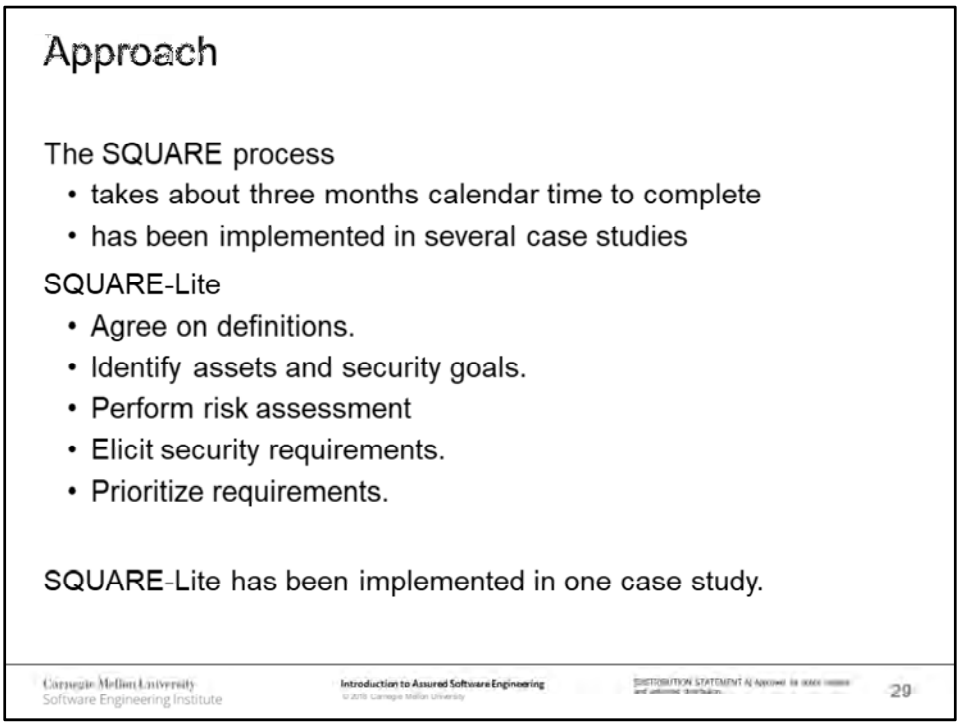


The last step of the SQUARE process, requirements inspection, is one of the most important elements in creating a set of accurate and verifiable security requirements.

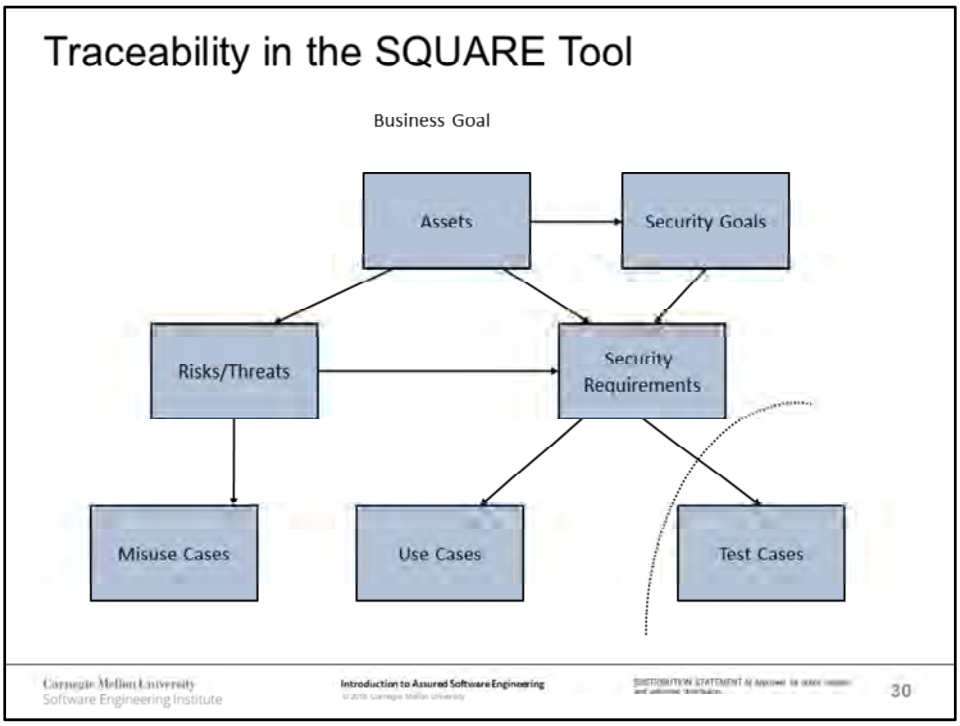
Inspection can be done at varying levels of formality, from Fagan Inspections to peer reviews [Fagan 86, Wiegers 02]. The goal of any inspection method, however, is to find any defects in the requirements such as ambiguities, inconsistencies, or mistaken assumptions.

Stakeholders and the requirements engineering team should come to a consensus on the validity of each security requirement. Verify that each requirement is verifiable, in scope, within financial means, and feasible to implement.

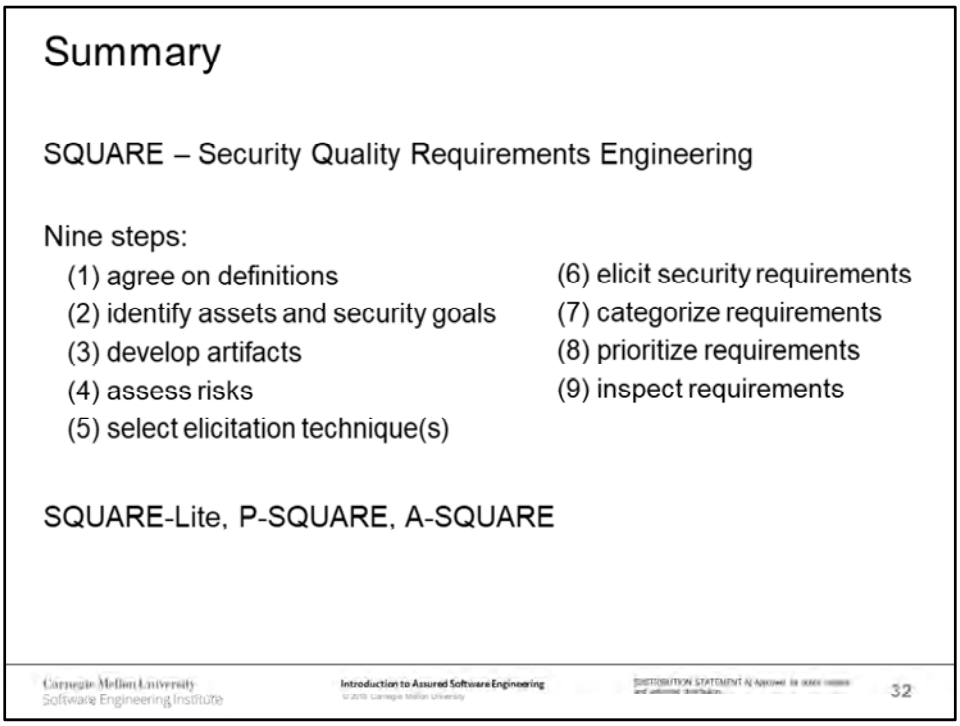
Last chance to remove any requirements from the working set.



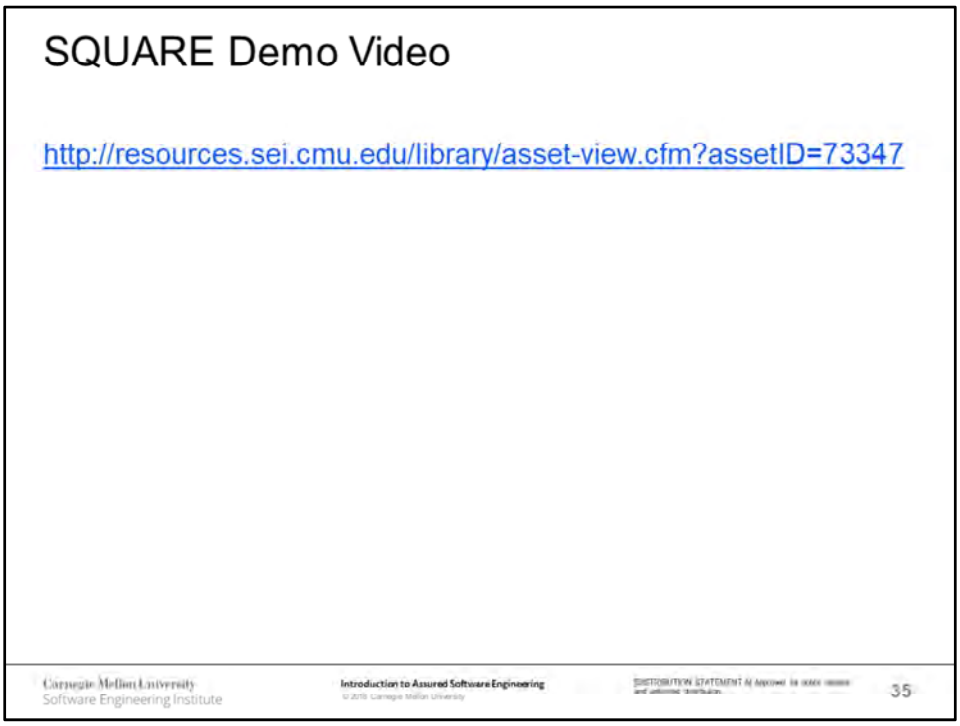
Describe what SQUARE-Lite is. The five steps are extracted from the nine steps of the SQUARE process. SQUARE-Lite can be used by organizations that already have a requirements engineering process, and just want to fit security requirements into it, or by organizations that have not yet bought into the full SQUARE but still want some of the benefits.



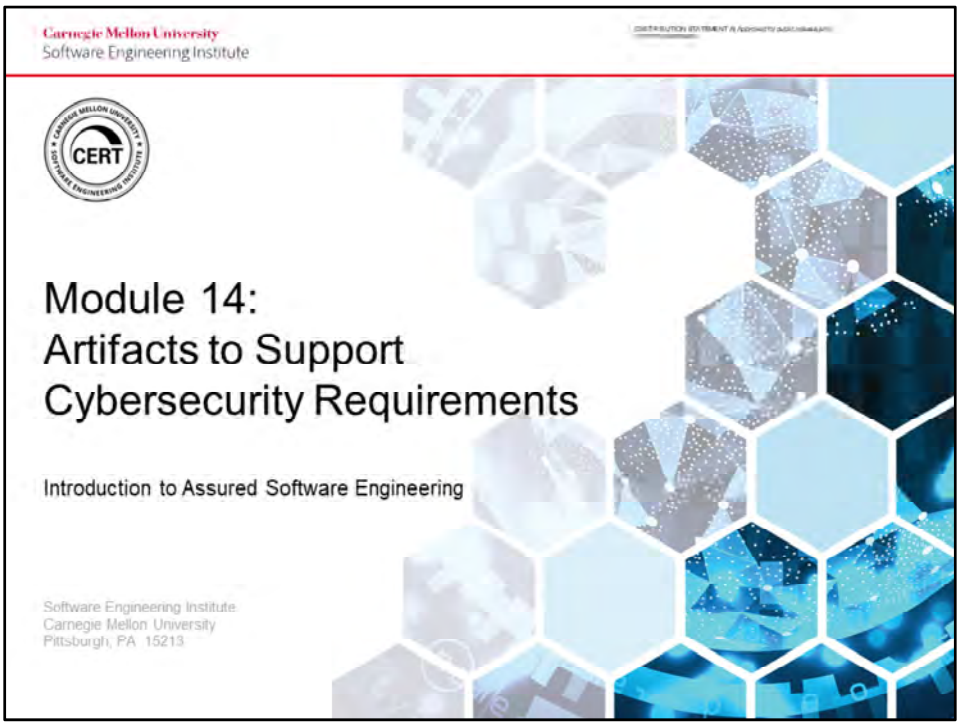
Note that when the SQUARE tool was developed, we were able to provide automated support for traceability. However, this traceability, which would normally extend to test cases, is limited to the 9 steps as implemented in the tool. Of course traceability to test cases could be done manually, OR under the umbrella of another tool.



Summarize the definition and the process.
Quickly go over the nine steps again.
Mention SQUARE-Lite as an alternative.
Talk about the next lecture – SQUARE in detail.



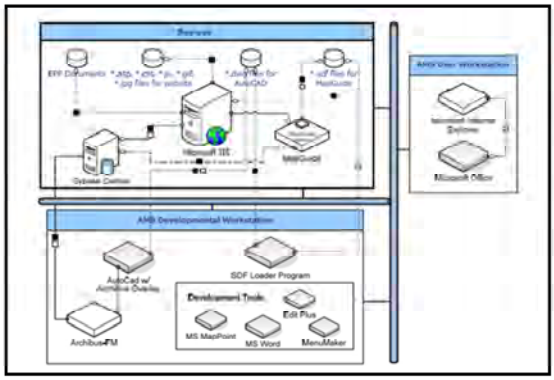
Use of the video is optional


Carnegie Mellon University
Software Engineering Institute
CERT
Module 14:
Artifacts to Support
Cybersecurity Requirements
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Develop Artifacts (corresponds to SQUARE Step 3)

Types of artifacts to collect

- System architecture diagrams (should exist for the project)



Carnegie Mellon University
Software Engineering Institute

Introduction to Assured Software Engineering

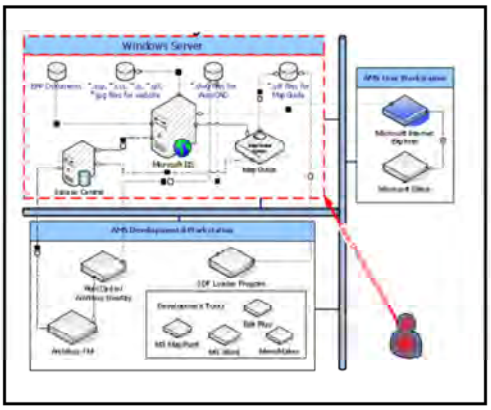
3

Collect or create a set of artifacts of the system. These artifacts will help the RE team better understand the system and its different vulnerabilities.

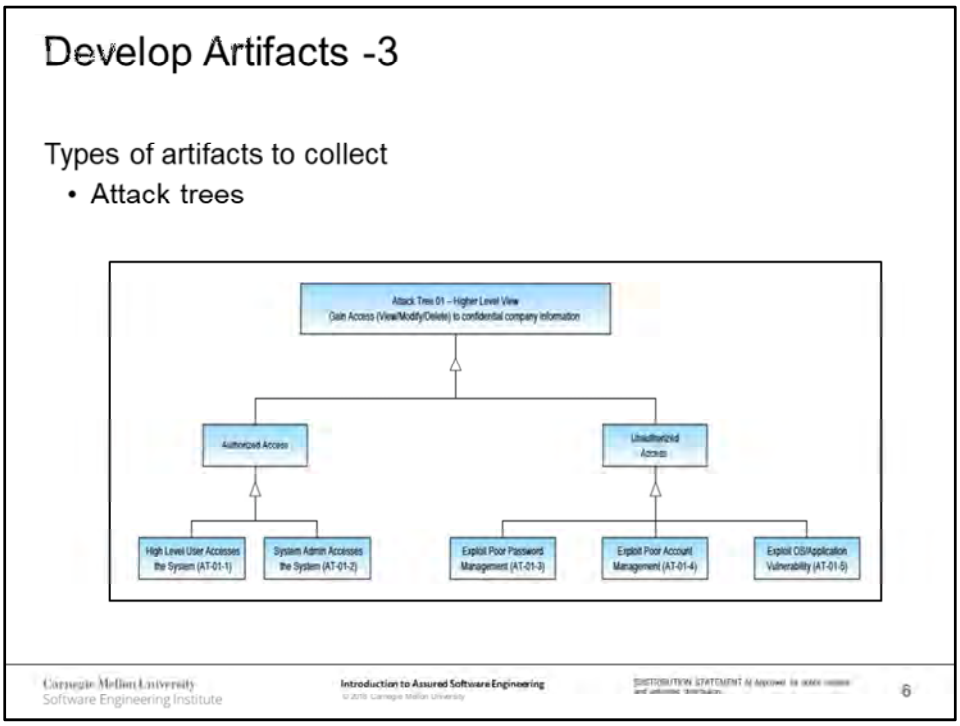
Develop Artifacts -2

Types of artifacts to collect

- Misuse case scenarios/ diagrams (exemplar misuse cases)



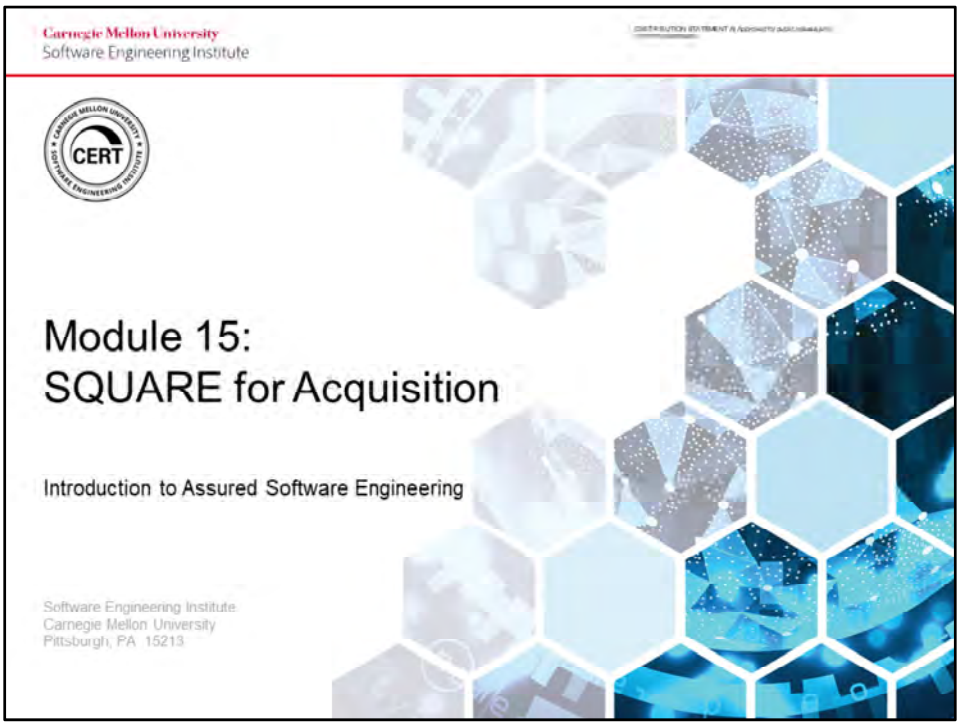
Note that projects that have not considered security will not have misuse cases. In those cases, the RE team will have to work with the stakeholders to develop the needed misuse cases.

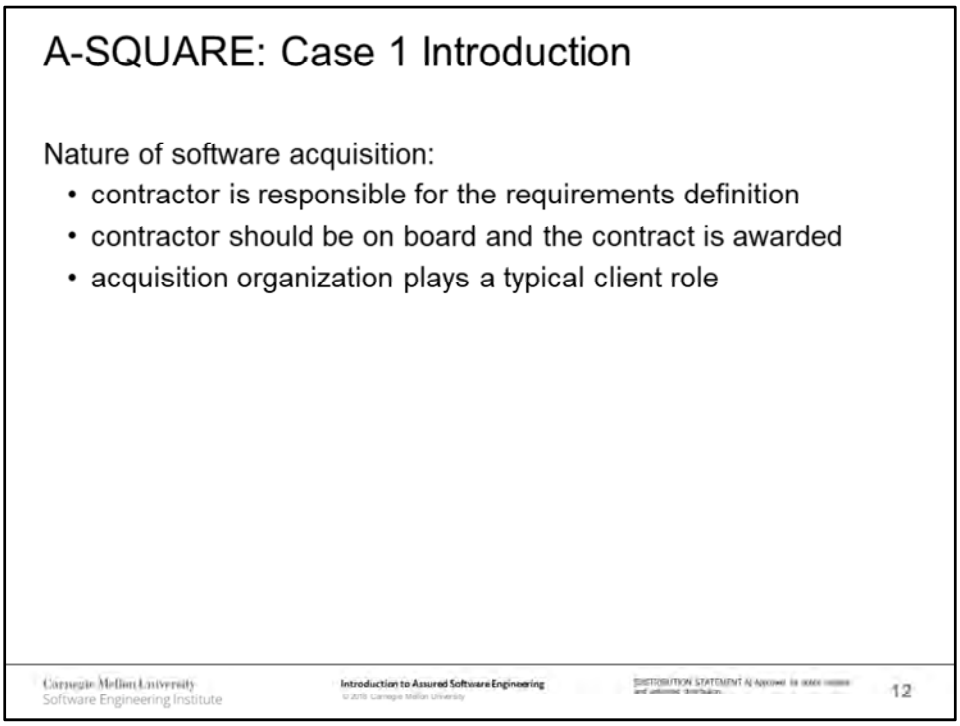


Again, existence of attack trees implies that some attention has been paid to security. Attack trees are a formal hierarchical way of describing a threat to a system based on the type of attack. Goals are at the root and ways to achieve it are the leaves.

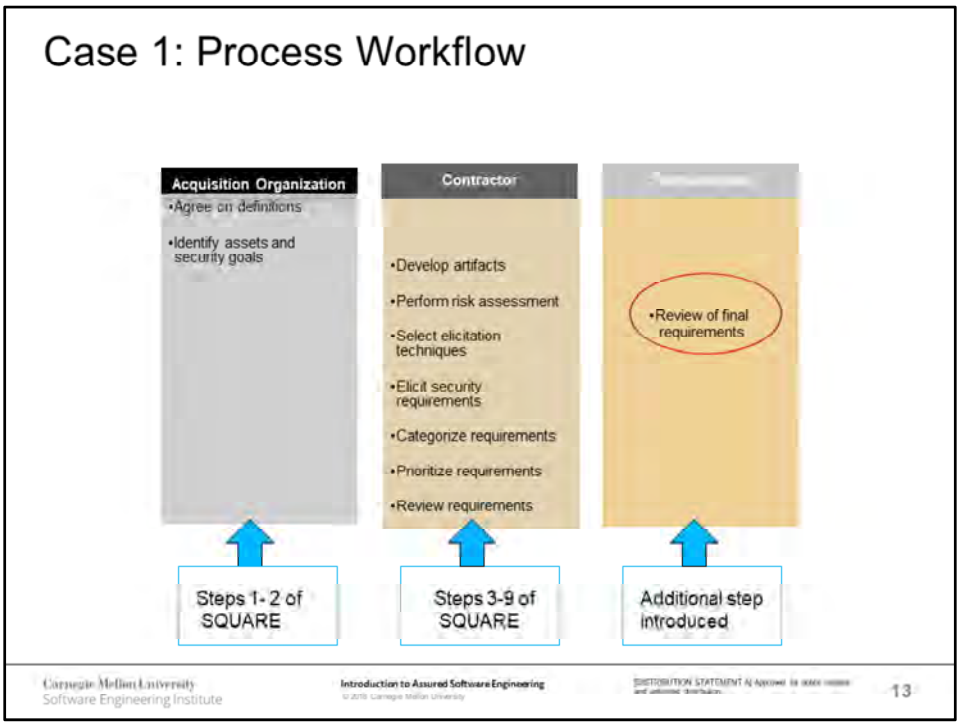
Explain the symbols used – and, or, scenario, and connector (no 'and's in the example)

Another type of artifact to collect (not on a slide): standardized templates and forms

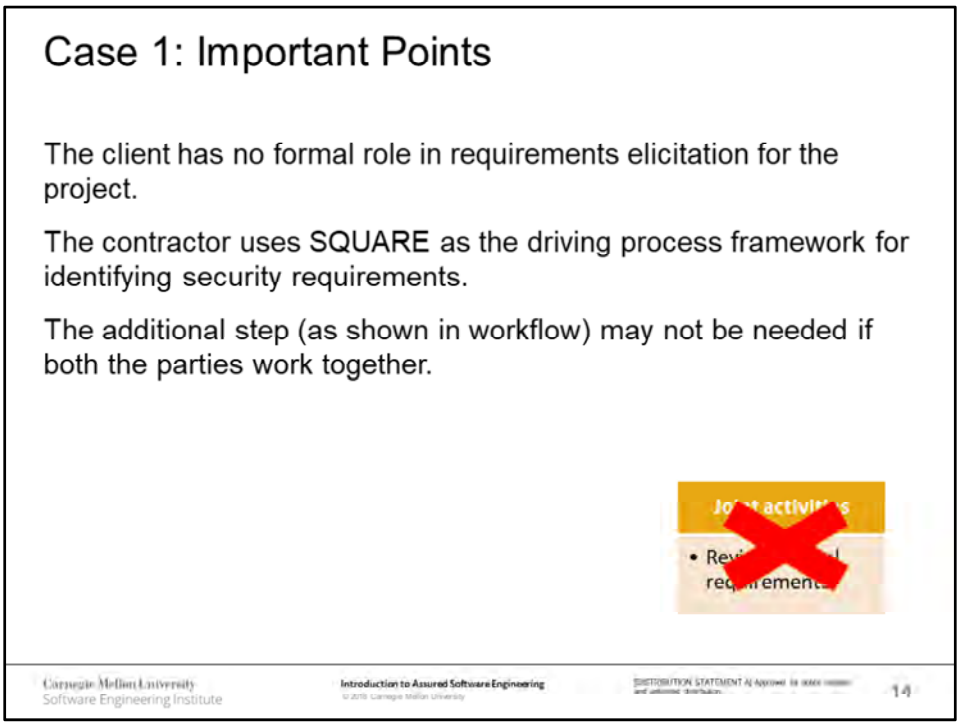

Carnegie Mellon University
Software Engineering Institute
CERT
Module 15:
SQUARE for Acquisition
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



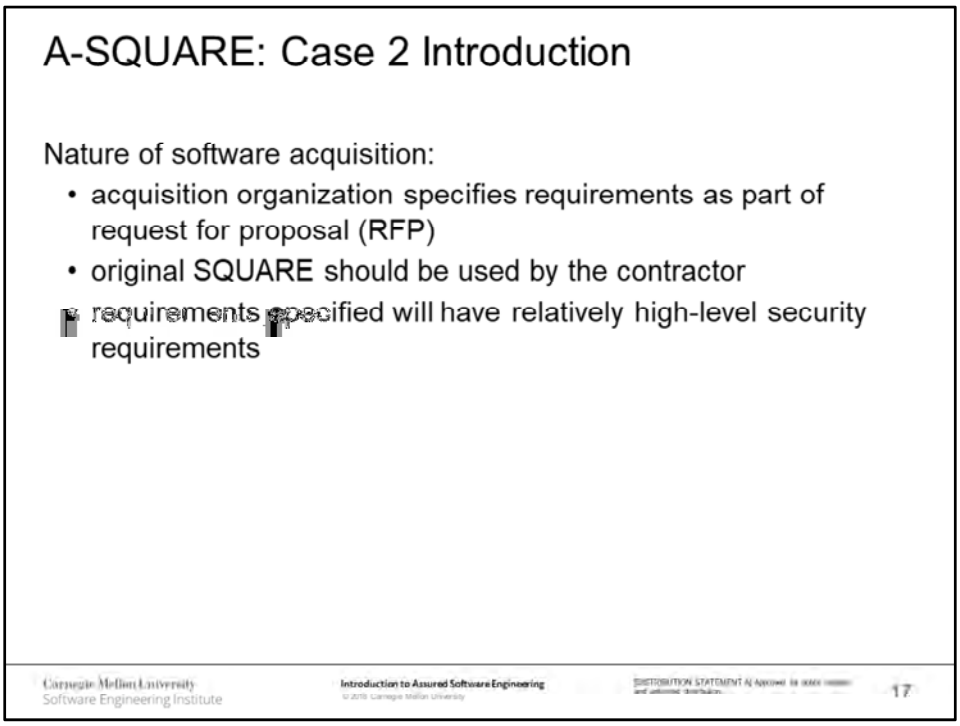
In this example, the contractor is responsible for requirements identification. We have used SQUARE as the underlying method, but depending on how the contract is written, presumably the contractor could use another method to identify the security requirements.



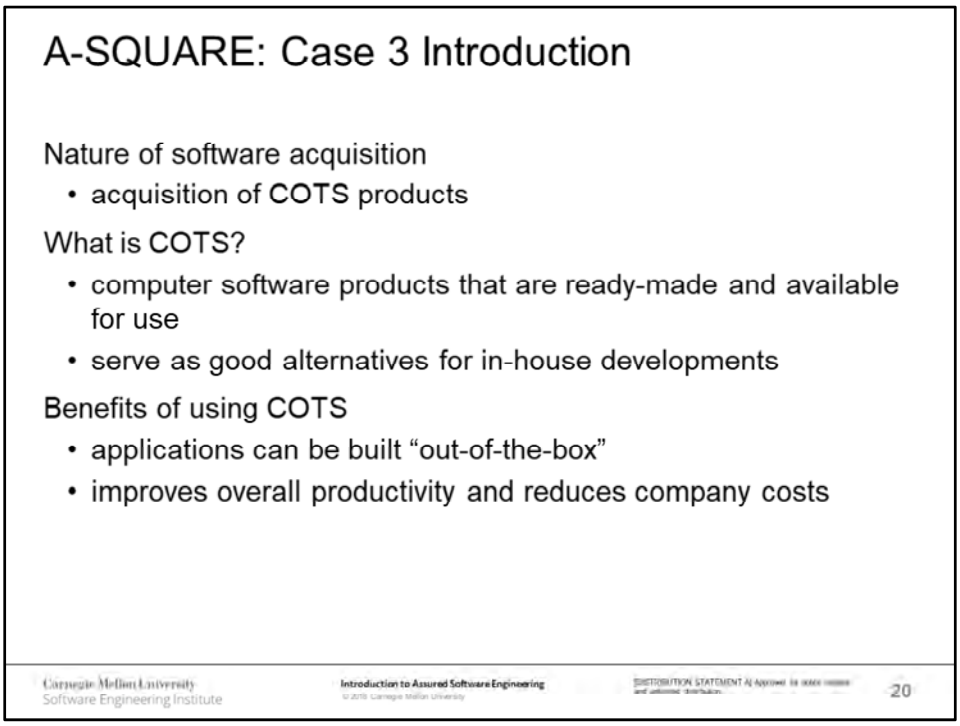
If SQUARE is used throughout, Steps 3-9 are performed by the contractor. Otherwise the contractor may use some other method for identifying security requirements. This presumes that the contract award has been made and the contractor is on board.



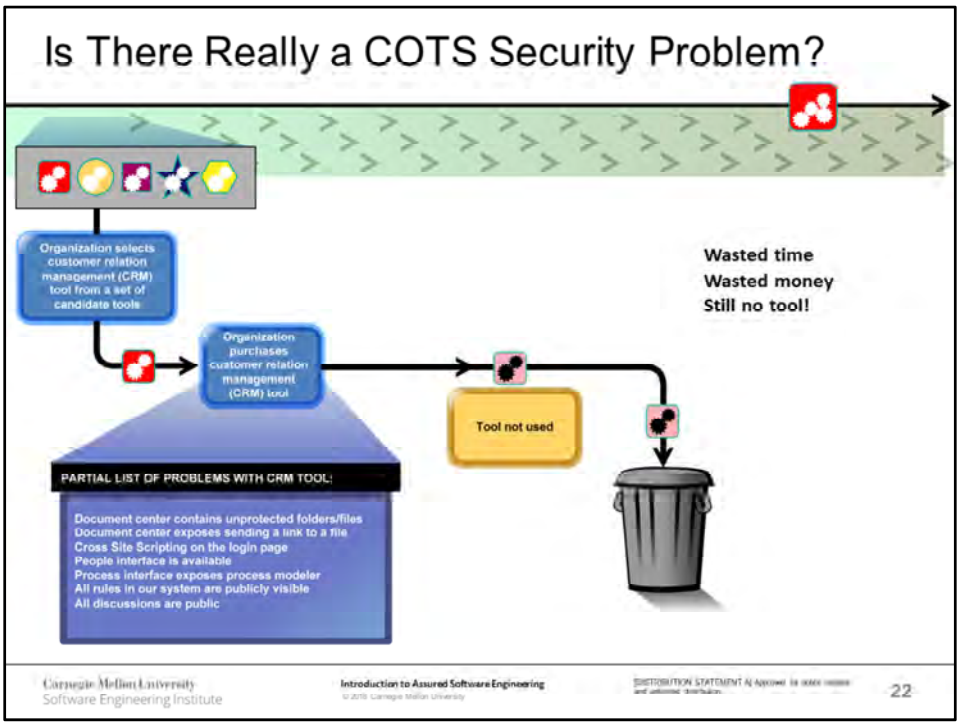
The acquisition organization has the typical client role in this example and reviews the resultant requirements, but does not necessarily specify which method to use in developing the requirements. It's important to note the client involvement in Steps 1, 2, and 10. Note that if the acquisition organization works side by side with the contractor, the separate review in Step 10 could be eliminated, as the client inputs would already be taken into account in the earlier steps.



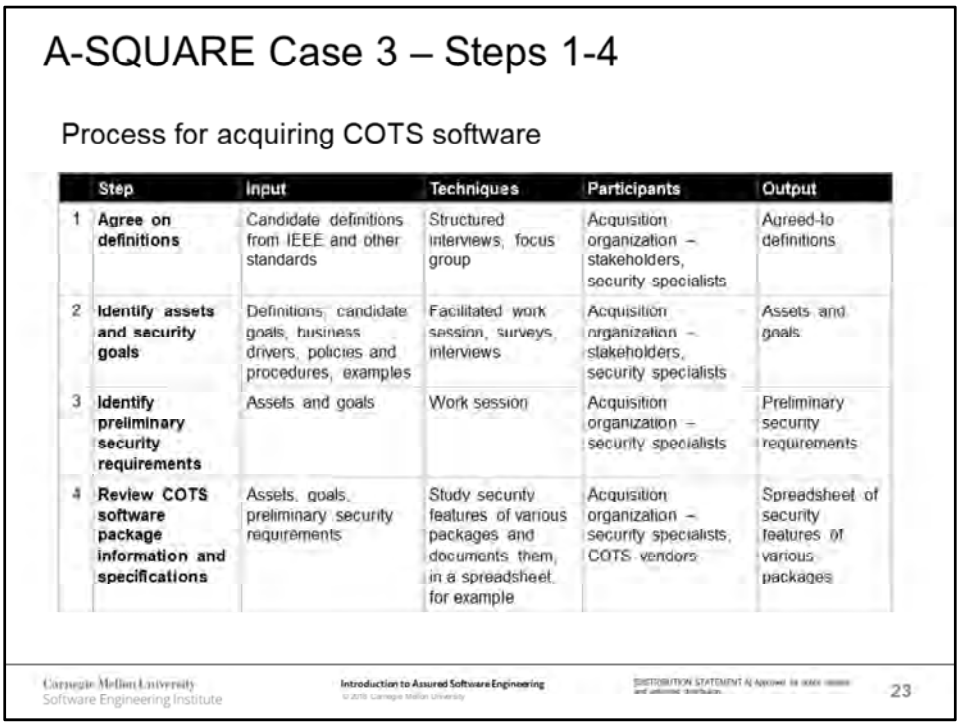
If the acquisition organization specifies the requirements as part of the request for proposal (RFP), then the original SQUARE for development should be used. Note that these may be relatively high-level security requirements that result from this exercise, since the acquisition organization may be developing the requirements in the absence of a broader system context. Also, the acquisition organization will want to avoid identifying requirements at an implementation level of granularity as that will overly constrain the contractor.



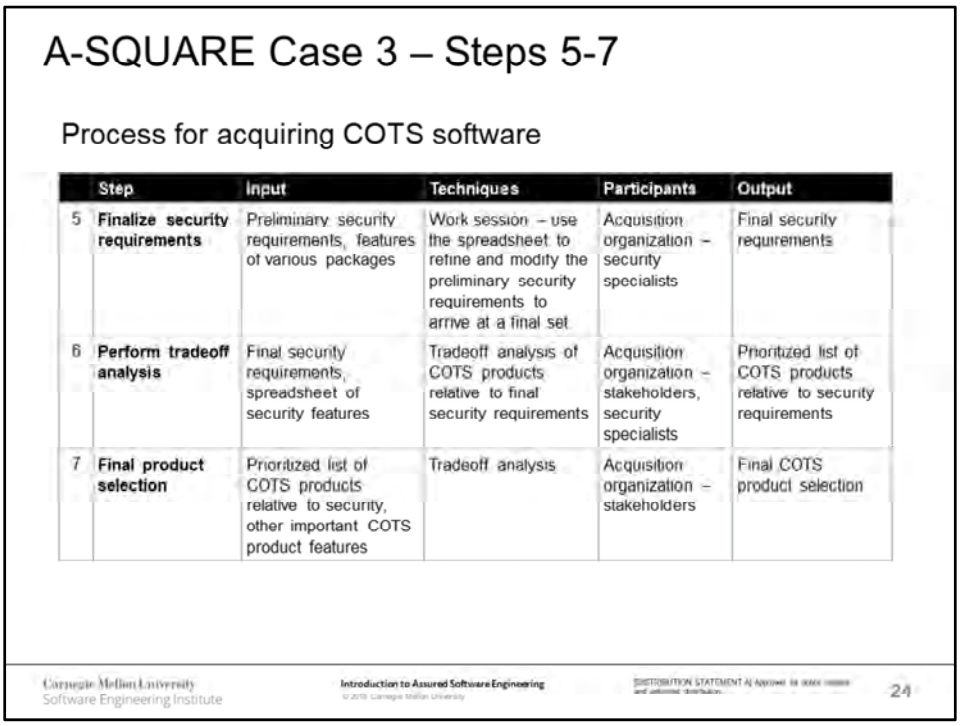
In acquisition of COTS software, the organization will have to develop a list of requirements for the software and compare those requirements with the software packages under consideration. Securityrequirements may need to be prioritized together with other requirements. Compromises and tradeoffs mayneed to be made, and the organization may have to figure out how to satisfy some security requirements outside the software itself—for example with system level requirements, security policy, or physical security. The requirements themselves are likely to be high-level requirements that map to security goals rather than detailed requirements used in software development.



Note that in acquiring COTS software, organizations often do minimal tradeoff analysis and may not consider security requirements at all, even when they do such tradeoff analysis. The acquiring organization will need to consider “must have” versus “nice to have” security requirements. It is also the case that reviewing the security features of specific offerings may help the acquiring organization to identify the security requirements that are important to them.



Here is a complete process. In this case, we envision an iteration. Once the preliminary security requirements are identified, the acquirer has to review available COTS products, refine and finalize the security requirements and then do tradeoff analysis among the available products. Then a final product spec can be written and a product selected.



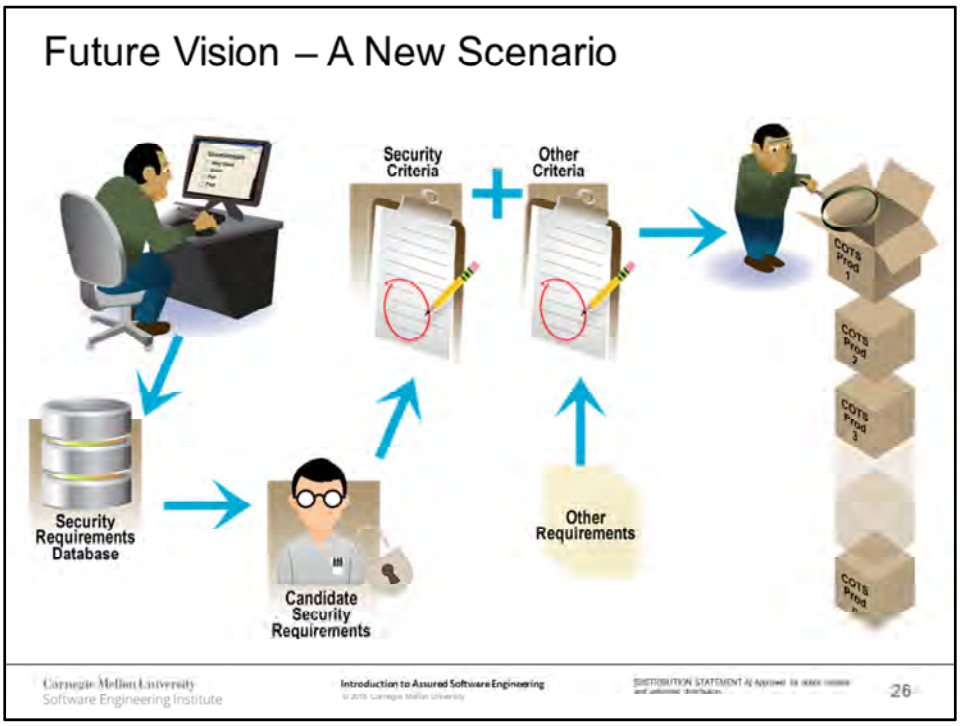
A-SQUARE Case 3 – Steps 5-7

Process for acquiring COTS software

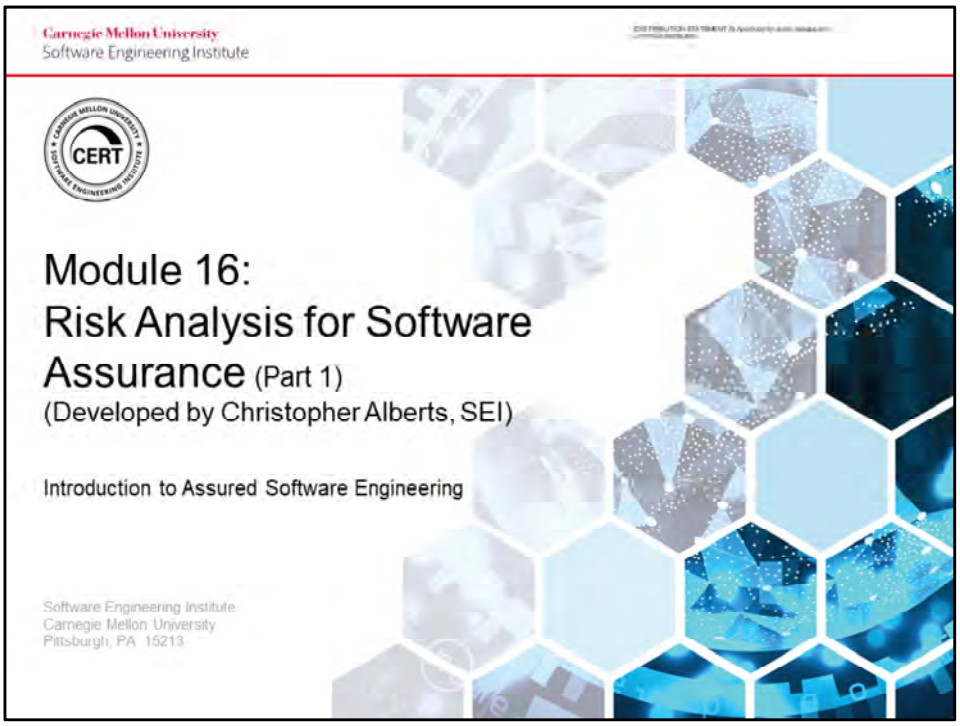
	Step	Input	Techniques	Participants	Output
5	**Finalize security requirements**	Preliminary security requirements, features of various packages	Work session – use the spreadsheet to refine and modify the preliminary security requirements to arrive at a final set	Acquisition organization – security specialists	Final security requirements
6	**Perform tradeoff analysis**	Final security requirements, spreadsheet of security features	Tradeoff analysis of COTS products relative to final security requirements	Acquisition organization – stakeholders, security specialists	Prioritized list of COTS products relative to security requirements
7	**Final product selection**	Prioritized list of COTS products relative to security, other important COTS product features	Tradeoff analysis	Acquisition organization – stakeholders	Final COTS product selection

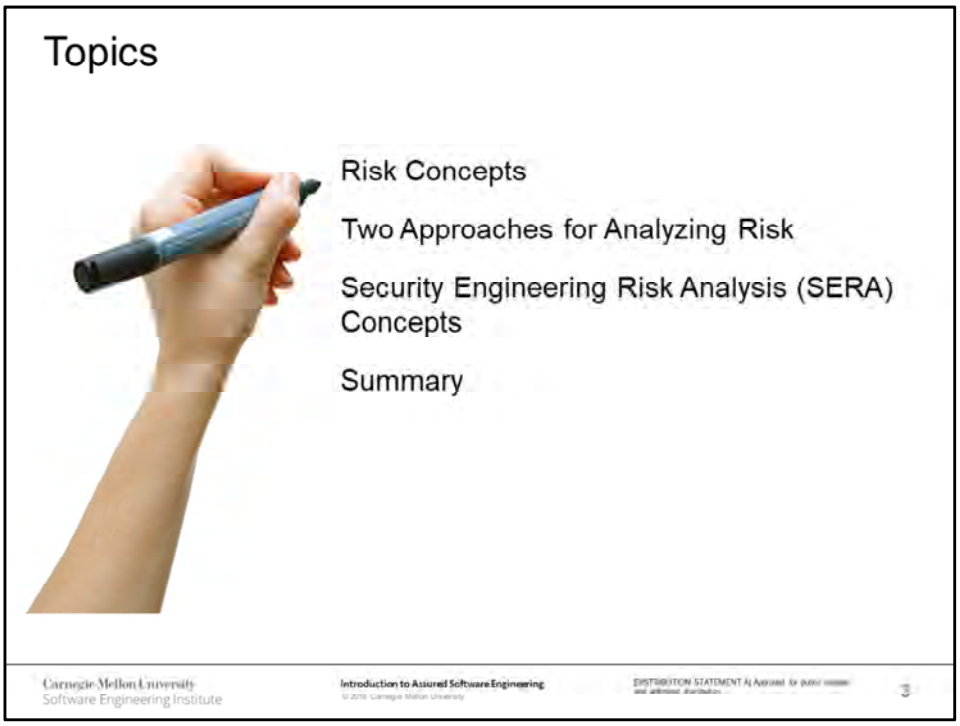
Table Templates

- We have provided some pre-formatted tables as guides for data display.
- Columns and Rows are independent and can be edited, added, deleted and formatted to suit your individual needs.
- Table Templates are for guidance only.



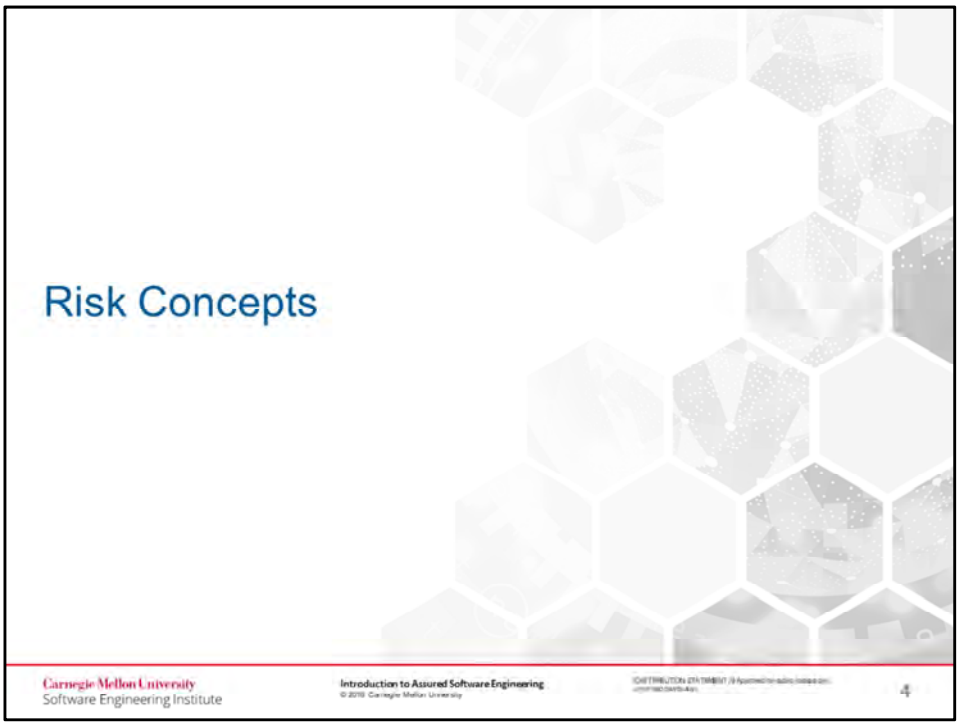
Since acquisition of COTS products is often done by buyers without security expertise, one future vision is to provide such buyers with a tool. Based on answers to a questionnaire on the intended usage and environment of the product, a set of security requirements could be provided to the buyer by the tool, and those requirements could be refined and merged with other requirements to come up with a complete set of COTS selection criteria. This is a future research idea that has not yet been implemented.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 16:
Risk Analysis for Software Assurance (Part 1)
(Developed by Christopher Alberts, SEI)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

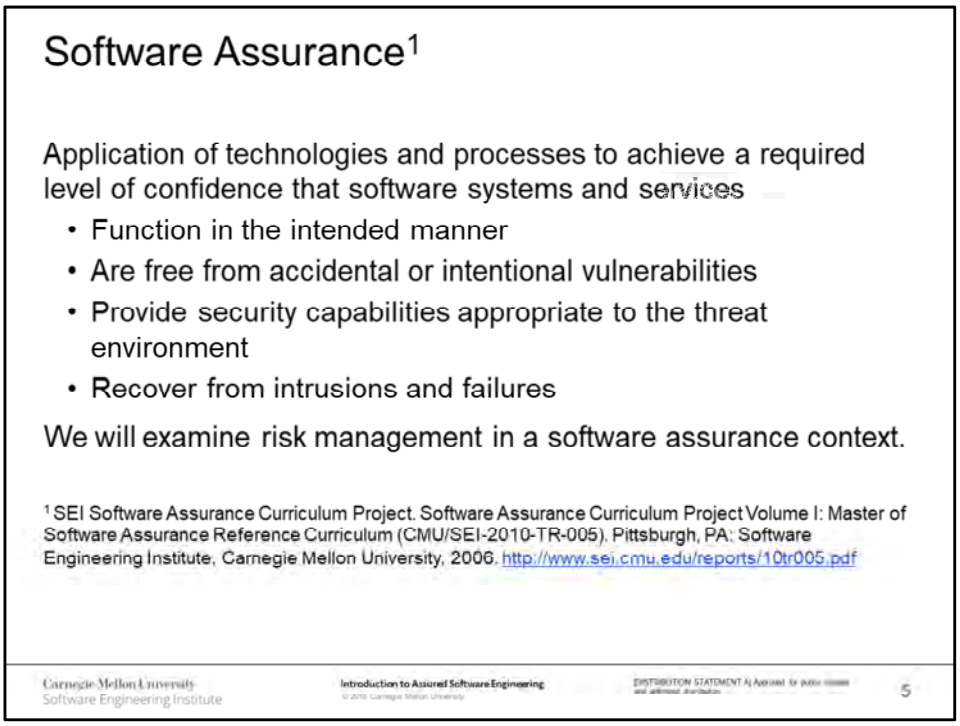


For risk analysis, the following topics will be covered:

- Risk Concepts—Provides a brief overview of important risk management definitions and ideas.
- Two Approaches for Analyzing Risk—Differentiates between mission risk (also referred to as *systemic risk*) and event risk (also referred to as *tactical risk*)
- Security Engineering Risk Analysis (SERA) Concepts—Presents an detailed walkthrough of the SERA method for analyzing security risks during system acquisition and development
- Summary—Highlights the key points of the presentation



Topic 1: Risk Concepts
This topic provides a brief overview of important risk management definitions and ideas.

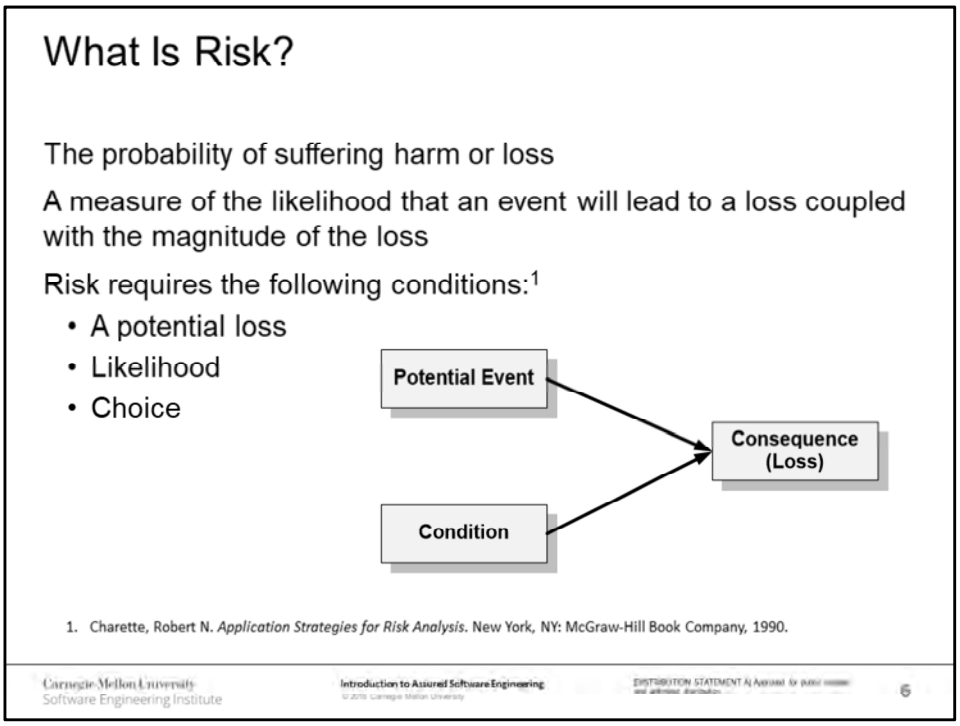


Software assurance is defined as the application of technologies and processes to achieve a required level of confidence that software systems and services

- Function in the intended manner
- Are free from accidental or intentional vulnerabilities
- Provide security capabilities appropriate to the threat environment
- Recover from intrusions and failures

This topic examines risk-management concepts from a software-assurance perspective. More specifically, this topic is focused on

- applying risk analysis during early life-cycle activates
- using risk analysis to specify security requirements



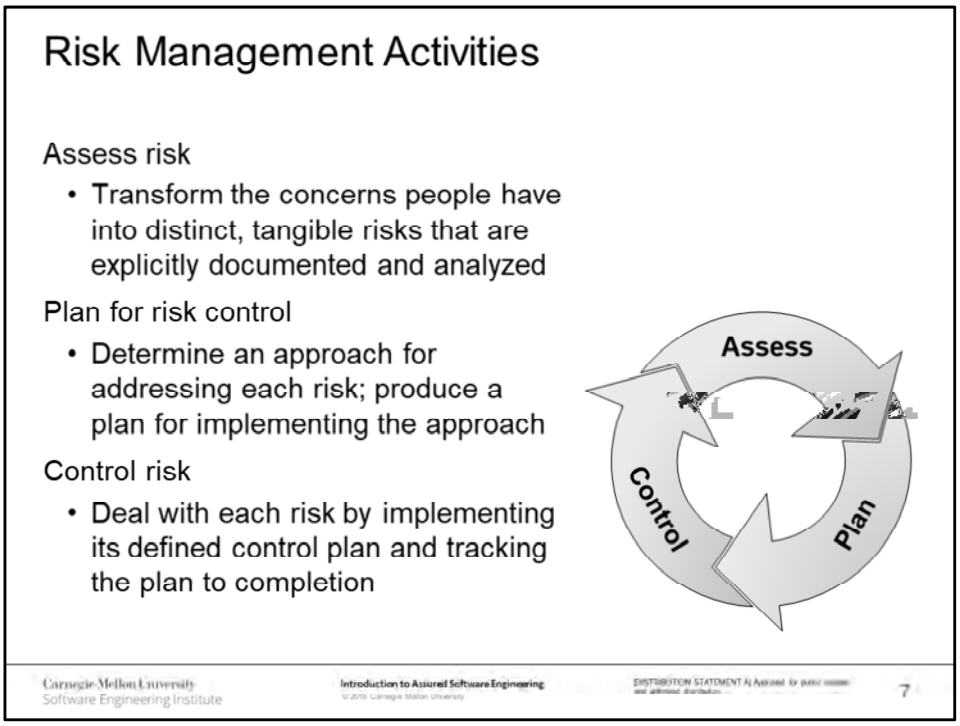
The essence of risk, no matter what the domain, can be succinctly captured by the following definition: *Risk is the probability of suffering harm or loss*.

The figure on the slide illustrates the three components of risk:

- *potential event* – an act, occurrence, or happening that alters current conditions and leads to a loss
- *condition* – the current set of circumstances that leads to or enables risk
- *consequence* – the loss that results when a potential event occurs; the loss is measured in relation to the status quo (i.e., current state)

From the risk perspective, a condition is a passive element. It exposes an entity (e.g., project, system) to the loss triggered by the occurrence of an event. However, by itself, a risk condition will *not* cause an entity to suffer a loss or experience an adverse consequence; it makes the entity *vulnerable* to the effects of an event.

Example of Risk: A project team is developing a software-reliant system for a customer. The team has enough people with the right skills to perform its tasks and complete its next milestone on time and within budget (status quo). However, the team does not have redundancy among team members' skills and abilities (condition). If the team loses people with certain key skills (potential event), then it will not be able to complete its assigned tasks (consequence/loss). This puts the next milestone in jeopardy, which is a loss when measured in relation to the status quo (on track to achieve the next milestone). However, if none of the team members leaves or is reassigned (i.e., the event does not occur), then the project should suffer no adverse consequences. Here, the condition enables the event to produce an adverse consequence or loss.



Risk management is a systematic approach for minimizing exposure to potential losses. It provides a disciplined environment for

- continuously assessing what could go wrong (i.e., assessing risks)
- determining which risks to address (i.e., setting mitigation priorities)
- implementing actions to address high-priority risks and bring those risks within tolerance

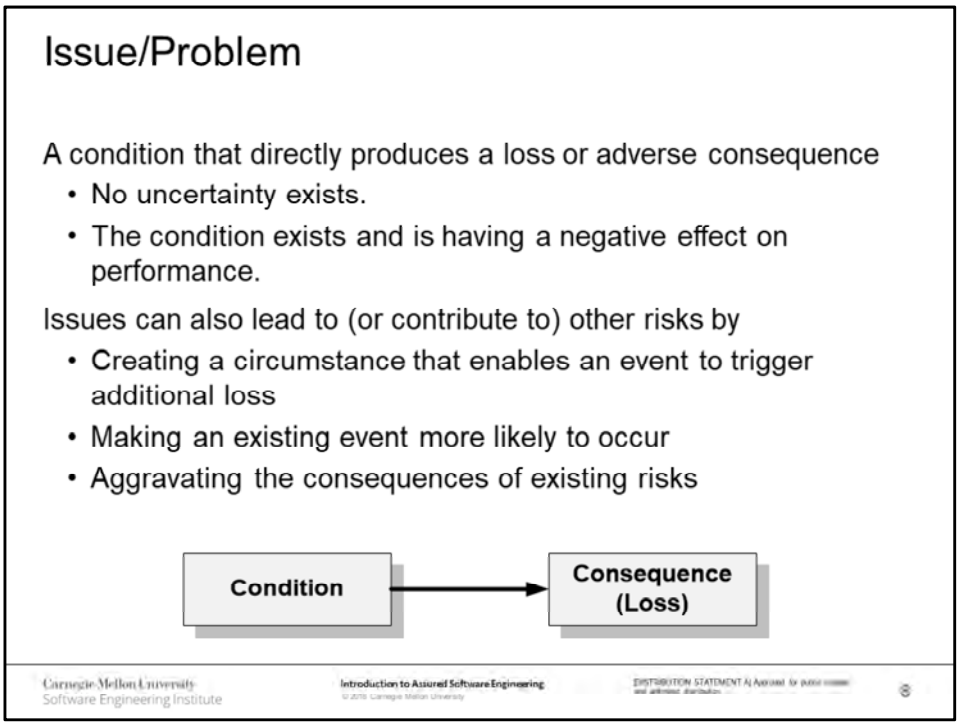
The main goal or risk management is to provide decision makers

- with the information they need
- when they need it
- in the right form

If decisions are not influenced by risk analysis activities, then risk analysis provides no added value.

The figure on the slide illustrates the three core risk management activities:

- *assess risk*—transform the concerns people have into distinct, tangible risks that are explicitly documented and analyzed
- *plan for controlling risk*—determine an approach for addressing each risk; produce a plan for implementing the approach
- *control risk*—deal with each risk by implementing its defined control plan and tracking the plan to completion



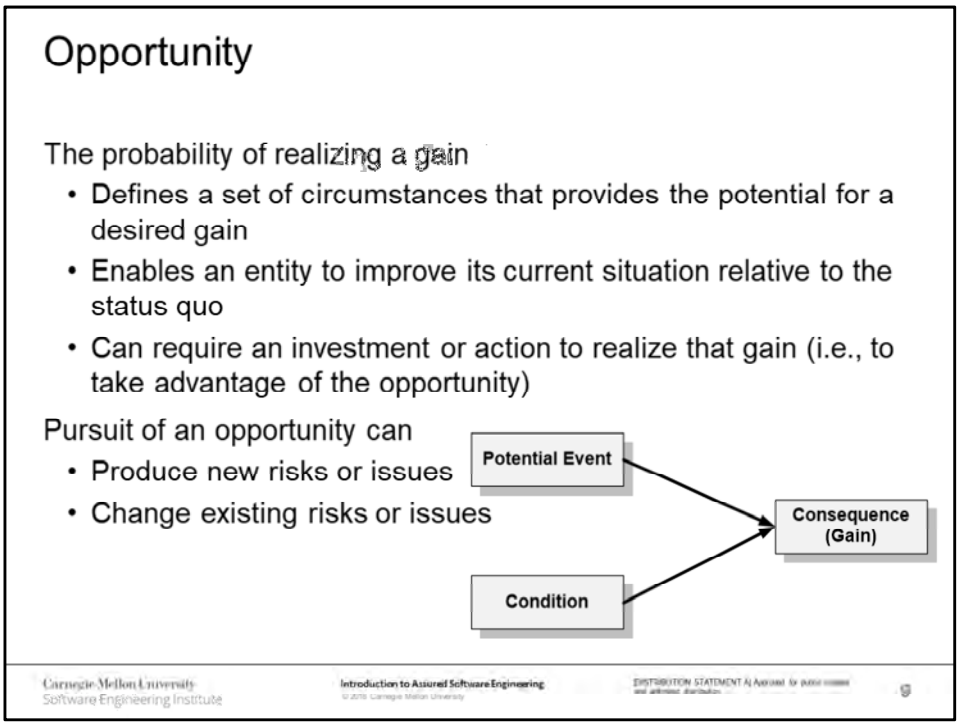
One of the fundamental conditions of risk is uncertainty regarding its occurrence. A risk, by definition, might or might not occur. In contrast, an *issue* (also referred to as a *problem*) is a condition that directly produces a loss or adverse consequence. With an issue, no uncertainty exists—the condition exists and is having a negative effect on performance.

Issues can also lead to (or contribute to) other risks by

- creating a circumstance that enables an event to trigger additional loss
- making an existing event more likely to occur
- aggravating the consequences of existing risks

People do not always find it easy to distinguish between an issue and the future risk posed by that issue (if left uncorrected). This confusion can result in issues being documented in a risk database and being treated like risks (and vice versa). Management must take great care to ensure that their approaches for managing issues and risks are integrated appropriately and understood by both management and staff.

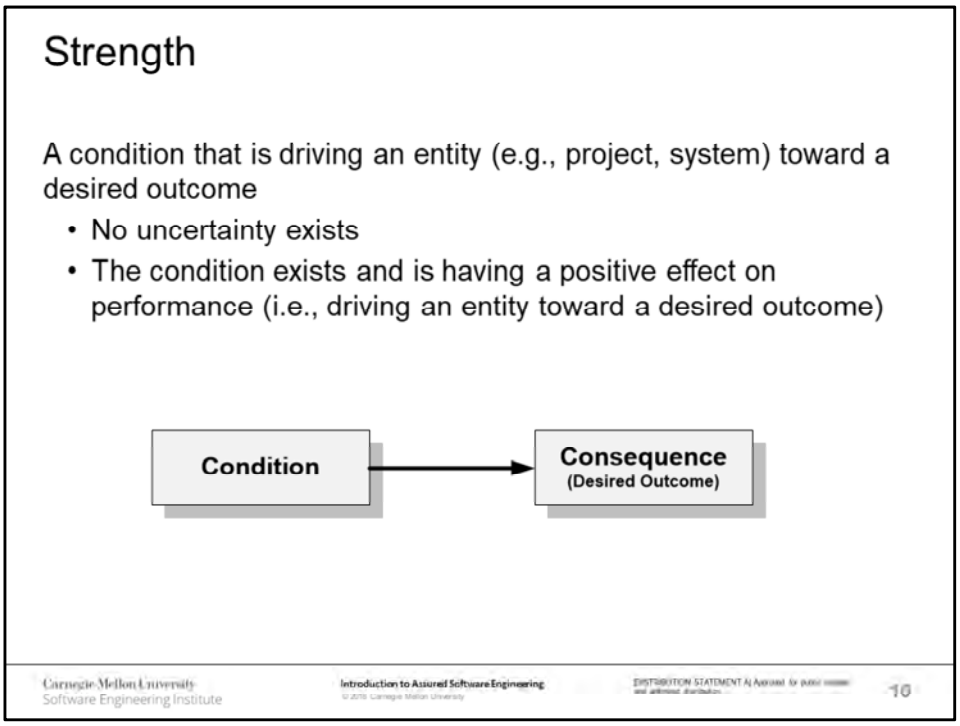
Example of Issue/Problem: A project team is developing a software-reliant system for a customer. The team does not have enough people with the right skills to perform the team's assigned tasks (condition). As a result, the team will not be able to complete all of its assigned tasks before the next milestone (consequence/loss). No event is required for the loss to occur, which distinguishes and issue/problem from a risk.



Risk is focused on the potential for loss; it does not address the potential for gain. The concept of opportunity is focused on the potential for a positive outcome. An *opportunity* is the probability of realizing a gain. It thus enables an entity to improve its current situation relative to the status quo.

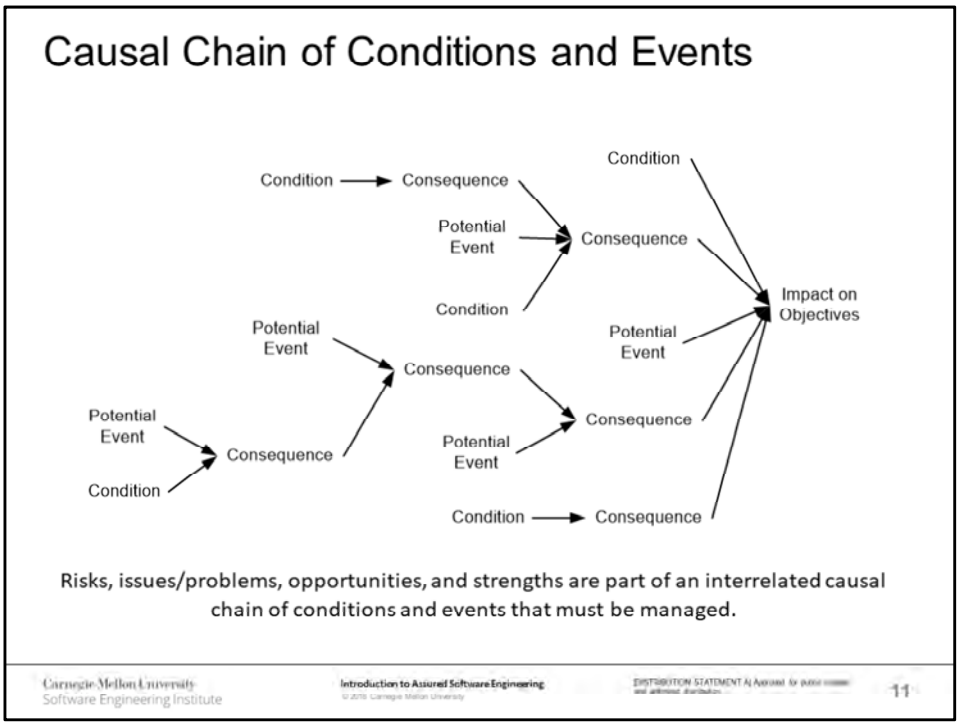
Very often, an opportunity is focused on the gain that could be realized from an allocation or reallocation of resources. It defines a set of circumstances that provides the potential for a desired gain and often requires an investment or action to realize that gain (i.e., to take advantage of the opportunity). Pursuit of an opportunity can produce new risks or issues, and it can also change existing risks or issues.

Example of Opportunity: A project team is developing a software-reliant system for a customer. Current status and quality reports indicate that the team is not on track to achieve its next milestone (status quo). Another project in the company has just delivered its product to its customer, and its team members will be made available to projects throughout the company (condition). If the project manager brings additional personnel who have the right knowledge, skills, and abilities onto the project (event), then the team might be able to increase its productivity and be in position to meet its next milestone (consequence/gain). Here, the gain is improved performance in relation to the status quo.



A *strength* is a condition that is driving an entity (e.g., project, system) toward a desired outcome. With a strength, no uncertainty exists—the condition exists and is having a positive effect on performance (i.e., driving an entity toward a desired outcome).

Example of Strength: A project team is developing a software-reliant system for a customer. The team has enough people with the right skills to perform the team's assigned tasks and has enough redundancy in skills needed to meet the next milestone (condition). Its people are its strength. As a result, the team is positioned to execute its tasks and activities effectively and efficiently, putting the project in position to achieve its next milestone.

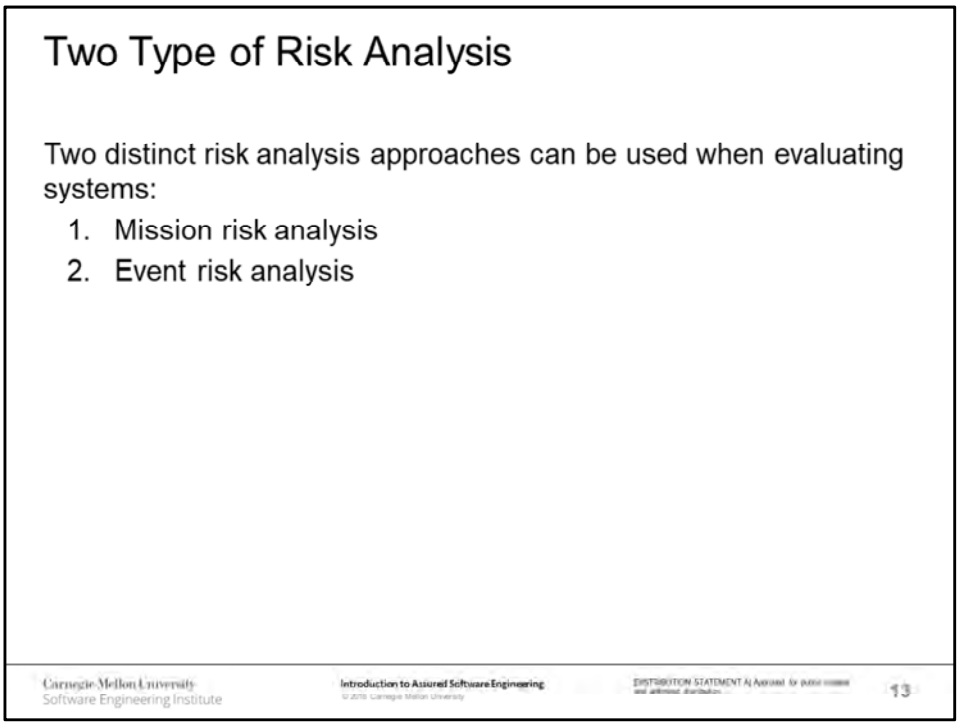


The success or failure of an activity or endeavor is influenced by the range of circumstances that are present. The figure on the slide depicts a causal chain of conditions and events that affect whether an activity will achieve a desired set of objectives. This causal chain includes

- strengths that are driving the activity toward a successful outcome
- issues or problems that are driving the activity toward a failed outcome
- risks that could degrade performance and make a failed outcome more likely
- opportunities that could improve performance and make a successful outcome more likely

Effective risk management requires navigating through this causal chain, assessing the current potential for loss, and implementing strategies for minimizing the potential for loss.

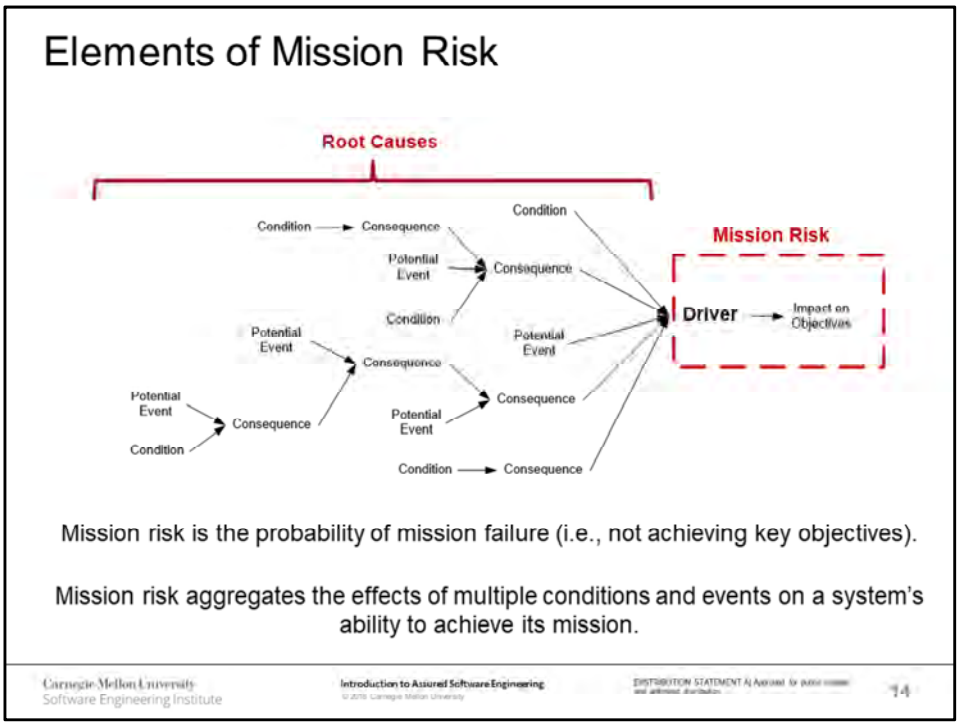
Topic 2: Two Approaches for Analyzing Risk
This topic differentiates between mission risk (also referred to as *systemic risk*) and event risk (also referred to as *tactical risk*).



Two distinct risk analysis approaches can be used when evaluating systems:

- event risk analysis
- mission risk analysis

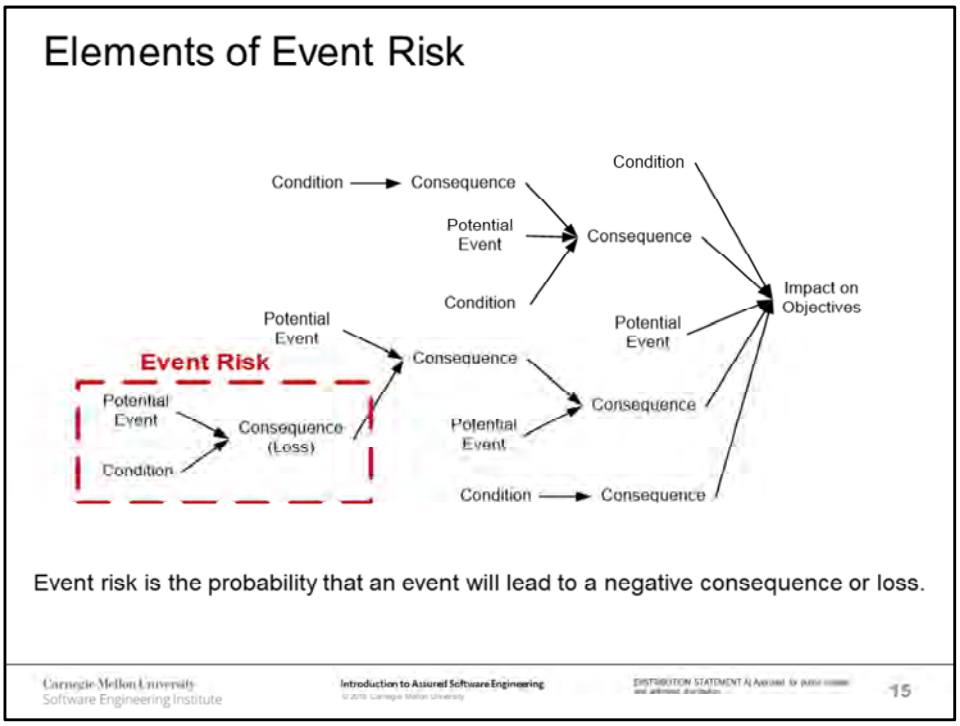
Each type is briefly explored in this topic.



From the mission perspective, risk is defined as the probability of mission failure (i.e., not achieving key objectives). Mission risk (also referred to as systemic risk) aggregates the effects of multiple conditions and events on a system's ability to achieve its mission.

Mission risk analysis provides a holistic view of the risk to an interactively complex, socio-technical system. The first step in this type of risk analysis is to establish the objectives that must be achieved. The objectives define the desired outcome, or "picture of success," for a system. Next, systemic factors that have a strong influence on the outcome (i.e., whether or not the objectives will be achieved) are identified. These systemic factors, called *drivers*, are important because they define a small set of factors that can be used to assess a system's performance and gauge whether it is on track to achieve its key objectives. The drivers are then analyzed, which enables decision makers to gauge the overall risk to the system's mission.

A driver is a construct that is used to aggregate the effects of multiple conditions and events in order to determine their combined influence on the mission's key objectives. Each driver *directly* influences whether or not objectives will be achieved. The conditions and events within the causal chain are considered to be the root causes of mission risk.

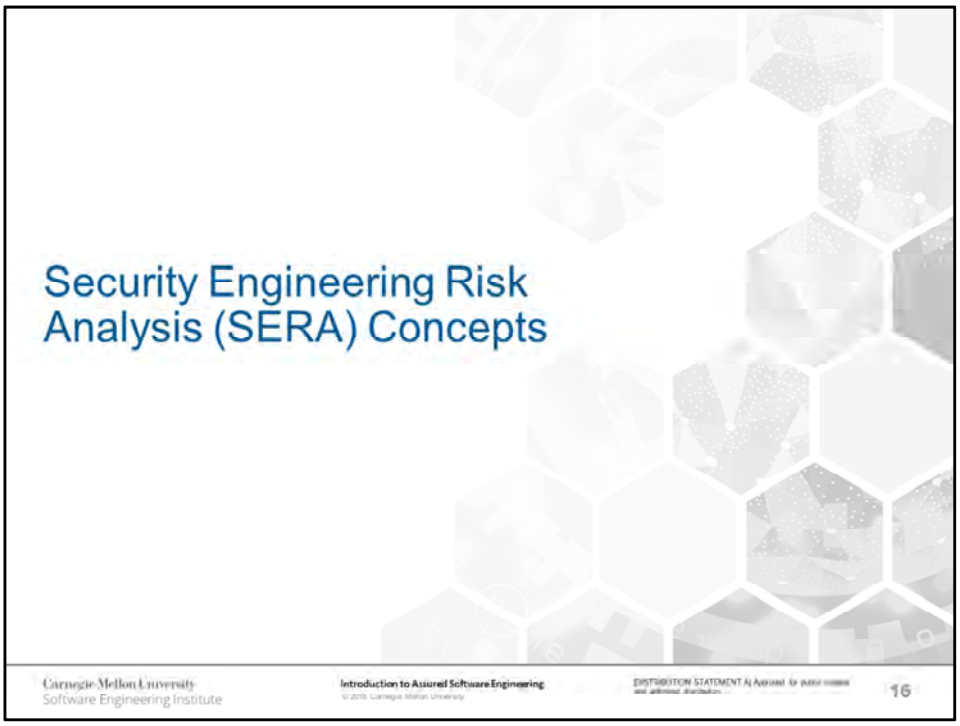


From the tactical perspective, risk is defined as the probability that an event will lead to a negative consequence or loss. The figure on the slide shows the causal chain of conditions and events that was introduced in the previous topic. As depicted in the figure, event risk (also referred to as tactical risk) is focused on the risk that is triggered by an individual event.

The basic goal of event risk analysis is to evaluate a system's components for potential failures. Event risk analysis is based on the principle of system decomposition and component analysis. The first step of this approach is to decompose a system into its constituent components. The individual components are then prioritized, and a subset of components is designated as being critical. Next, the risks to each critical component are analyzed.

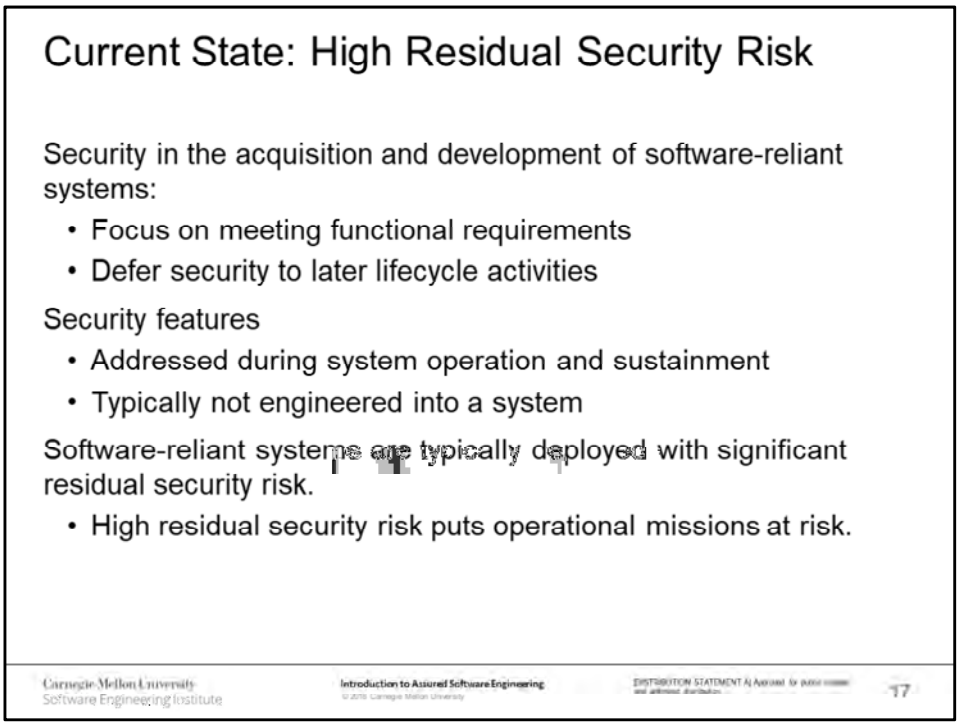
Event risk analysis enables stakeholders to (1) determine which components are most critical to a system and (2) analyze ways in which those critical components might fail (i.e., analyze the risk to critical components). Stakeholders can then implement effective controls designed to mitigate those potential failures. Because of its focus on preventing potential failures, event risk analysis has been applied extensively within the discipline of systems engineering.

This presentation is focused on the Security Engineering Risk Analysis (SERA) method, which incorporates the principles of event (or tactical) risk analysis.

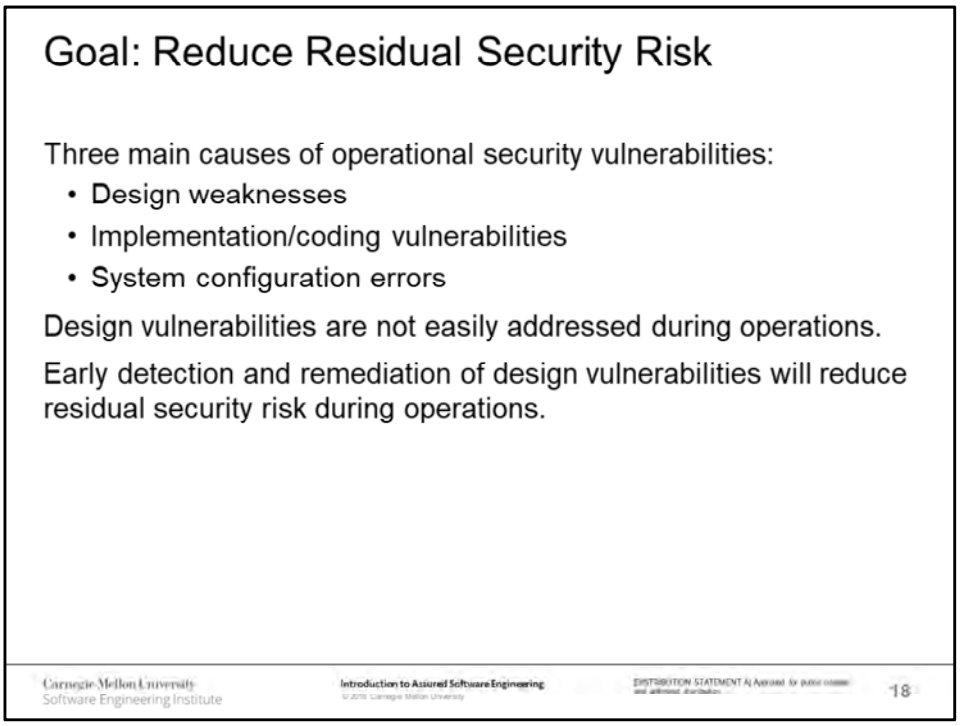


Topic 3: Security Engineering Risk Analysis (SERA) Concepts

This topic presents an detailed walkthrough of the SERA method for analyzing security risks during system acquisition and development.

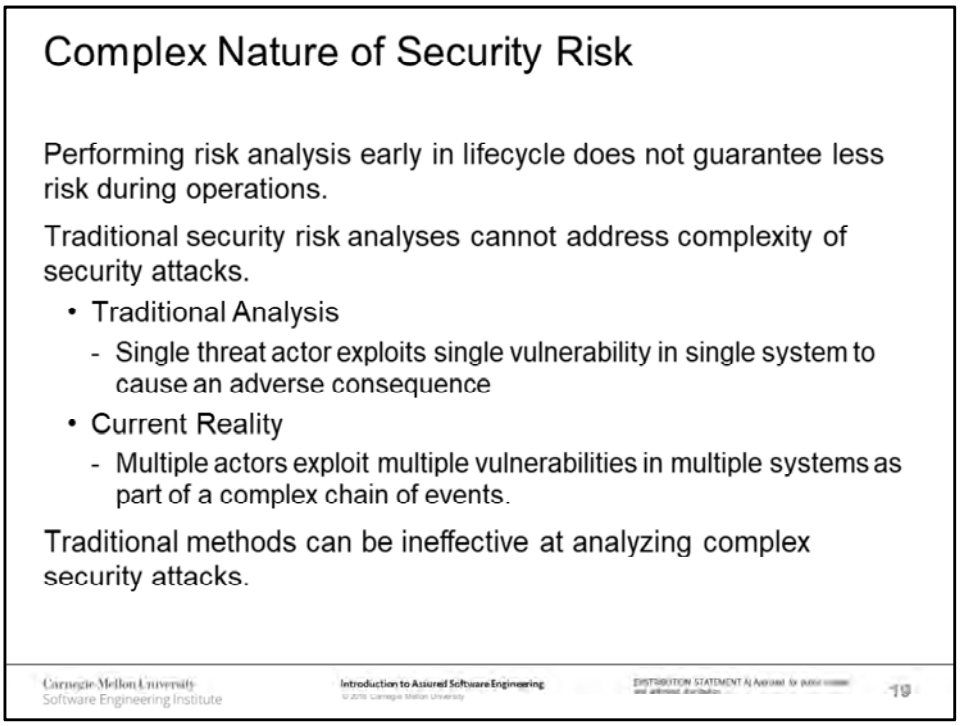


During the acquisition and development of software-reliant systems, program personnel normally focus on meeting functional requirements, often deferring security to later life-cycle activities. Security features are usually addressed during system operation and sustainment rather than being engineered into a system. As a result, many software-reliant systems are deployed with significant residual security risk.

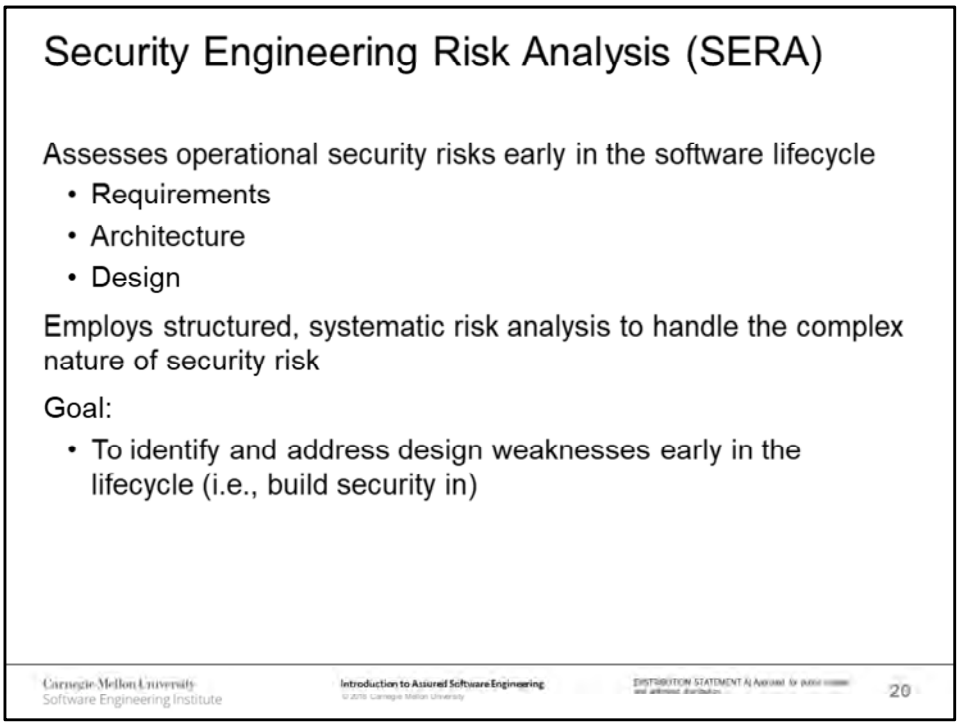


Operational security vulnerabilities generally have three main causes: (1) design weaknesses, (2) implementation/coding vulnerabilities, and (3) system configuration errors. This presentation is focused on design weaknesses. Early detection and remediation of design weaknesses will reduce residual security risk when a system is deployed. Addressing design weaknesses as soon as possible is especially important because these weaknesses are not corrected easily after a system has been deployed.

For example, software maintenance organizations cannot simply issue a patch to correct a design weakness. Remediation normally requires extensive redesign of the system, which is costly and often proves to be impractical. As a result, software-reliant systems with design weaknesses are often allowed to operate under a high degree of residual security risk, putting their associated operational missions in jeopardy.



Performing a risk analysis early in the life cycle does not guarantee that security risks will be handled effectively. Many traditional security risk-analysis methods cannot handle the inherent complexity of modern cybersecurity attacks. These methods are based on a simple, linear view of risk that assumes a single threat actor exploits a single vulnerability in a single system to cause an adverse consequence. In reality, multiple actors exploit multiple vulnerabilities in multiple systems as part of a complex chain of events. Traditional methods are often unable to analyze the complex cybersecurity attacks effectively.

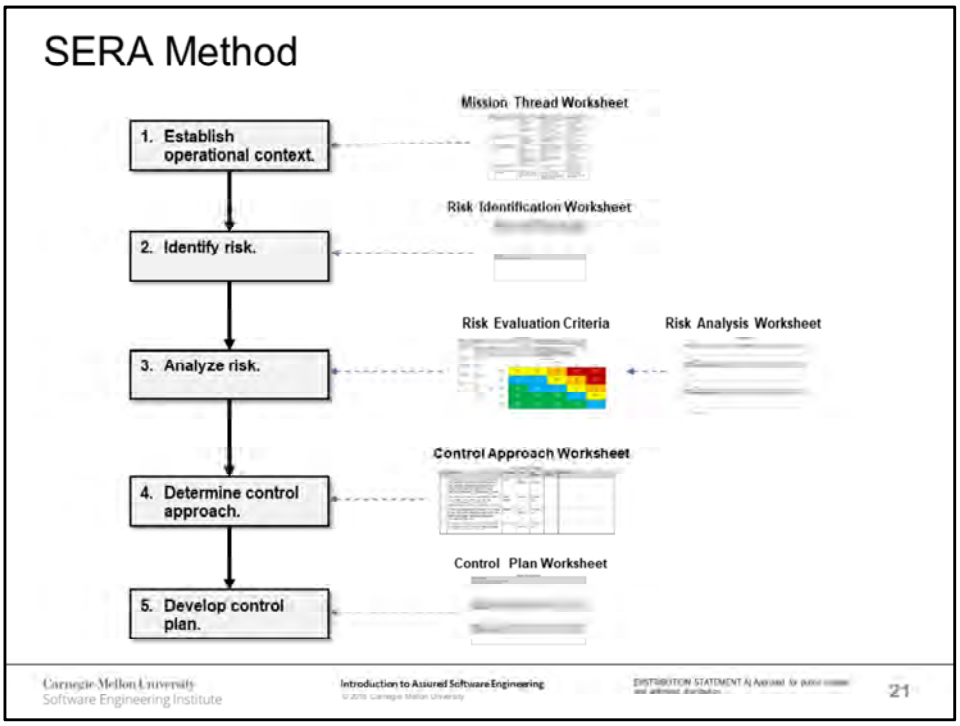


The Security Engineering Risk Analysis (SERA) method is designed for use during early life-cycle activities (e.g., during requirements, architecture, and design). The SERA method employs scenario-based risk analysis to handle the complex nature of cybersecurity risk. The goal is to identify design weaknesses early in the life cycle and enable corrective action to be taken. In this way, a subset of critical operational security risks can be mitigated long before a system is deployed.

The SERA method can be

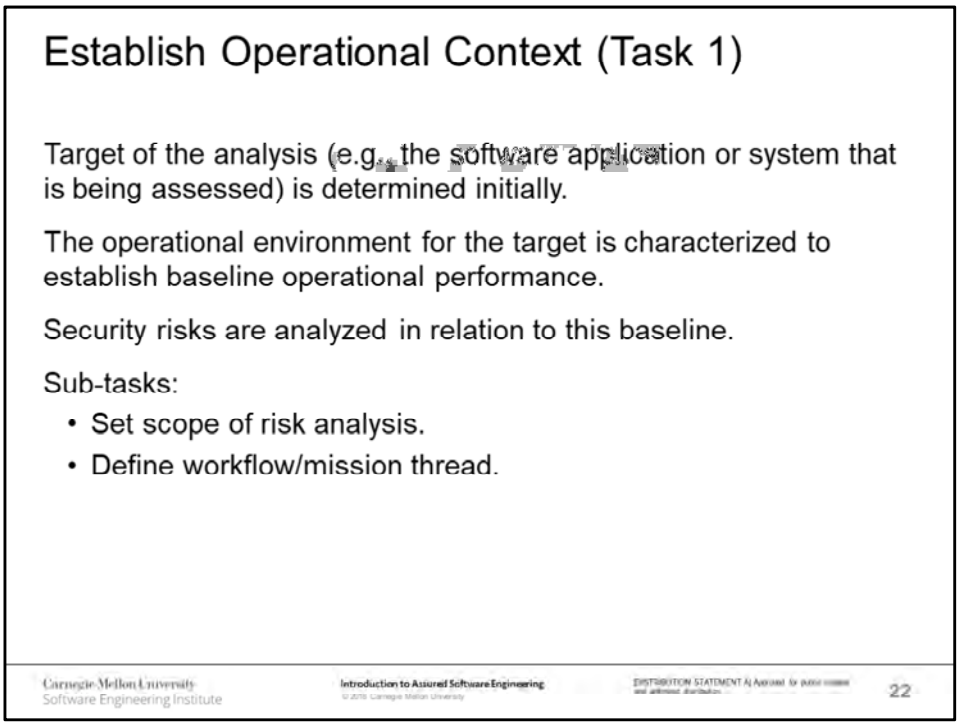
1. self-applied by the person or group that is responsible for acquiring and developing a software-reliant system
2. conducted by external parties on behalf of the responsible person or group.

This module highlights the key concepts of the SERA method without addressing specific implementation details.



The SERA method comprises the following five tasks:

1. *Establish operational context*—The target of the assessment (e.g., the software application or system that is being assessed) is determined initially. Next, the operational environment for the target of the assessment is characterized to establish baseline operational performance. Cybersecurity risks are analyzed in relation to this baseline.
2. *Identify risk*—Cybersecurity concerns are transformed into distinct, tangible risk scenarios that can be described and measured. The following elements are documented for each cybersecurity risk: risk statement, threat, consequence, and enablers.
3. *Analyze risk*—Each risk is evaluated in relation to predefined criteria to determine its probability, impact, and risk exposure.
4. *Determine control approach*—A strategy for controlling each risk is determined and documented based on predefined criteria and current constraints (e.g., resources and funding available for control activities). Control approaches for cybersecurity risks include accept, transfer, avoid, and mitigate.
5. *Develop control plan*—A control plan is defined and documented for all cybersecurity risks that are not accepted (i.e., risks that will be mitigated, transferred, or accepted). Risk mitigation plans typically include actions from the following categories: (1) monitor and respond, (2) protect, and (3) recover.

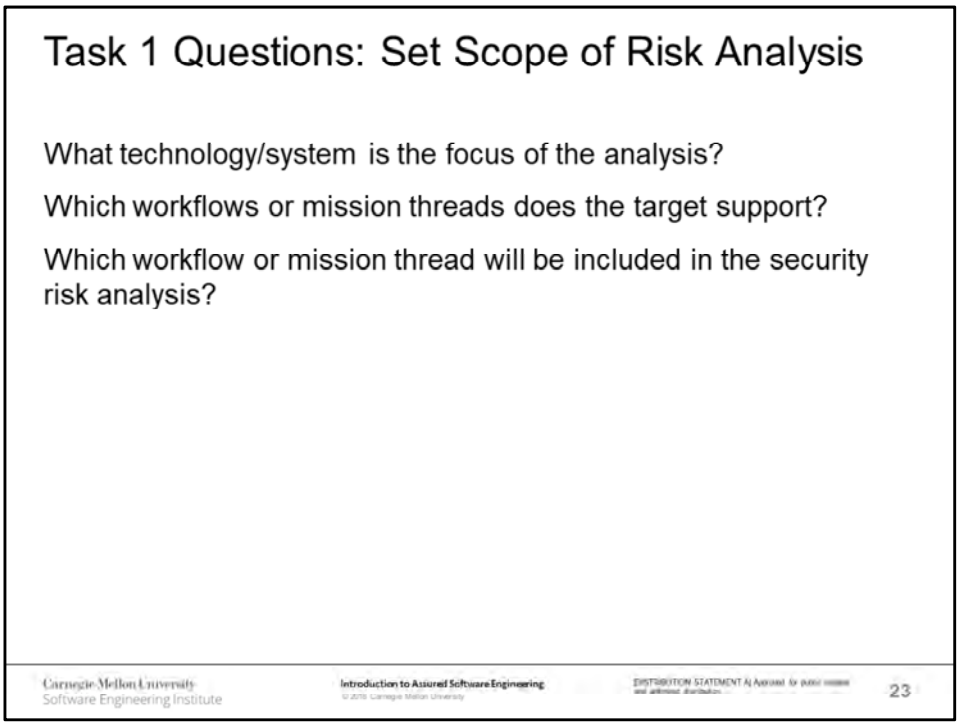


Task 1 comprises two sub-tasks:

- Set scope of risk analysis. (Sub-Task 1.1)
- Define workflow/mission thread. (Sub-Task 1.2)

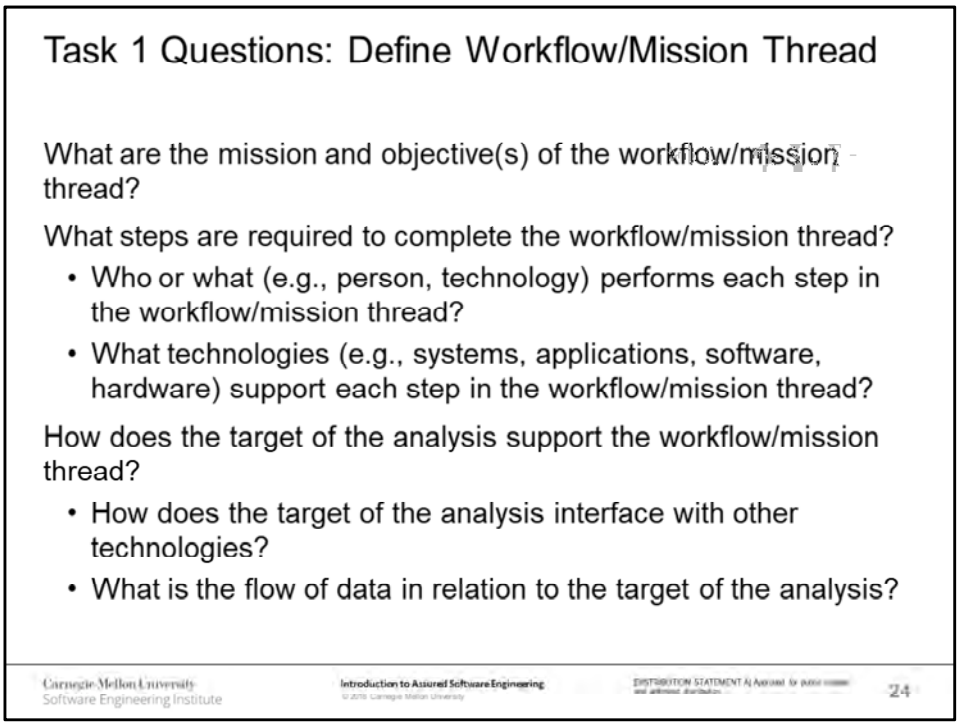
In Sub-Task 1.1, the target of the analysis is determined. The target is typically the software application or system that is the focus of the SERA method. This task helps to set the scope of the resulting risks analysis. In Sub-Task 1.2, the operational workflow (or mission thread) for the target is established (or is projected to support operations if the target is not yet deployed).

Each software application or system typically supports multiple operational workflows or mission threads during operations. The goal is to (1) select which operational workflow or mission thread will be included in the analysis and (2) document how the target of the analysis supports the selected workflow or mission thread. This establishes a baseline of operational performance for the target. The target is then analyzed cybersecurity risks in relation to this baseline.



This slide highlights the key questions answered when performing Sub-Task 1.1.

The goals of this sub-task are(1) to establish which systems is the focus of the security-risk analysis and (2) identify the workflow/mission thread that will provide the operational context for the analysis.



This slide highlights the key questions answered when performing Sub-Task 1.2.

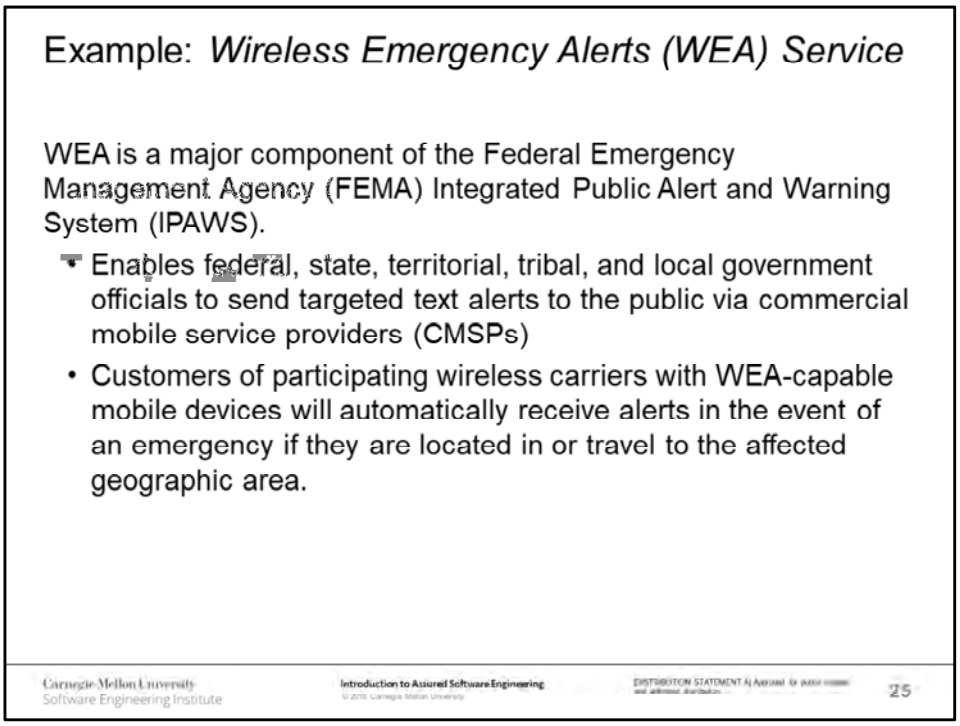
The goal of this sub-task is to document the steps that must be completed when performing the selected workflow/mission thread.

The mission and objective(s) of a workflow/mission thread are used to define the overarching "picture of success" for the workflow/mission thread. Here, *mission* is defined as the fundamental purpose of the workflow/mission thread that is being analyzed. An *objective* is defined as a tangible outcome or result that must be achieved when pursuing a mission.

At a minimum, the following performance parameters should be documented for the workflow/mission thread being analyzed:

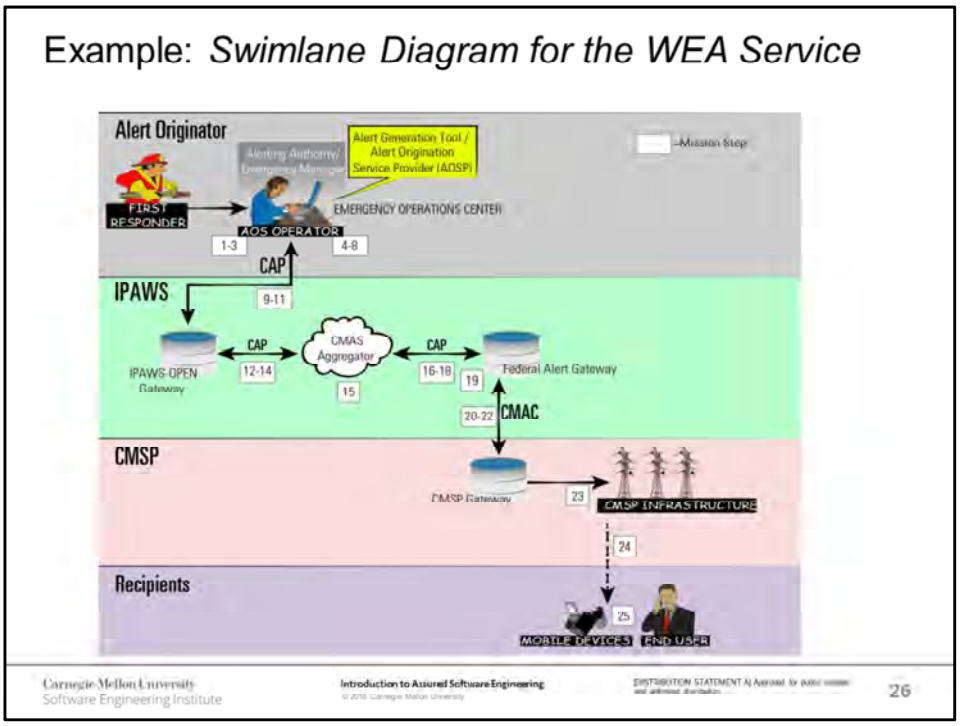
- The sequence and timing of all steps needed to achieve the mission and objective(s), including relevant interrelationships and dependencies among the activities
- Roles and responsibilities for completing each step
- Technologies (e.g., systems, applications, software, hardware) supporting each step

Two common ways for documenting a workflow/mission thread are diagramming techniques (e.g., process flow, swim-lane diagram) and spreadsheets. The workflow /mission thread should be documented using a format (e.g., diagram, spreadsheet) that is preferred by the stakeholders of the analysis.



The Wireless Emergency Alerts (WEA) service is a collaborative partnership that includes the cellular industry, Federal Communications Commission, Federal Emergency Management Agency (FEMA), and U.S. Department of Homeland Security (DHS) Science and Technology Directorate (S&T). The WEA capability disseminates emergency alerts to users of capable mobile devices in an affected geographic area. However, like other cyber-enabled services, WEA is subject to cyber threats that may prevent its use or damage the credibility of the service it provides. Attackers may attempt to delay, destroy, or modify alerts, or even to insert false alerts, actions that may pose a significant risk to the public.

WEA is a major component of the FEMA Integrated Public Alert and Warning System (IPAWS). It enables federal, state, territorial, tribal, and local government officials to send targeted text alerts to the public via commercial mobile service providers (CMSPs). Customers of participating wireless carriers with WEA-capable mobile devices will automatically receive alerts in the event of an emergency if they are located in or travel to the affected geographic area.



This slide shows a visual representation (i.e., a swimlane diagram) of the workflow/mission thread for the end-to-end WEA alerting pipeline.

As shown in the figure, the end-to-end WEA alerting pipeline is a complex system of systems comprising four major elements. The *alert originators* element consists of the people, information, technology, and facilities that initiate and create an alert. The *IPAWS-OPEN* element receives, validates, authenticates, and routes various types of alerts to the appropriate disseminator, such as WEA, the Emergency Alert System (EAS), or the National Oceanic and Atmospheric Administration. For WEA, IPAWS-OPEN and transmits WEA alerts to the commercial mobile service providers (CMSPs) element. The *CMSPs* element then broadcasts alerts to *alert recipients*, the WEA-capable mobile devices located in the targeted alert area.

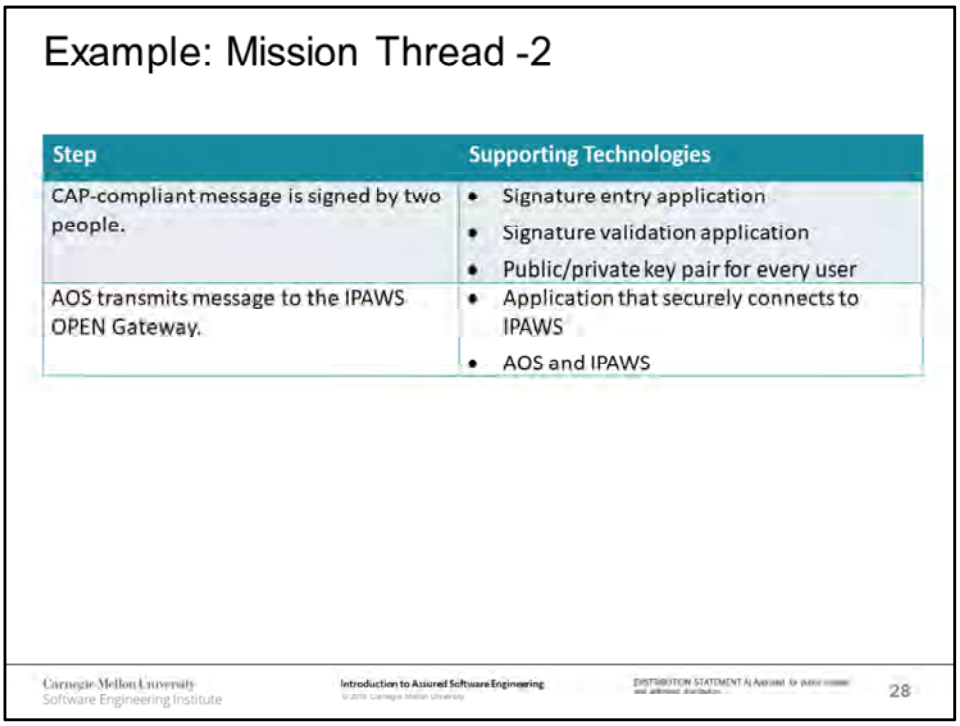
Note: A system of systems is defined as a set or arrangement of interdependent systems that are related or connected (i.e., networked) to provide a given capability.

Example: Mission Thread -1

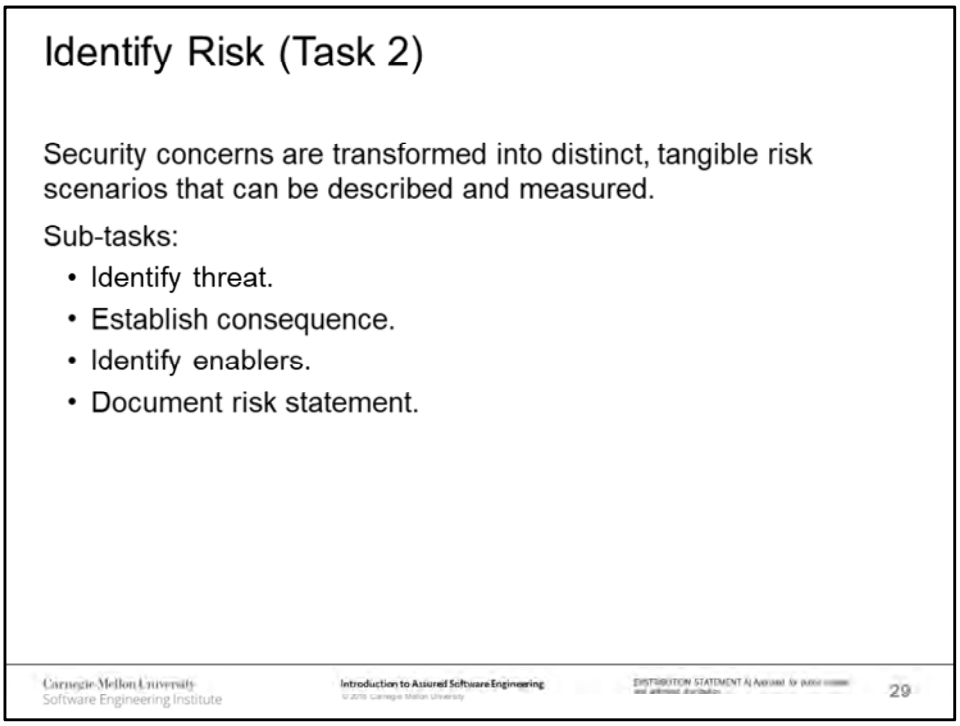
Step	Supporting Technologies
Alert Originating System (AOS) operator attempts to log on to the AOS.	• Server (valid accounts/authentication information) • Logon application • Communications between logon software/ server/AOS
AOS logon activates auditing of the operator's session.	• Auditing application • Communications from accounts to auditing application • Local/remote storage devices
AOS operator enters alert/cancel/update message with status of "actual."	• Alert scripts • Graphical user interface (GUI) application • Communications between GUI application and alert-generation software (including server and application)
AOS converts message to Common Alerting Protocol (CAP) compliant format.	• Conversion application

This slide shows a spreadsheet or table representation of the workflow/mission thread for the Alert Originator potion of the end-to-end WEA pipeline.

More specifically, this slide and the next highlight the main steps needed to enter an alert message into an Alert Originating System (AOS). In the steps highlighted on this slide, an operator enters an alert message into the AOS, and the AOS converts the message to the Common Alerting Protocol (CAP) compliant format.



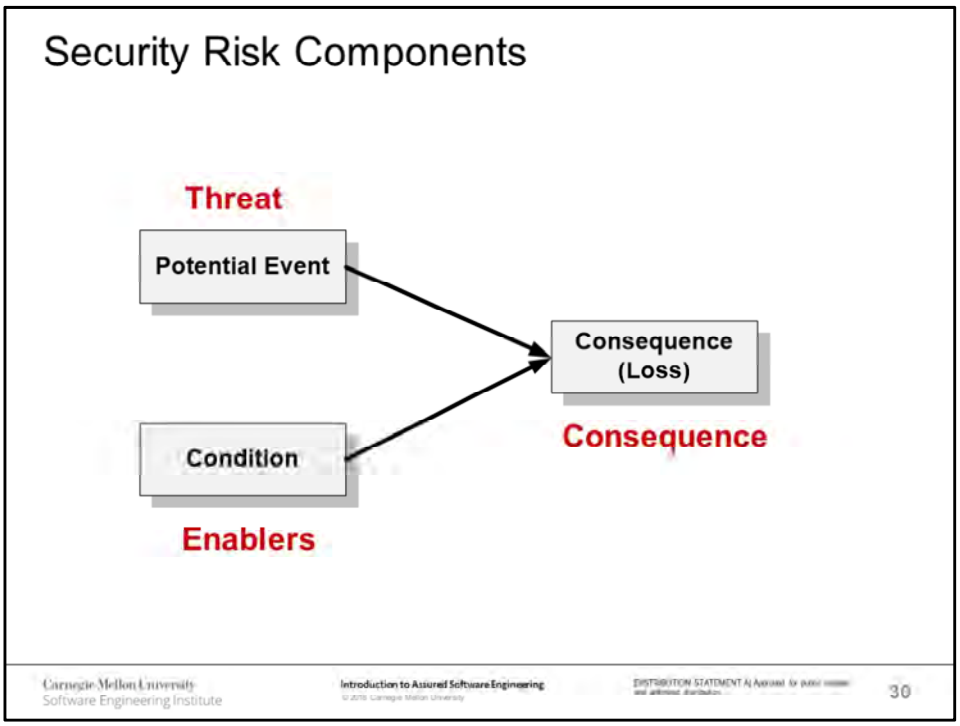
This slide continues the workflow/mission thread for the alert originator. The steps on this slide show how the CAP-compliant message is signed by two people and then transmitted to the next part of the WEA pipeline (i.e., the IPAWS OPEN Gateway).



Task 2 comprises four sub-tasks:

- Identify threat. (Sub-Task 2.1)
- Establish consequence. (Sub-Task 2.2)
- Identify enablers. (Sub-Task 2.3)
- Document risk statement. (Sub-Task 2.4)

The overarching goal of Task 2 is to transform security concerns into distinct, tangible risk scenarios that can be described and measured.



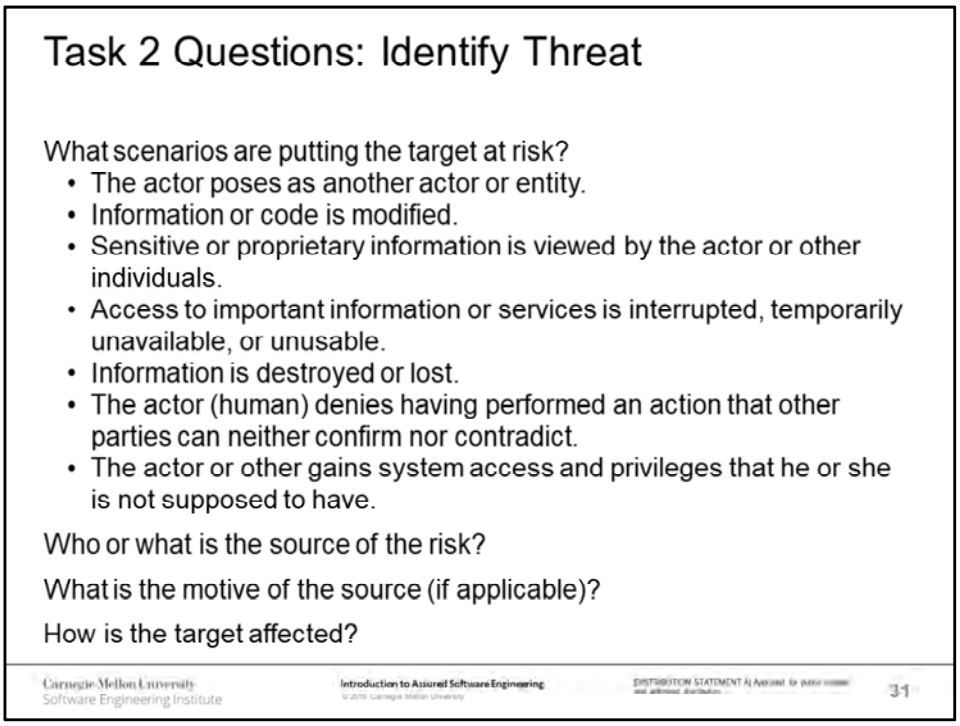
Cybersecurity risk is a measure of the likelihood that a threat will exploit one or more vulnerabilities to produce an adverse consequence, or loss, coupled with the magnitude of the loss. The figure on the slide illustrates the three core components of cybersecurity risk:

- *Threat* – a cyber-based act, occurrence, or event that exploits one or more vulnerabilities and leads to an adverse consequence or loss
- *Vulnerability* – a weakness in an information system, system security procedures, internal controls, or implementation that a threat could exploit to produce an adverse consequence or loss; a current condition that leads to or enables cybersecurity risk
- *Consequence* – the loss that results when a threat exploits one or more vulnerabilities; the loss is measured in relation to the status quo (i.e., current state)

The basic elements of risk were first introduced on slide 6. This slide takes those basic elements and maps them to the cybersecurity context.

From the cybersecurity perspective, a vulnerability is the passive element of risk. It exposes cyber technologies (e.g., software application, software-reliant system) to

threats and the losses that those threats can produce. However, by itself, a vulnerability will not cause an entity to suffer a loss or experience an adverse consequence; rather, the vulnerability makes the entity susceptible to the effects of a threat .



This slide highlights the key questions answered when performing Sub-Task 2.1.

The goal of this sub-task is to identify threats that are causing concern.

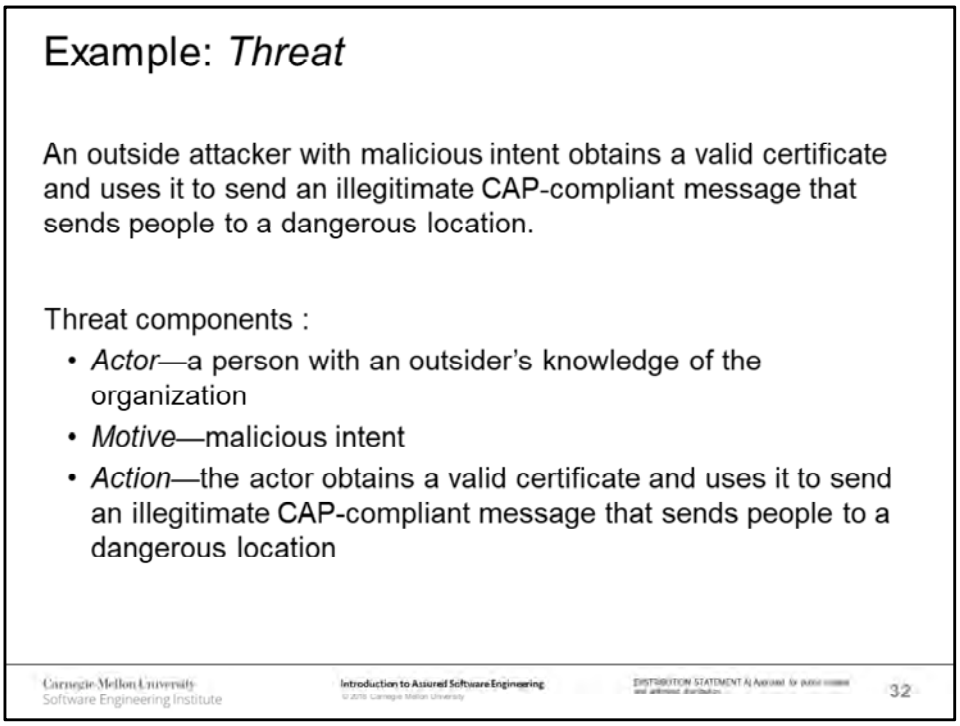
The first component of threat, actor, is the source of the threat. It describes who or what causes the threat. Examples of typical actors for cybersecurity threats include:

- *Outsider* – a person with an outsider's knowledge of the organization
- *Insider* – a person with an insider's knowledge of the organization
- *Malicious code* – code that is intended to cause undesired effects, security breaches, or damage to a system (e.g., scripts, viruses, worms, Trojan horses, backdoors, and malicious active content).

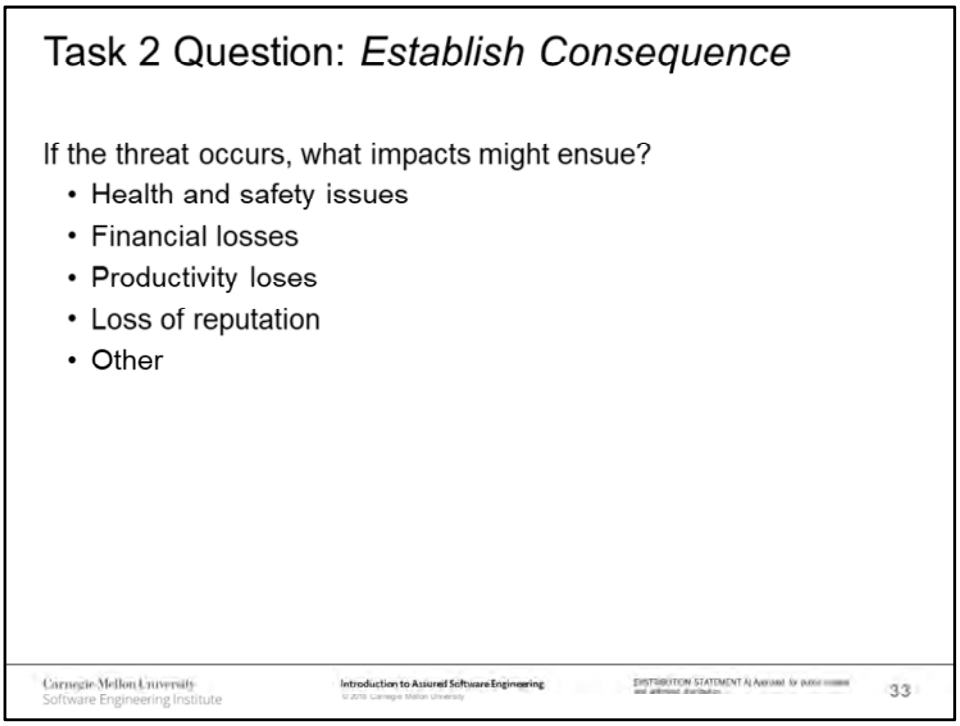
Motive is the second component of a threat. It defines the reason why the actor attempts to carry out the threat. Examples of motive include:

- *Intentional or malicious* – a person intentionally tries to cause the action
- *Accidental* – a person inadvertently causes the action to occur

In general, motive applies only to human actors. The final component of threat, action, describes what the actor does to place the target at risk.



This slide illustrates a threat to the AOS.

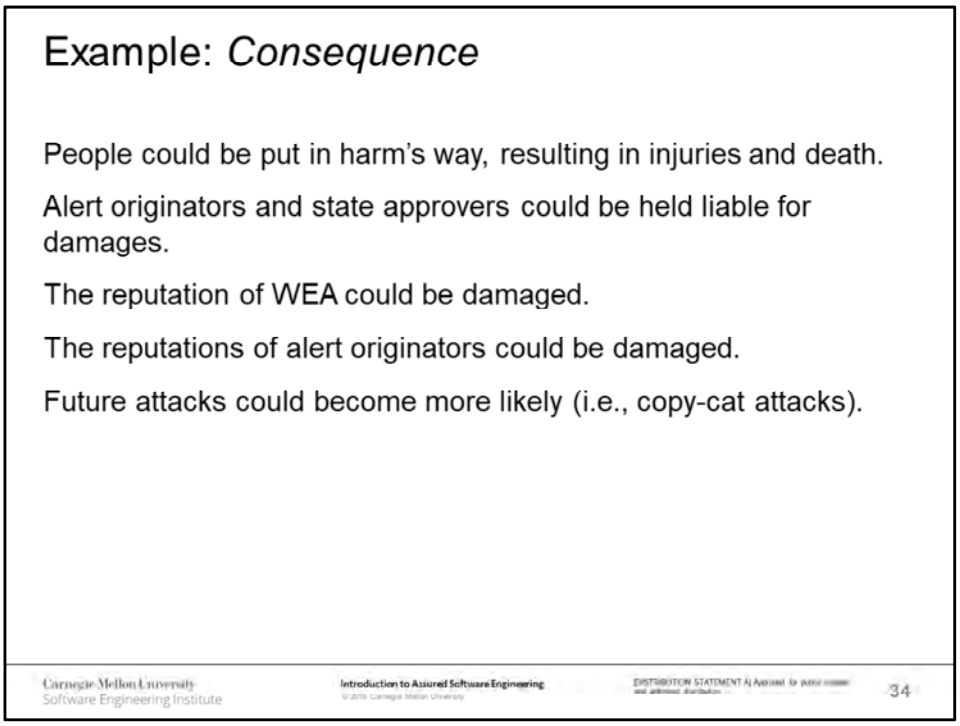


This slide highlights the key questions answered when performing Sub-Task 2.2.

The goal of this sub-task is to establish the consequences that could be produced by the threat.

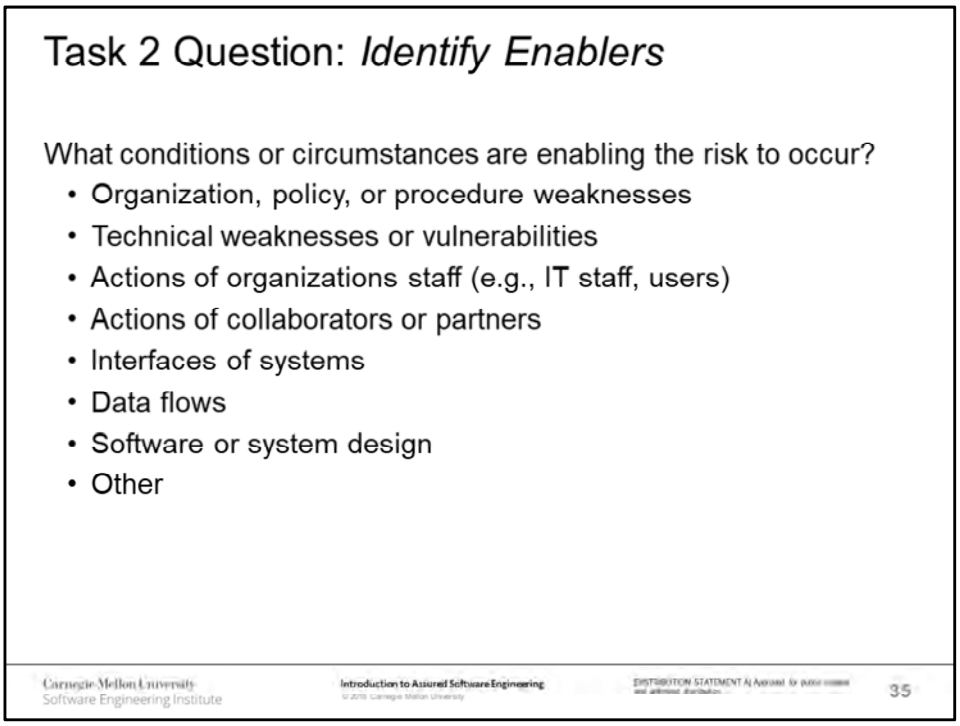
To focus the identification of consequences, it is often advisable to think about specific types of consequences, such as

- health and safety issues
- financial losses
- productivity losses
- loss of reputation
- other



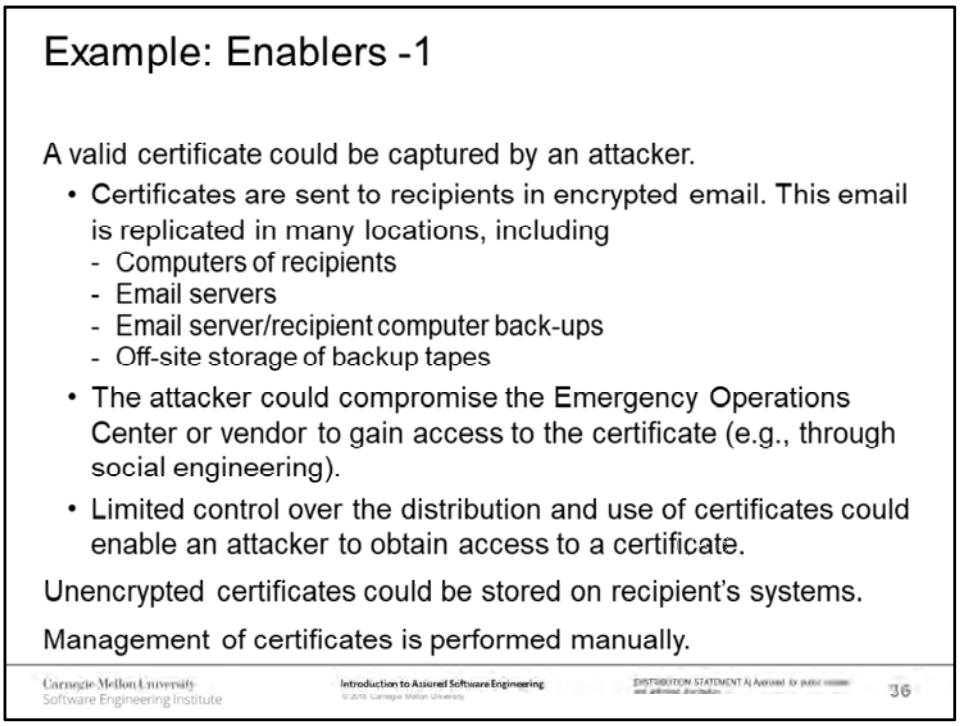
This slide illustrates the range of consequences for the threat shown on slide 35.

Consequences are generated by (1) examining the threat's impact on the workflow and (2) analyzing the threat's impact on key stakeholders.

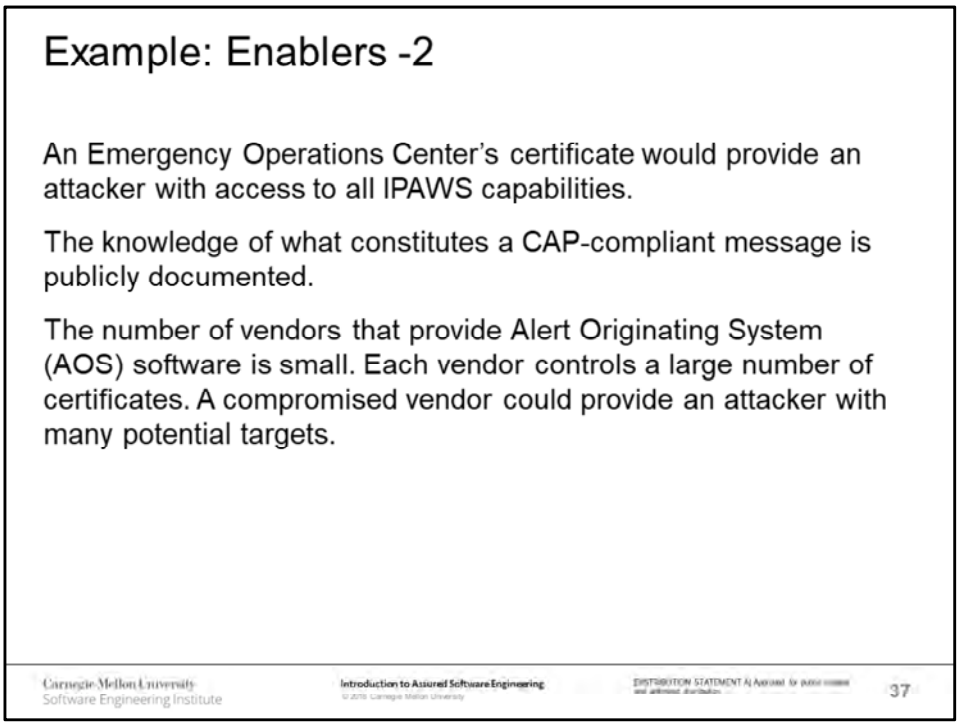


This slide highlights the key questions answered when performing Sub-Task 2.3.

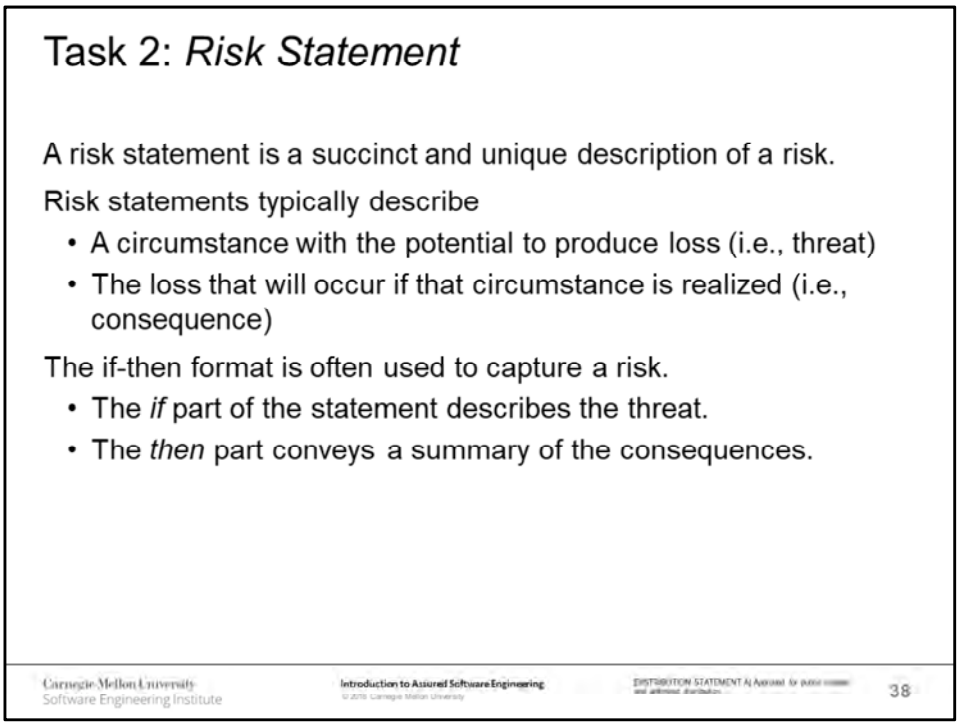
The goal of this sub-task is to determine the conditions or circumstances that will allow the risk (threat and consequence) to occur. These conditions or circumstances are referred to collectively as *enablers* and can include vulnerabilities, occurrence of related risks, actions that people might take, and dependencies on related technologies and data.



This slide illustrates the conditions and circumstances that enable the threat (slide 35) and consequences (slide 37) to occur.

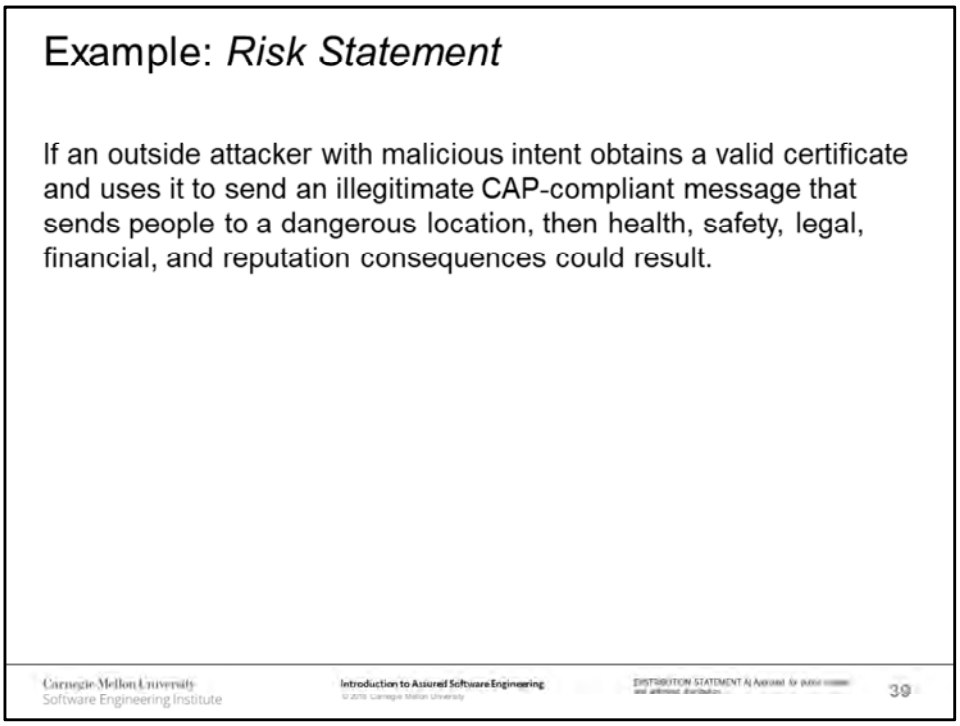


This slide continues the list of enablers from slide 39.

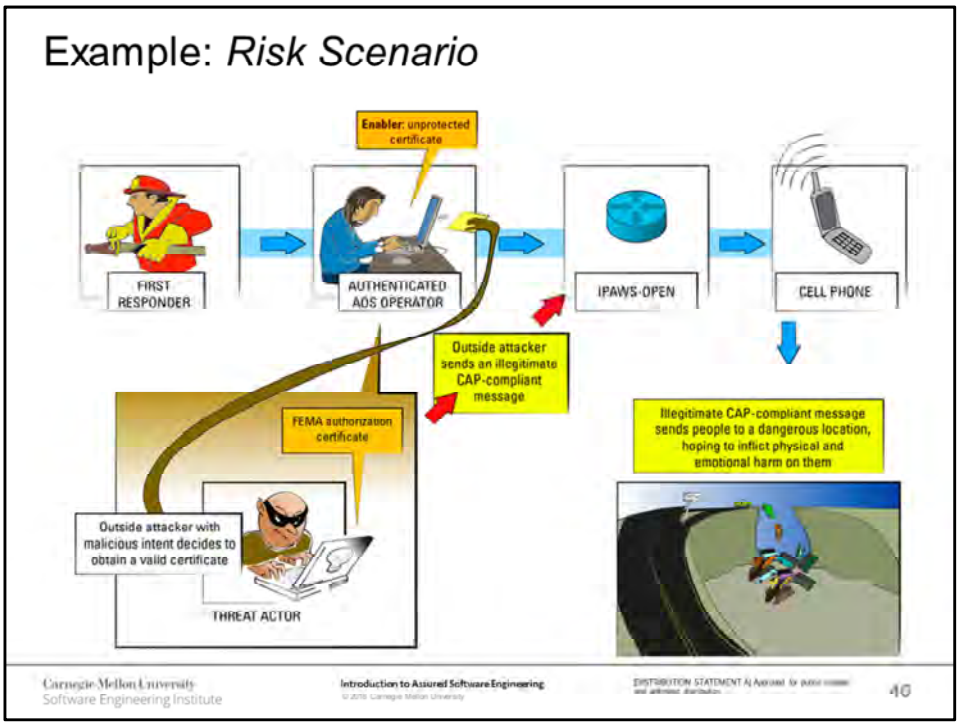


This slide highlights the key questions answered when performing Sub-Task 2.4.

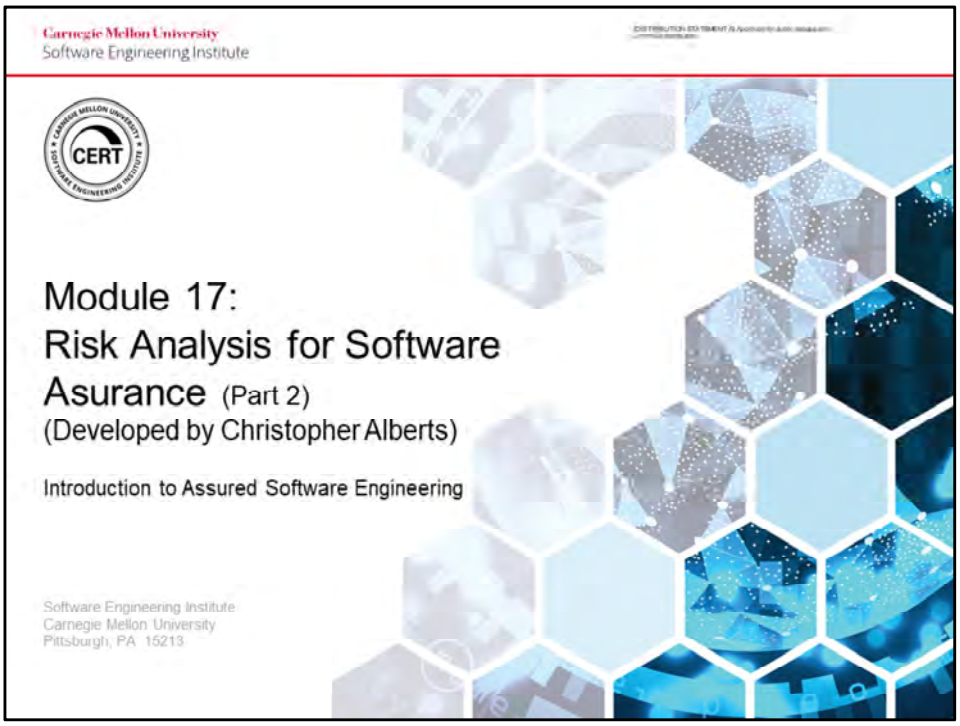
The goal of this sub-task is to document a risk statement for each risk that is identified. A risk statement is a succinct and specific description of a risk. Risk statements typically describe (1) a circumstance with the potential to produce loss (i.e., threat) and (2) the loss that will occur if that circumstance is realized (i.e., consequence). The if-then format is often used to capture a risk. The *if* part of the statement describes the threat, while the *then* part summarizes the consequences.

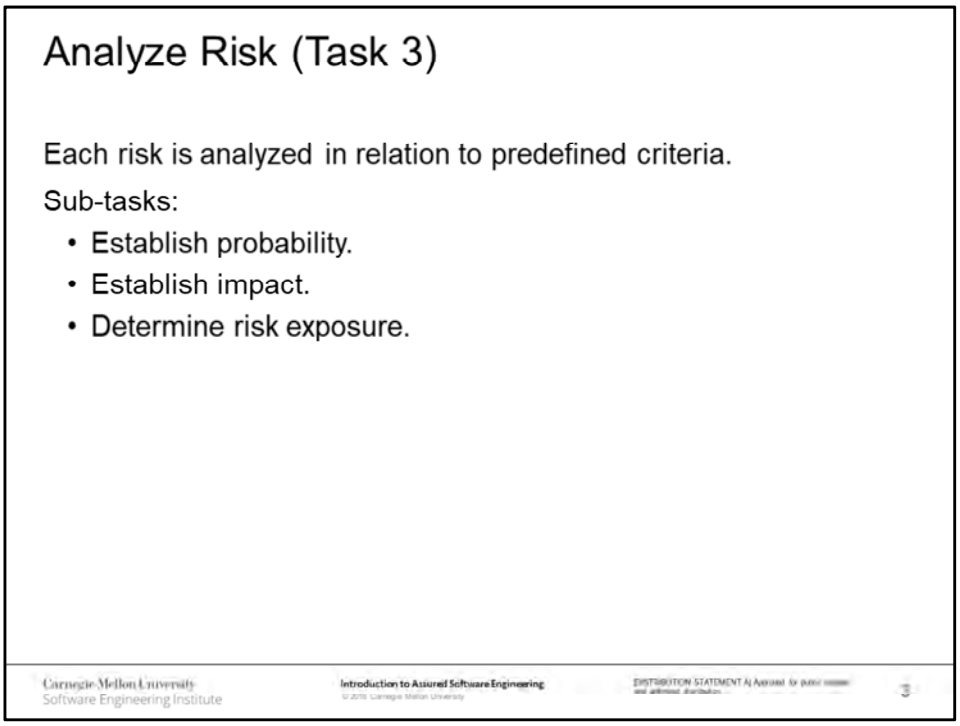


This slide highlights the risk statement for the threat from slide 35 and the consequences from slide 37.



This slide illustrates the risk in a scenario format. Here, the three elements of the risk (threat, enablers, consequences) are featured. A scenario-based expression of a risk can be useful for communicating the risk to certain audiences.

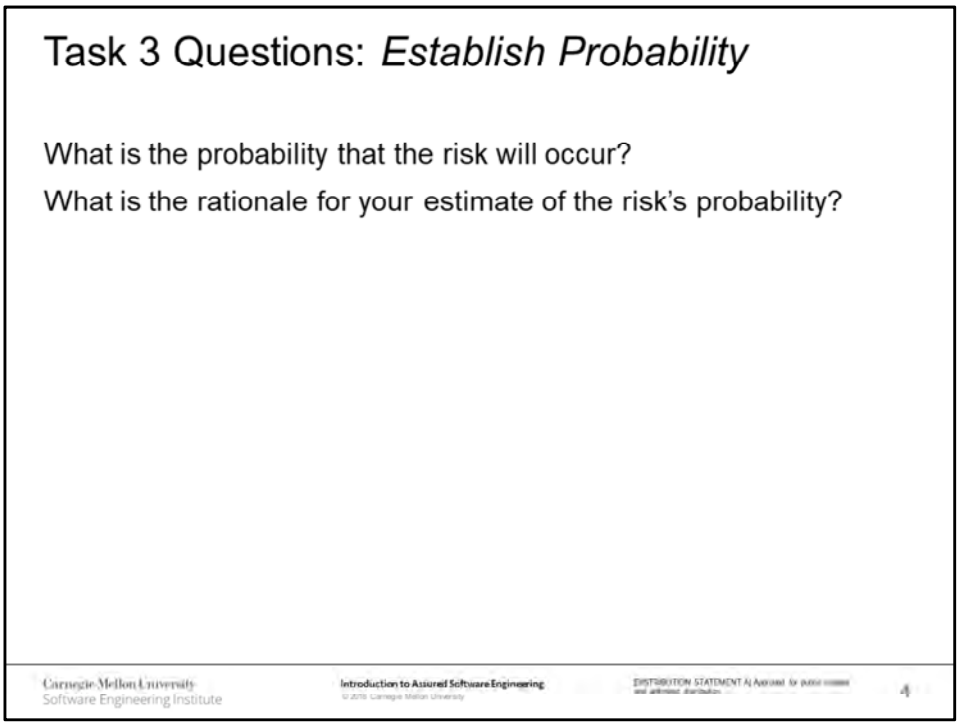

Carnegie Mellon University
Software Engineering Institute
CERT
Module 17:
Risk Analysis for Software Asurance (Part 2)
(Developed by Christopher Alberts)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Task 3 comprises three sub-tasks:

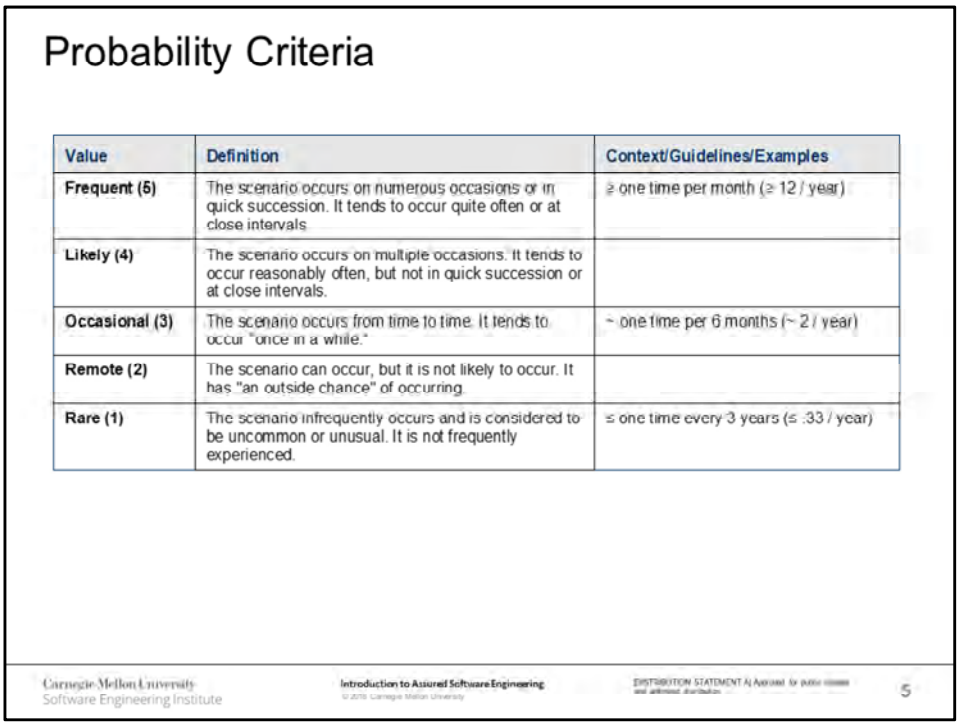
- Establish probability. (Sub-Task 3.1)
- Establish impact. (Sub-Task 3.2)
- Determine risk exposure. (Sub-Task 3.3)

The overarching goal of Task 3 is to analyze each risk identified during Task 2 in relation to predefined criteria.



This slide highlights the key questions answered when performing Sub-Task 3.1.

The goal of this sub-task is to establish the probability for each risk that is identified. Probability provides a measure of the likelihood that a risk will occur.



Probability Criteria

Value	Definition	Context/Guidelines/Examples
Frequent (5)	The scenario occurs on numerous occasions or in quick succession. It tends to occur quite often or at close intervals.	≥ one time per month (≥ 12 / year)
Likely (4)	The scenario occurs on multiple occasions. It tends to occur reasonably often, but not in quick succession or at close intervals.	
Occasional (3)	The scenario occurs from time to time. It tends to occur "once in a while."	~ one time per 6 months (~ 2 / year)
Remote (2)	The scenario can occur, but it is not likely to occur. It has "an outside chance" of occurring.	
Rare (1)	The scenario infrequently occurs and is considered to be uncommon or unusual. It is not frequently experienced.	≤ one time every 3 years (≤ .33 / year)

Carnegie Mellon University
Software Engineering Institute

Introduction to Assured Software Engineering

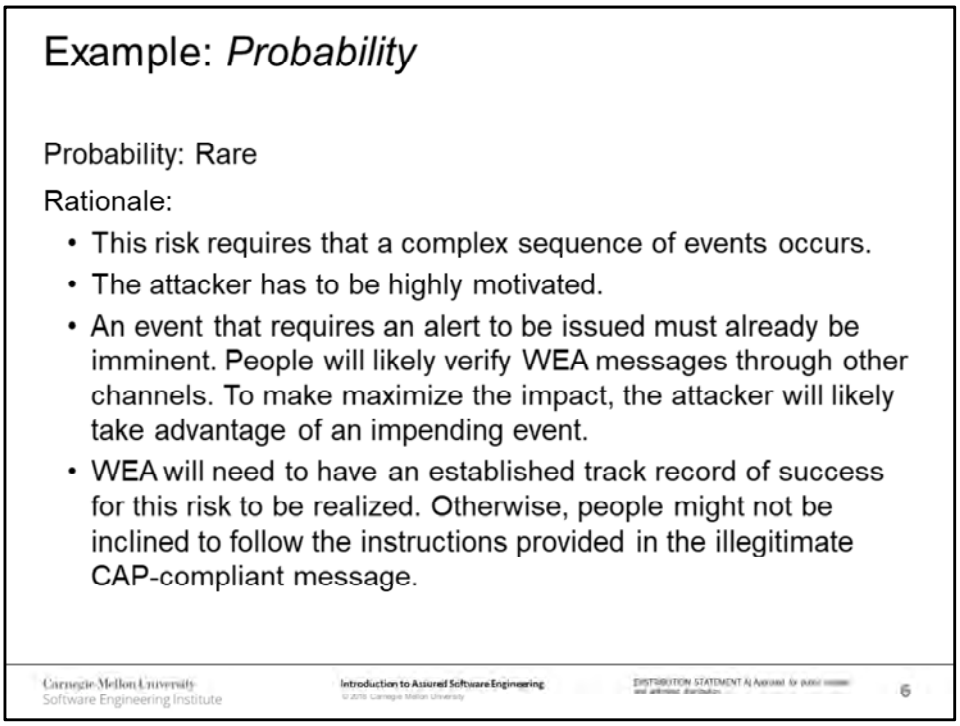
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

5

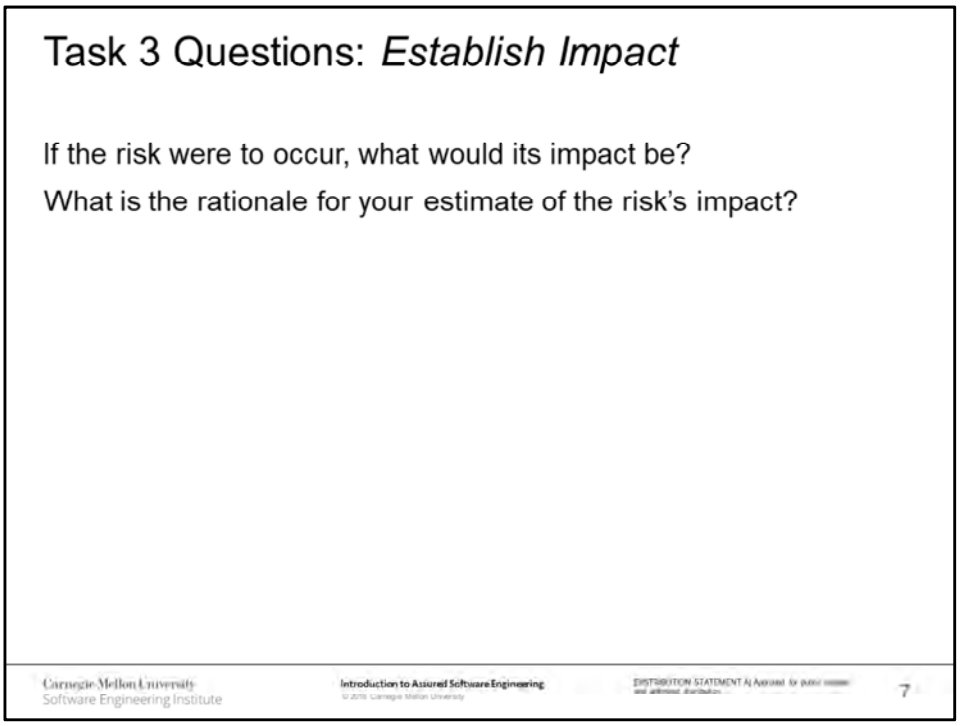
Probability is evaluated in relation to a set of predefined criteria. These criteria provide definitions for values of probability. Qualitative risk assessments can define different levels of probability values, such as

- Three levels – likely, occasional, remote
- Five levels – frequent, likely, occasional, remote, rare

This slide shows criteria for a five levels of probability. Criteria for evaluating probability must be tailored to the operational context in which risk is being evaluated.

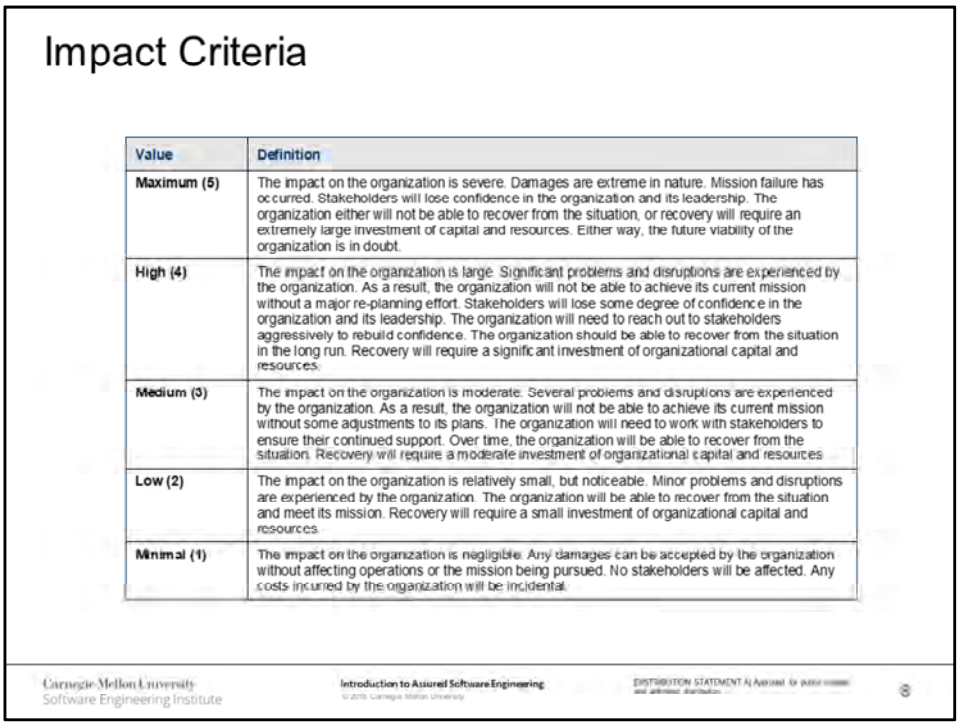


This slide shows the probability value for the risk being analyzed. The rationale for selecting the probability value is also documented.



This slide highlights the key questions answered when performing Sub-Task 3.2.

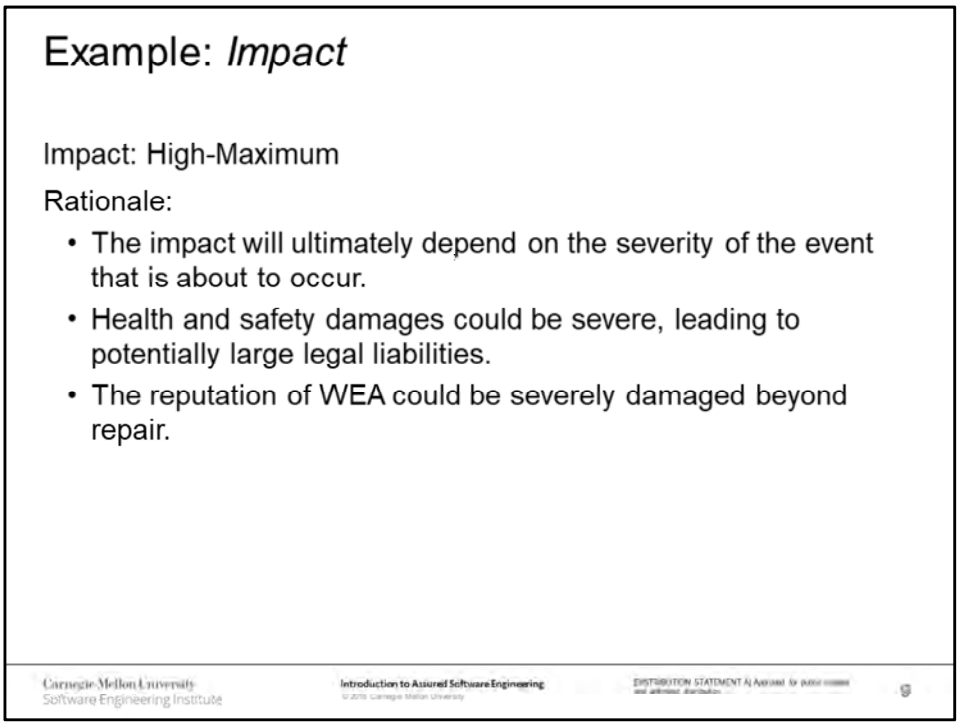
The goal of this sub-task is to establish the impact of each risk that is identified. Impact provides a measure of the severity of a risk's consequence if the risk were to occur.



Impact is evaluated in relation to a set of predefined criteria. These criteria provide definitions for values of impact. Qualitative risk assessments can define different levels of impact values, such as

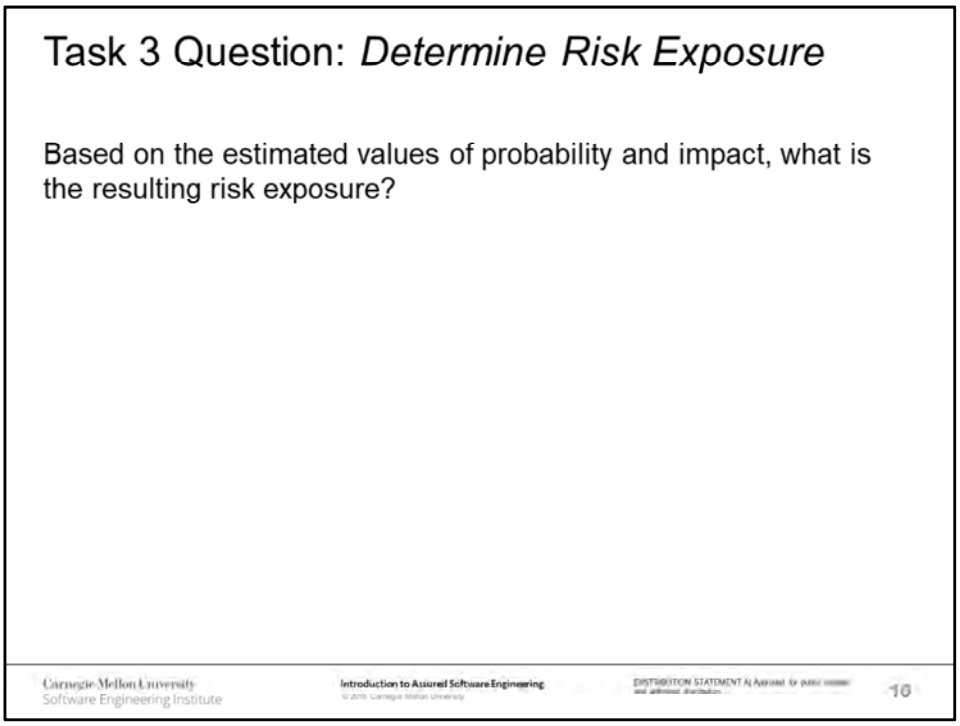
- Three levels – high, medium, low
- Five levels – maximum, high, medium, low, minimum

This slide shows criteria for a five levels of impact. Criteria for evaluating impact must be tailored to the operational context in which risk is being evaluated.



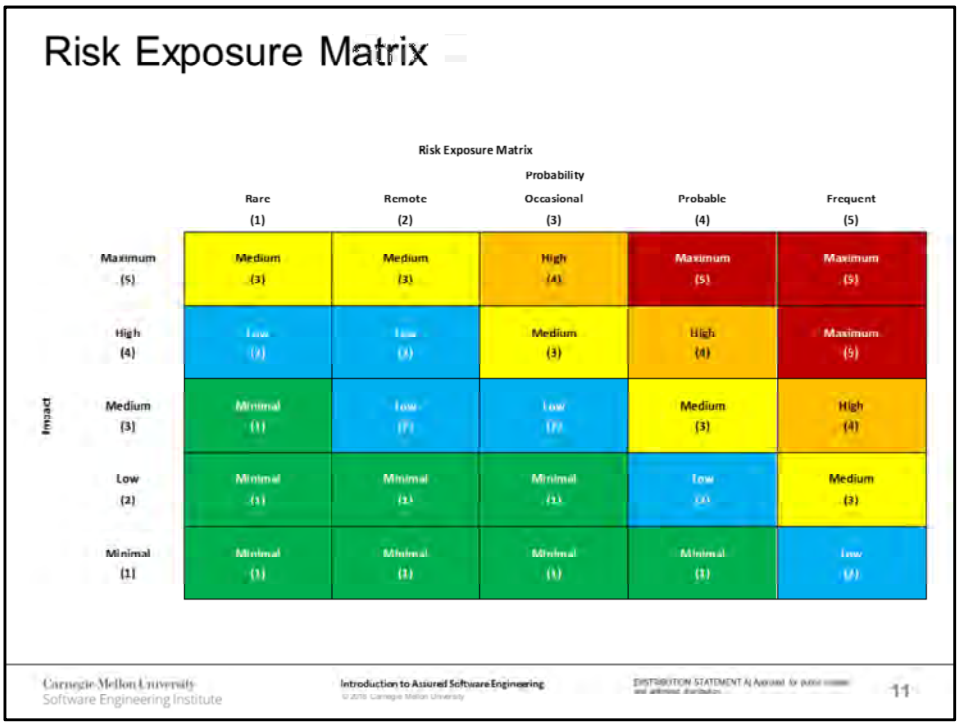
This slide shows the impact value for the risk being analyzed. (See slide 38 from Module 16 for the risk statement.) The rationale for selecting the impact value is also documented.

For the risk being analyzed, note that the impact value was judged to be between high and maximum based on the criteria.



This slide highlights the key question answered when performing Sub-Task 3.3.

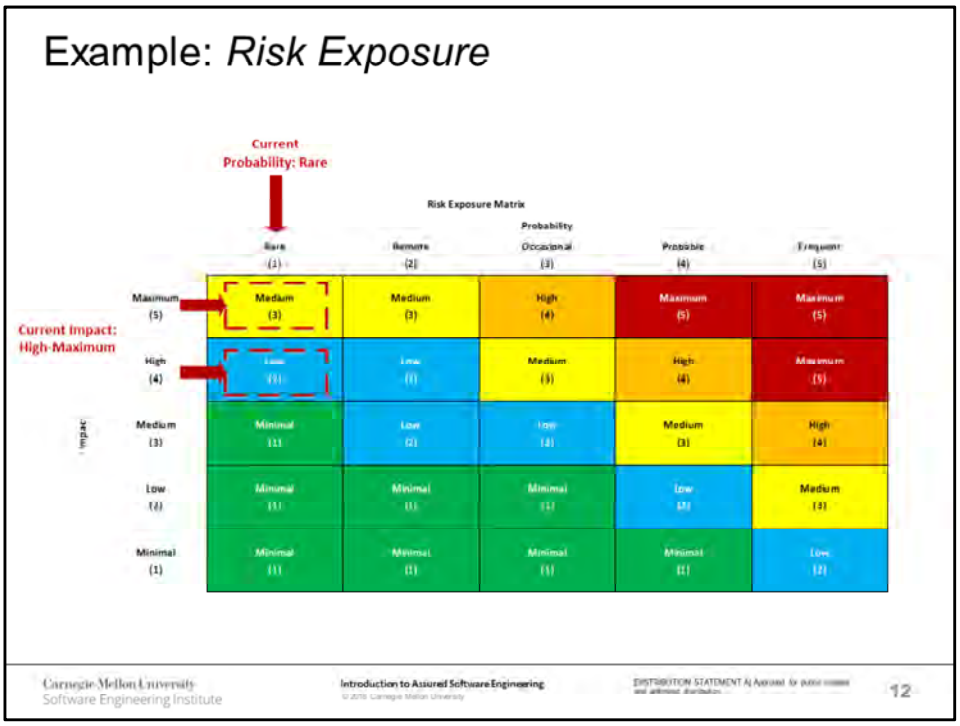
The goal of this sub-task is to establish the risk exposure for each risk that is identified. Risk exposure provides a measure of the magnitude of a risk based on its values of probability and impact.



Risk exposure is evaluated in relation to a set of predefined criteria. A matrix is used to derive risk exposure from the individual values of probability and impact.. Qualitative risk assessments can define different levels of risk exposure, such as

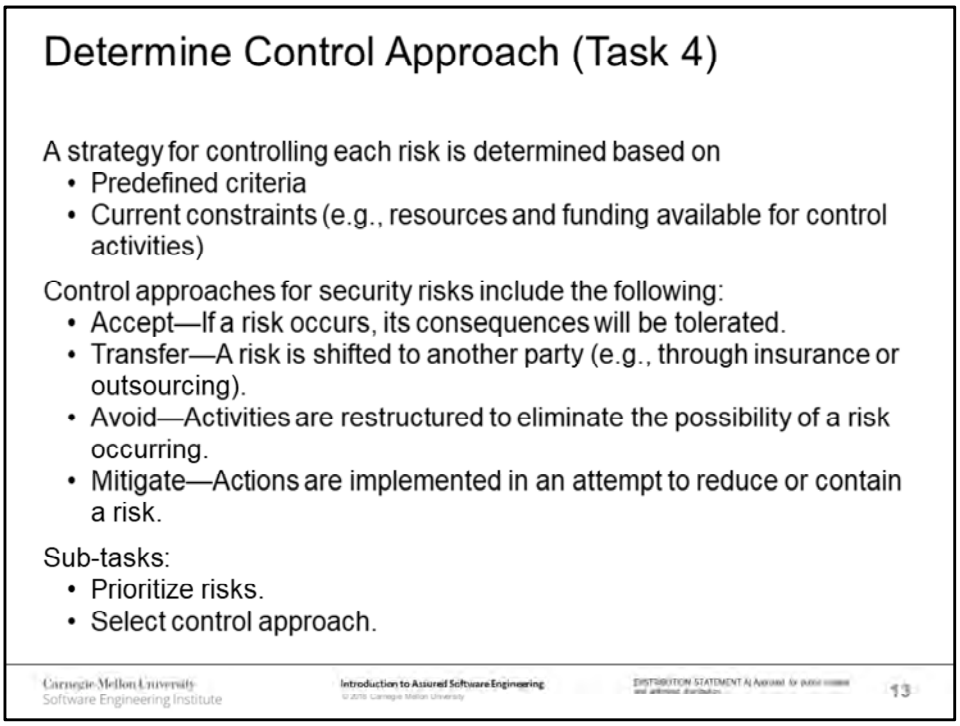
- Three levels – high, medium, low
- Five levels – maximum, high, medium, low, minimum

This slide shows a matrix for a five levels of risk exposure. Criteria for deriving risk exposure must be tailored to the operational context in which risk is being evaluated.



Risk exposure is determined using the current values of probability and impact. For the selected risk, the probability was determined to be rare and the impact was determined to be between high and maximum.

As shown in the figure, risk exposure is the intersection between the probability and impact values. As a result, the risk exposure for this example is low-medium.

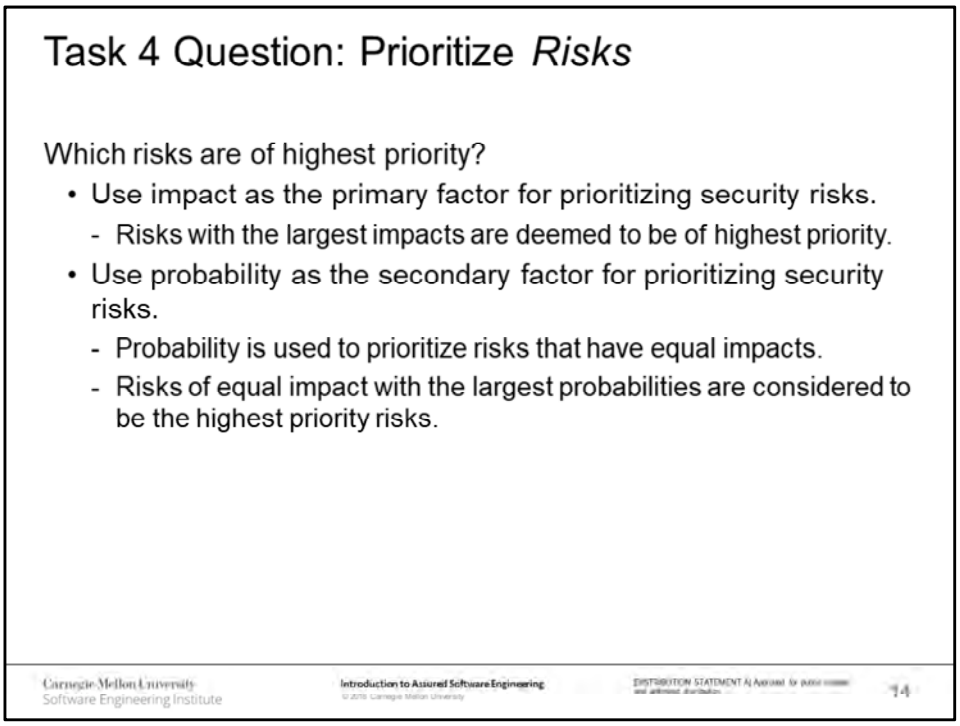


Task 4 comprises two sub-tasks:

- Prioritize risks. (Sub-Task 4.1)
- Select control approach. (Sub-Task 4.2)

The overarching goal of Task 4 is to decide how to address each risk. The strategy for controlling a risk is based on the measures for the risk (i.e., probability, impact, and risk exposure), which are established during the risk assessment. Decision-making criteria (e.g., for prioritizing risks or deciding when to escalate risks within an organization) may also be used to help determine the appropriate strategy for controlling a risk. Common control approaches include:

- *Accept* – If a risk occurs, its consequences will be tolerated; no proactive action to address the risk will be taken. When a risk is accepted, the rationale for doing so is documented.
- *Transfer* – A risk is shifted to another party (e.g., through insurance or outsourcing).
- *Avoid* – Activities are restructured to eliminate the possibility of a risk occurring.
- *Mitigate* – Actions are implemented in an attempt to reduce or contain a risk.



This slide highlights the key question answered when performing Sub-Task 4.1.

The goal of this sub-task is to prioritize risks based on their values of impact, probability, and risk exposure.

The following guidelines can be used when prioritizing a list of risks:

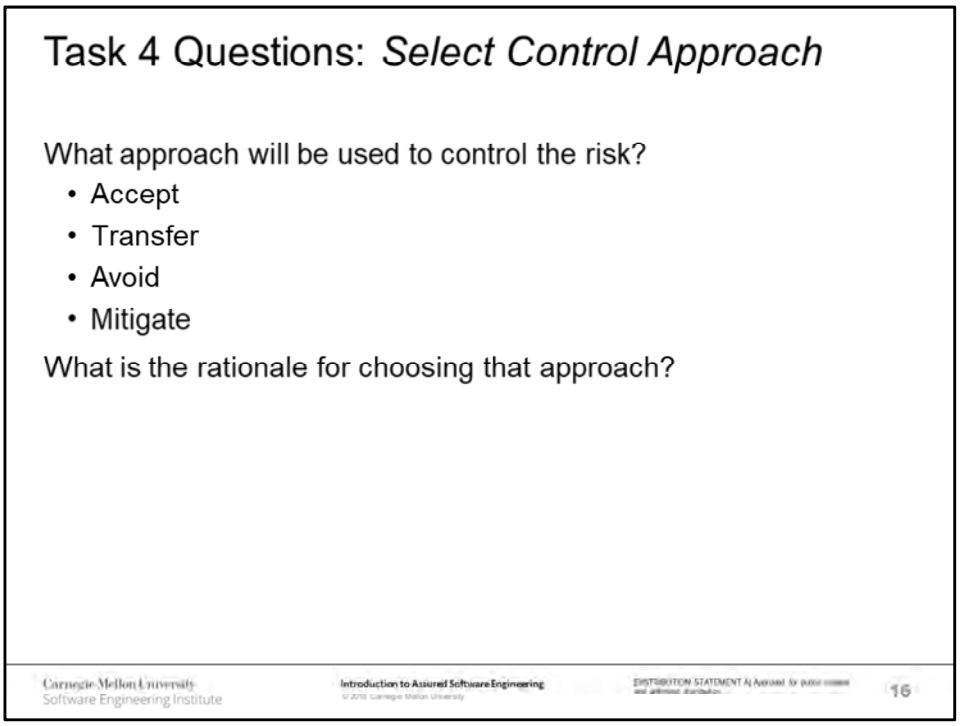
- Use impact as the primary factor for prioritizing cybersecurity risks. Risks with the largest impacts are deemed to be of highest priority.
- Use probability as the secondary factor for prioritizing cybersecurity risks. Probability is used to prioritize risks that have equal impacts. Risks of equal impact with the largest probabilities are considered to be the highest priority risks.

The prioritization guidelines should be tailored to the decision-making needs of key stakeholders.

Example: *Prioritized Risk Spreadsheet*

ID	Risk Statement	Impact	Prob	Risk Exp
1	If an outside attacker with malicious intent obtains a valid certificate and uses it to send an illegitimate CAP-compliant message that directs people to a dangerous location, then health, safety, legal, financial, and reputation consequences could result.	High-Max	Rare	Low-Med
3	If an insider with malicious intent spoofs the identity of a colleague and sends an illegitimate CAP-compliant message, then individual and organizational reputation consequences could result.	Med	Rare-Remote	Min-Low
2	If malicious code prevents an operator from entering an alert into the Alert Originating System (AOS), then health, safety, legal, financial, and productivity consequences could result.	Low-Med	Remote	Min-Low
4	If the internet communication channel for the AOS is unavailable due to a cybersecurity attack on the internet service provider, then health and safety consequences could result.	Low-Med	Remote	Min-Low

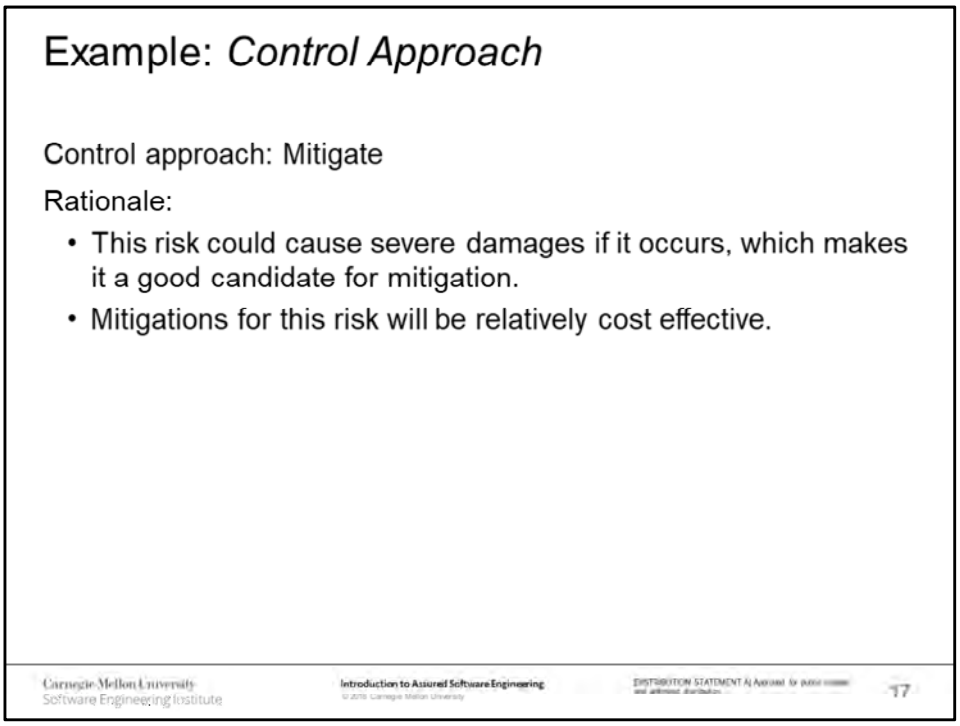
This slide shows a spreadsheet that documents the results of risk prioritization. The risk that has been analyzed in this presentation is Risk 1 from the spreadsheet. The spreadsheet show how Risk1 ranks in relation to other risks that were identified.



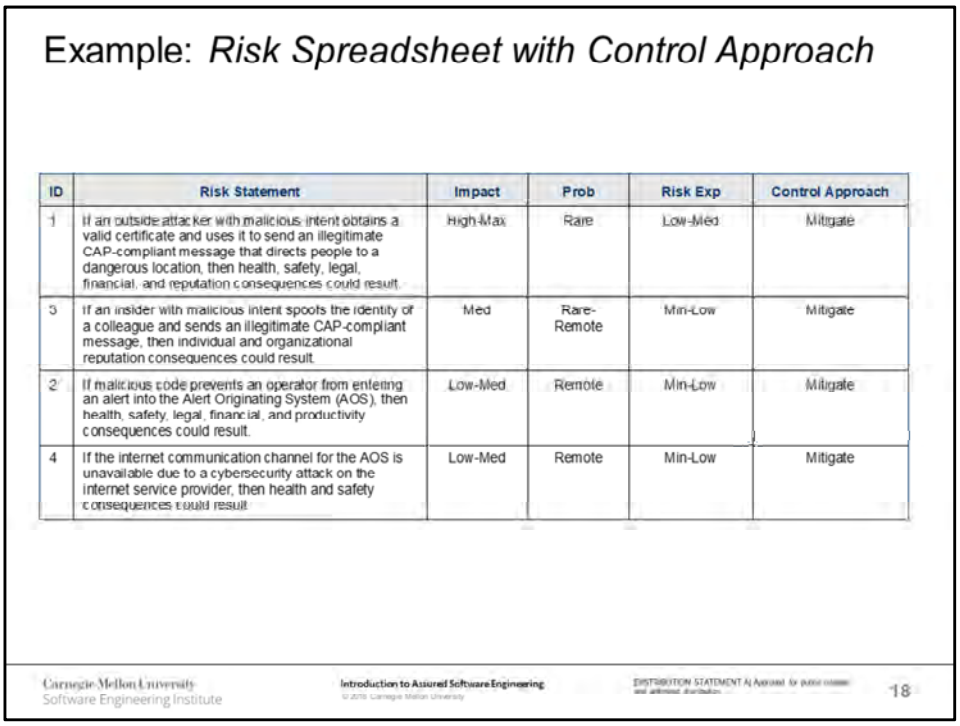
This slide highlights the key questions answered when performing Sub-Task 4.2.

The goal of this sub-task is to select a control approach for each risk that has been identified. Common control approaches include:

- *Accept* – If a risk occurs, its consequences will be tolerated; no proactive action to address the risk will be taken. When a risk is accepted, the rationale for doing so is documented.
- *Transfer* – A risk is shifted to another party (e.g., through insurance or outsourcing).
- *Avoid* – Activities are restructured to eliminate the possibility of a risk occurring.
- *Mitigate* – Actions are implemented in an attempt to reduce or contain a risk.



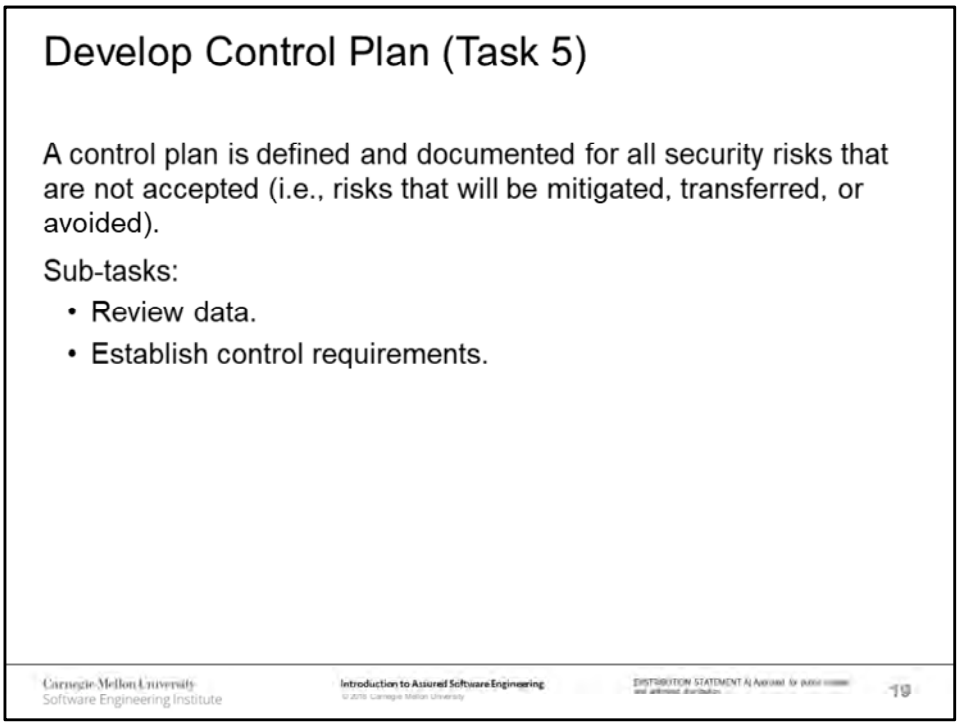
This slide shows the control approach for the risk that has been analyzed in this presentation. The rationale for selecting the control approach is also documented.



Example: *Risk Spreadsheet with Control Approach*

ID	Risk Statement	Impact	Prob	Risk Exp	Control Approach
1	If an outside attacker with malicious intent obtains a valid certificate and uses it to send an illegitimate CAP-compliant message that directs people to a dangerous location, then health, safety, legal, financial, and reputation consequences could result.	High-Max	Rare	Low-Med	Mitigate
3	If an insider with malicious intent spoofs the identity of a colleague and sends an illegitimate CAP-compliant message, then individual and organizational reputation consequences could result.	Med	Rare-Remote	Min-Low	Mitigate
2	If malicious code prevents an operator from entering an alert into the Alert Originating System (AOS), then health, safety, legal, financial, and productivity consequences could result.	Low-Med	Remote	Min-Low	Mitigate
4	If the internet communication channel for the AOS is unavailable due to a cybersecurity attack on the internet service provider, then health and safety consequences could result.	Low-Med	Remote	Min-Low	Mitigate

This slide shows the risk spreadsheet from slide 15 with the control approach that was selected for each risk.

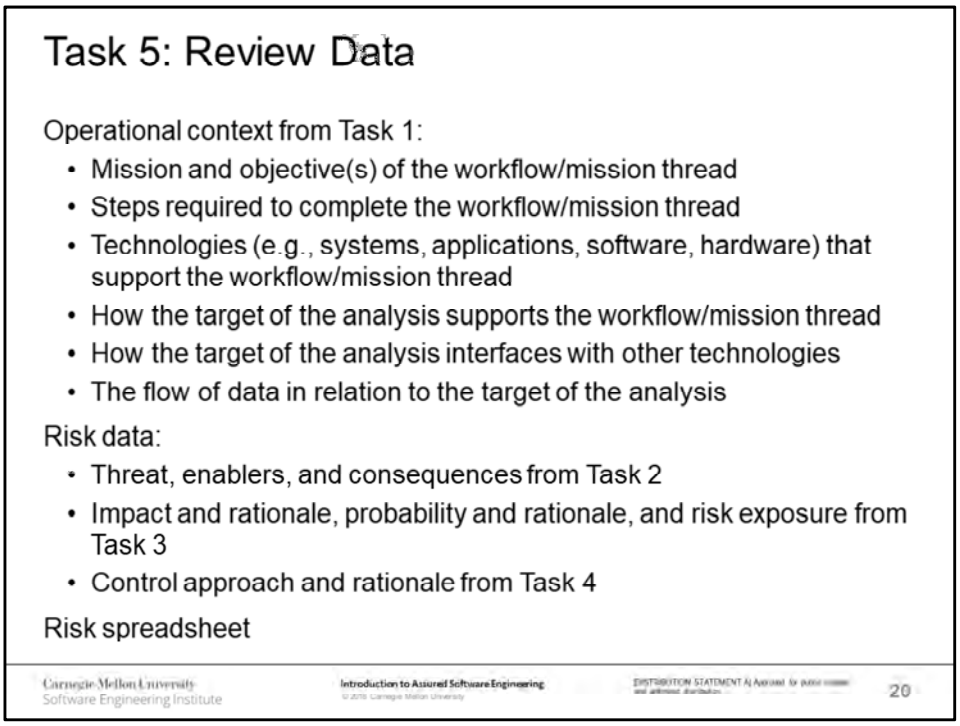


Task 5 comprises two sub-tasks:

- Review data. (Sub-Task 5.1)
- Establish control requirements. (Sub-Task 5.2)

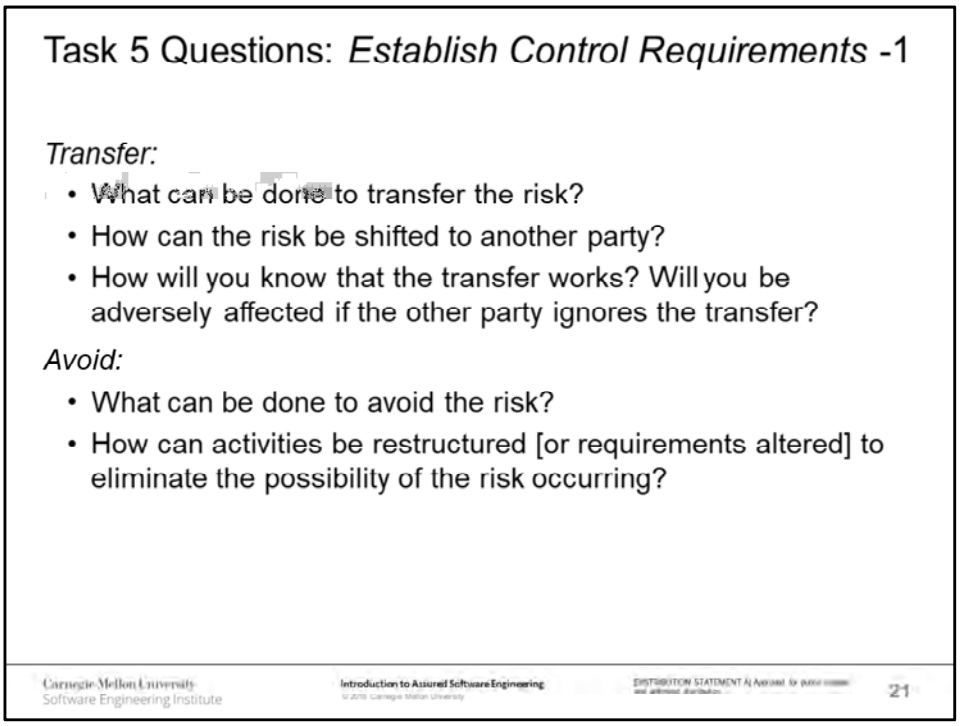
The overarching goal of Task 5 is to develop a control plan for each risk for any cybersecurity risk that is not accepted. A control plan defines a set of actions for implementing the selected control approach. For risks that are being mitigated, their plans can include actions from the following categories:

- *Monitor* and *respond* – Monitor the threat and take action when it is detected.
- *Protect* – Implement protection measures to reduce vulnerability to the threat and to minimize any consequences that might occur.
- *Recover* – Recover from the risk if the consequences or losses are realized.

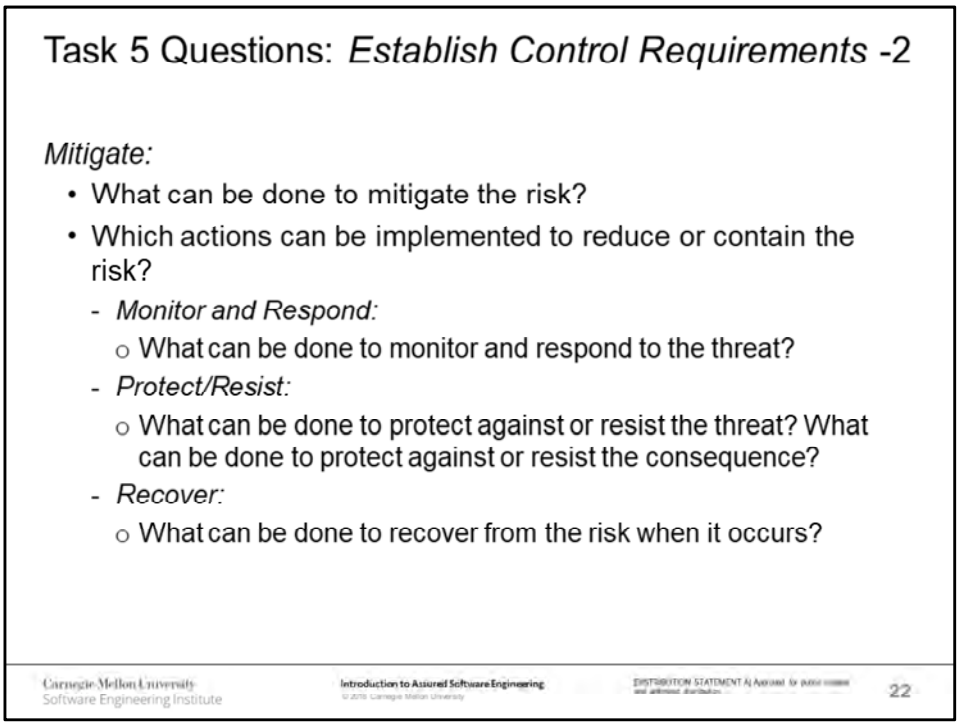


This slide highlights the data that are reviewed during Sub-Task 5.1.

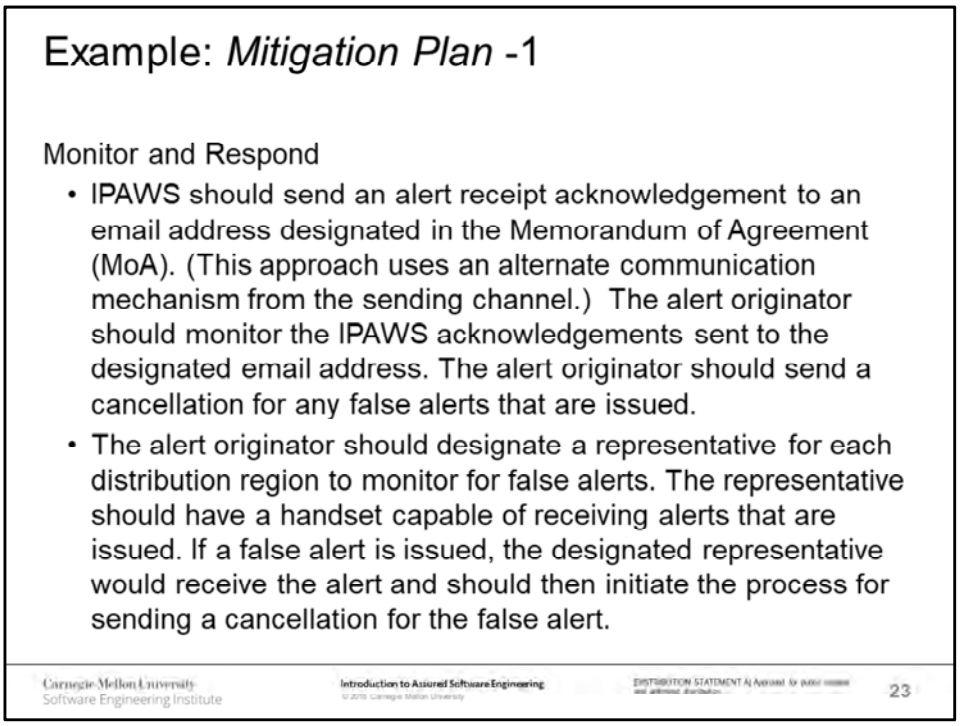
The overarching goal of Task 5 is to develop a control plan for each risk for any cybersecurity risk that will be mitigated, transferred, or avoided.



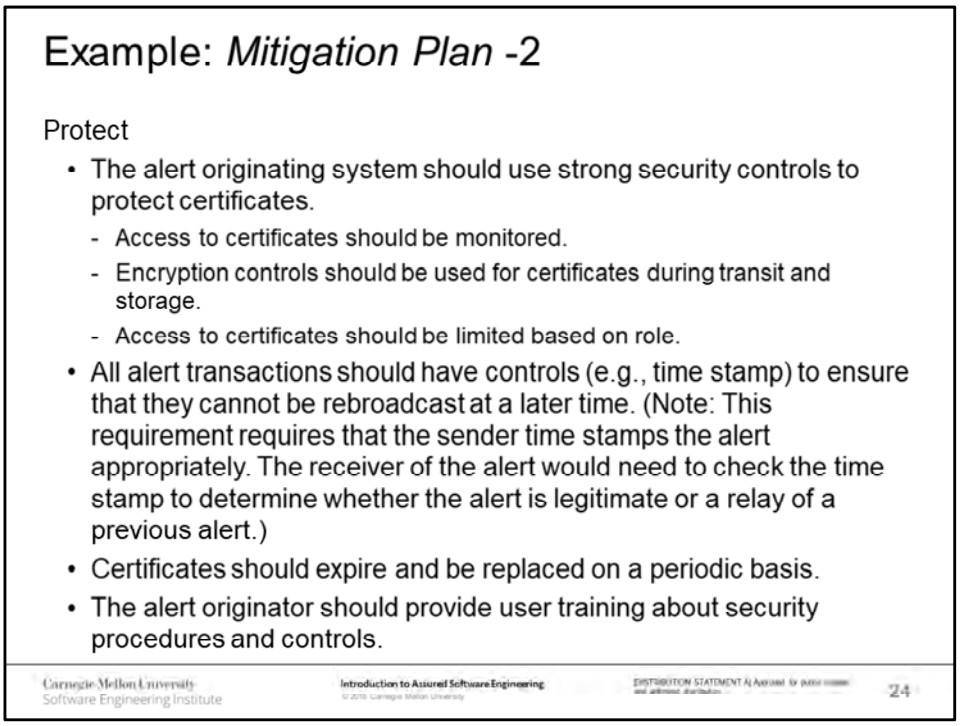
This slide highlights the key questions answered when performing Sub-Task 5.2. The questions on this slide are for risks that are being transferred or avoided.



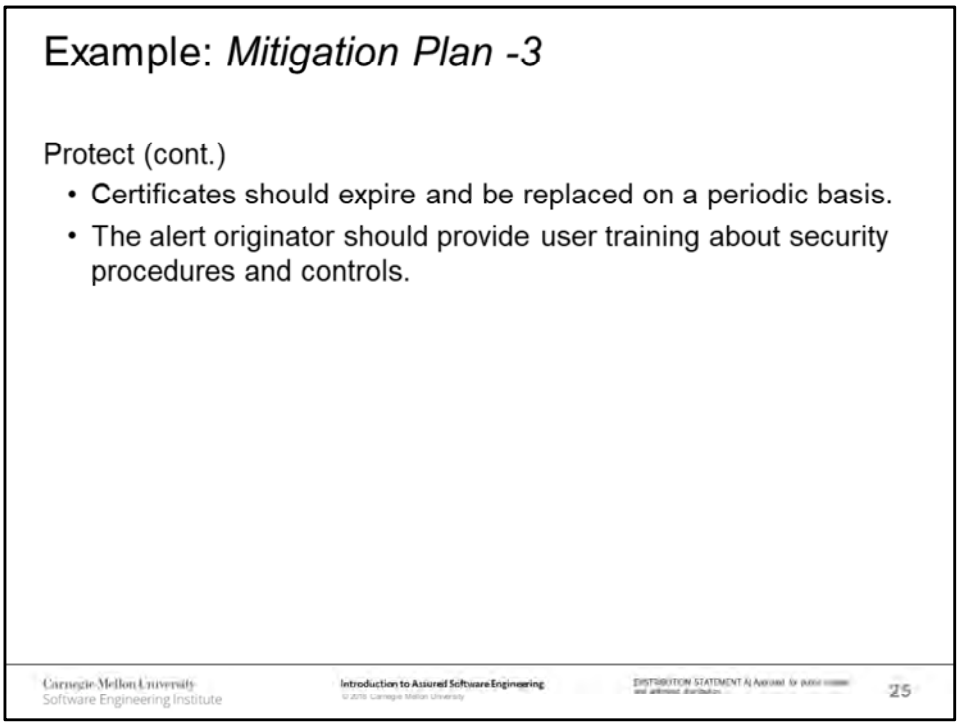
This slide continues the list of key questions from the previous slide. The questions on this slide are for risks that are being mitigated.



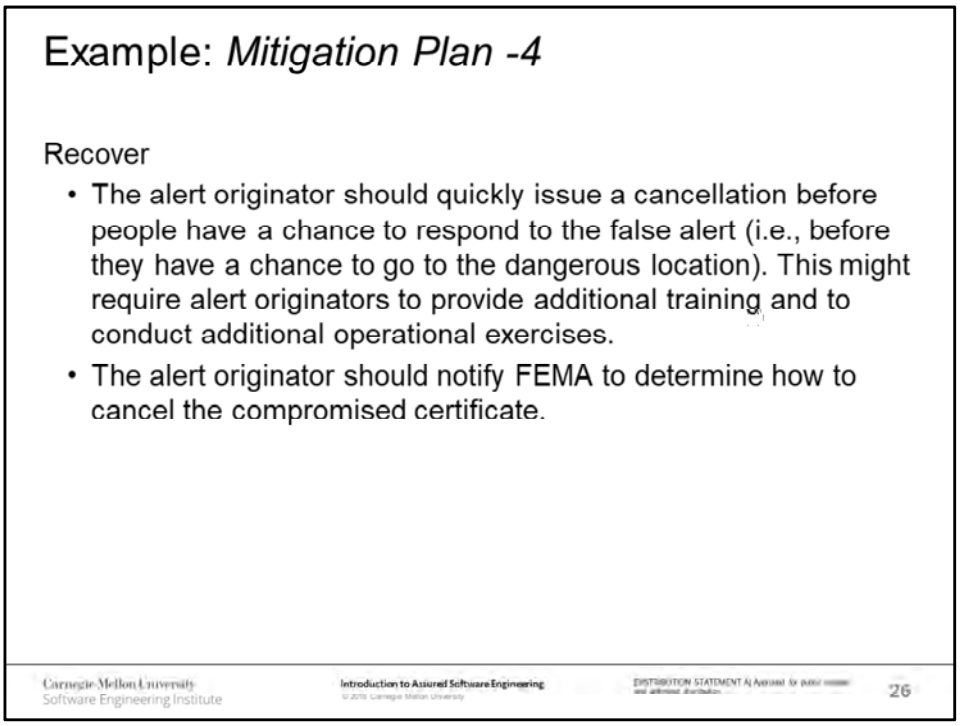
This slide presents part 1 of the mitigation plan for the risk being analyzed (risk 1 from the spreadsheet on slide 18). Mitigation actions for *monitor and respond* are shown.



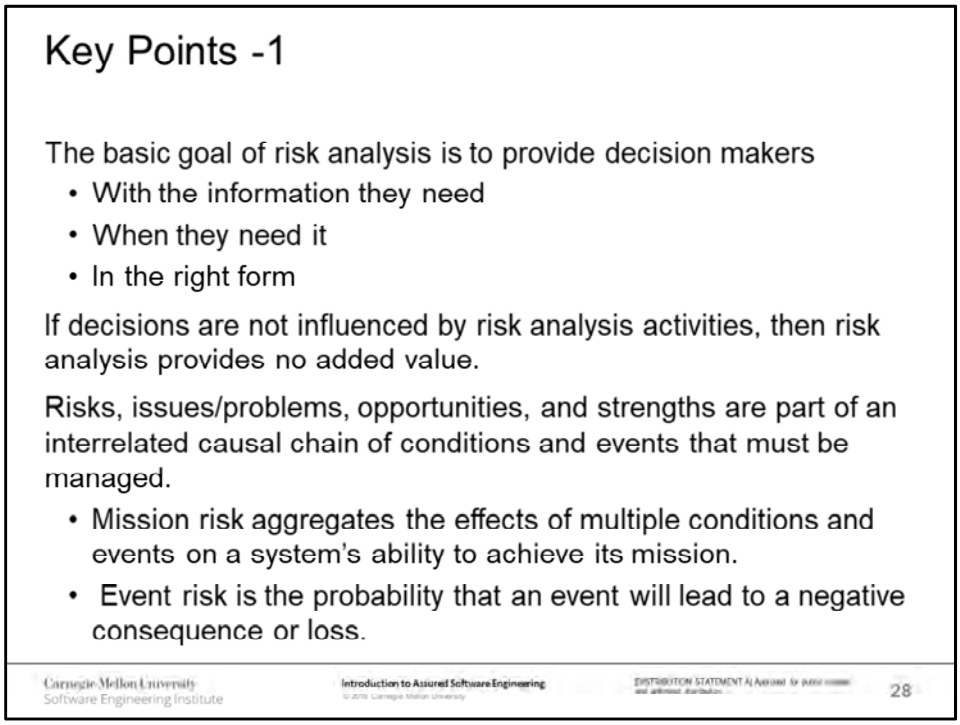
This slide presents part 2 of the mitigation plan for the risk being analyzed (risk 1 from the spreadsheet on slide 18). Mitigation actions for *protect* are shown.



This slide presents part 3 of the mitigation plan for the risk being analyzed (risk 1 from the spreadsheet on slide 18). Additional mitigation actions for *protect* are shown.



This slide presents part 4 of the mitigation plan for the risk being analyzed (risk 1 from the spreadsheet on slide 18). Mitigation actions for *recover* are shown.



This slide summarizes some of the key points from this presentation.

Risk is defined as the probability of suffering harm or loss. Risk management is a systematic approach for minimizing exposure to potential losses. It provides a disciplined environment for

- continuously assessing what could go wrong (assess risk)
- determining which risks to address (plan for controlling risk)
- implementing actions to address high-priority risks and bring those risks within tolerance (plan for controlling risk)

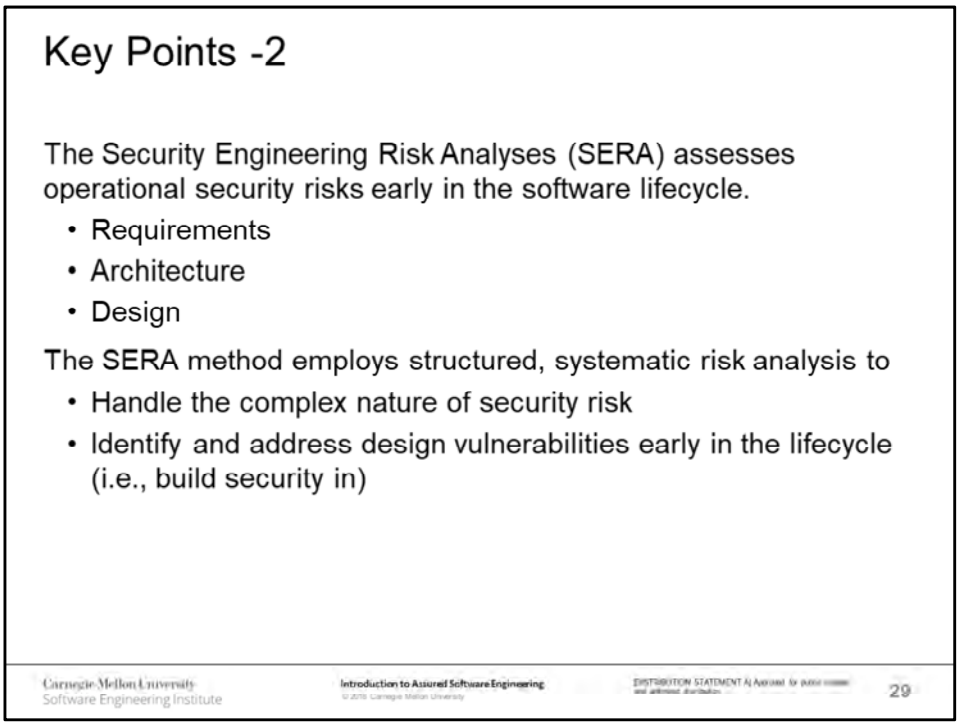
The main goal of any risk management process is to provide decision makers with the information they need

- when they need it
- in the right form
- If decisions are not influenced by risk analysis activities, then risk analysis provides no added value.

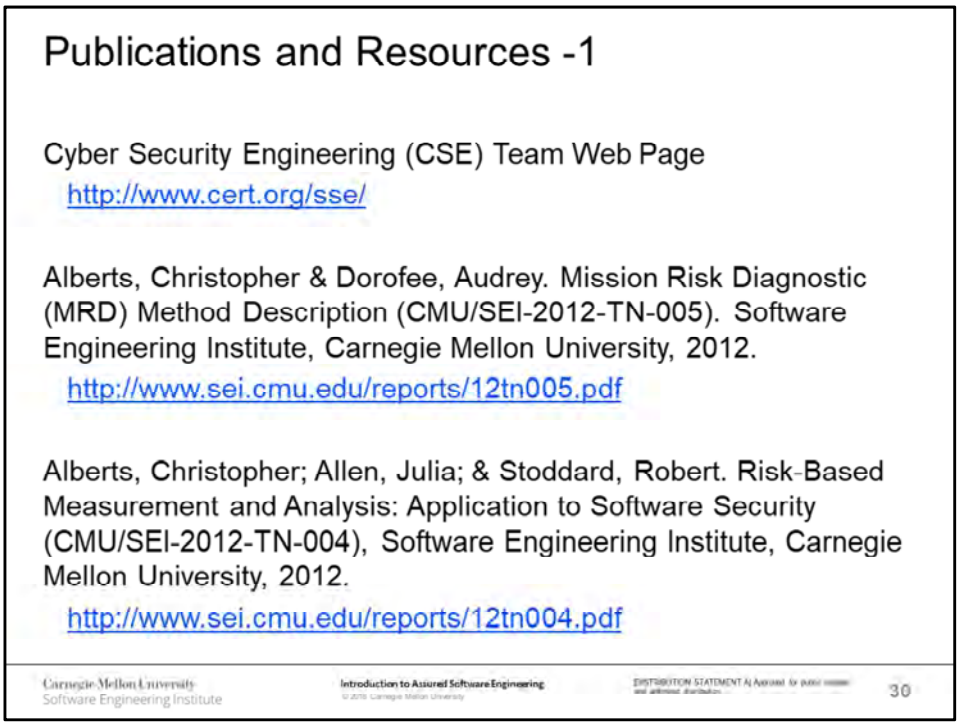
The success or failure of a mission is influenced by the range of circumstances that are present. Risks, issues/problems, opportunities, and strengths are part of an interrelated causal chain of conditions and events that must be managed. Effective risk management requires navigating through this causal chain, assessing the current potential for loss, and implementing strategies for minimizing the potential for loss.

The causal chain can be viewed from two distinct risk perspectives:

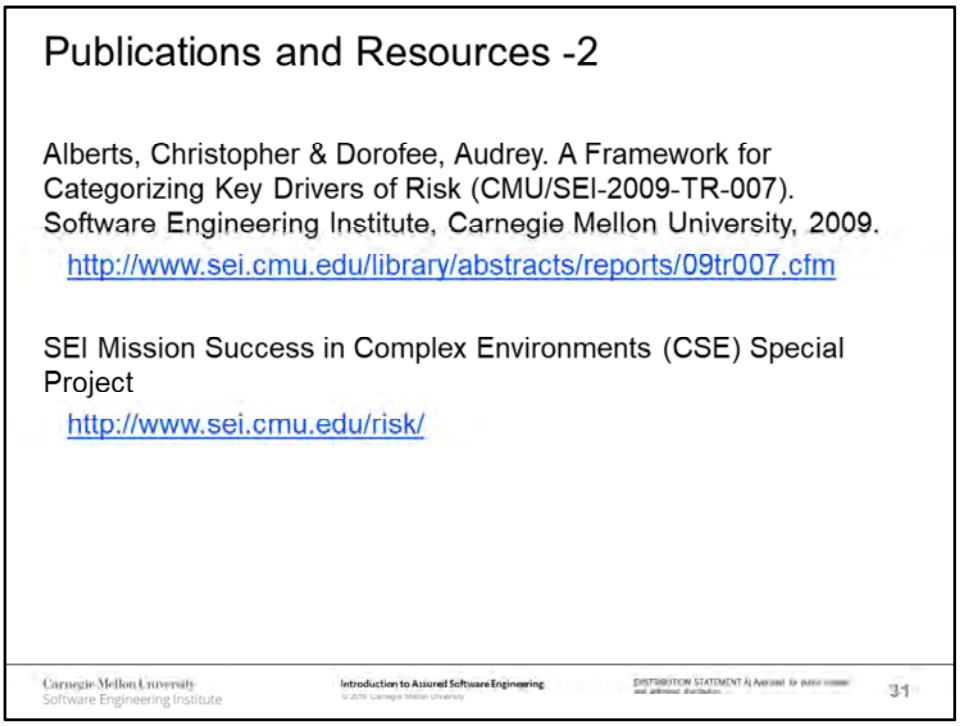
1. Mission risk aggregates the effects of multiple conditions and events on a system's ability to achieve its mission.
2. Event risk is the probability that an event will lead to a negative consequence or loss.



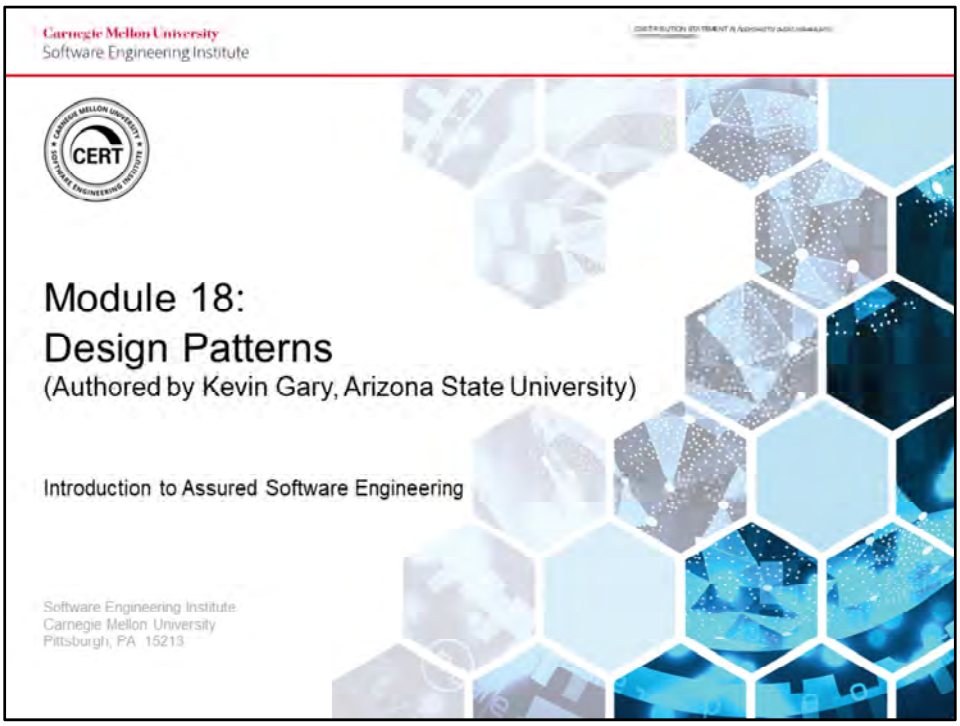
The SERA method is designed for use during early life-cycle activities (e.g., during requirements, architecture, and design). It employs scenario-based risk analysis to handle the complex nature of cybersecurity risk. The goal is to identify design weaknesses early in the life cycle and enable corrective action to be taken. In this way, a subset of critical operational security risks can be mitigated long before a system is deployed.



This slide provides publications and resources for more information on risk management.

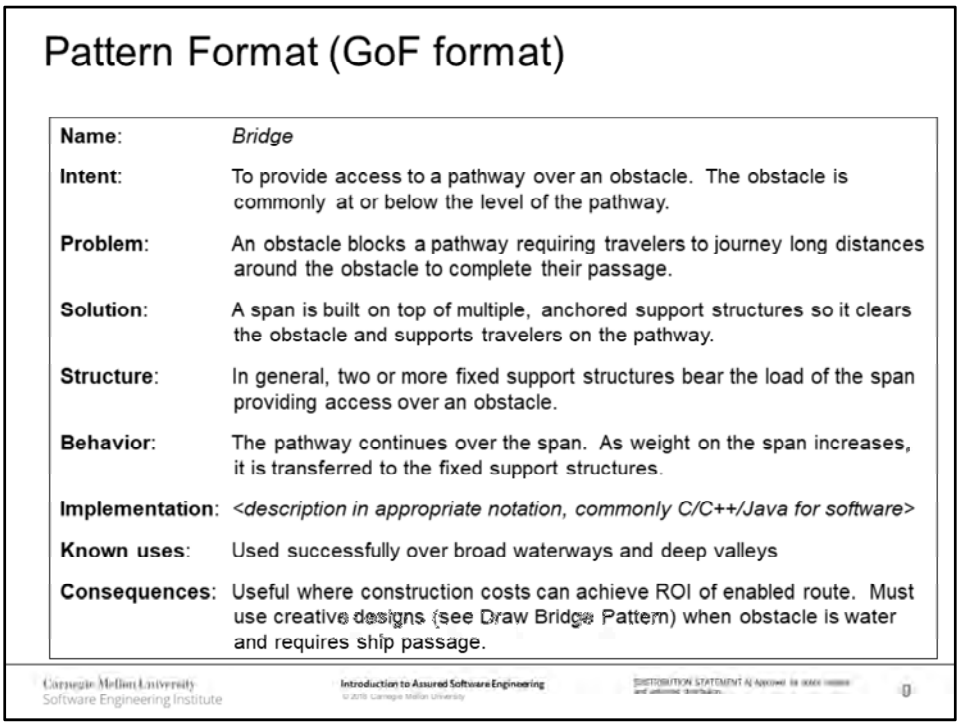


This slide provides additional publications and resources.



ASU site has resources, quizzes and exams that could be useful.

For more resources: https://softwareenterprise.asu.edu/curricular-modules



This is the standard descriptive elements from the GOF, which is also adopted by several others.

Important note is it has it's problems – the primary being the 1-to-1 problem-solution enforcement. Can a problem have multiple solutions? Yes. Can a solution apply to multiple problems? Less likely, but should we rule it out?

Motive the need to describe structure and behavior.

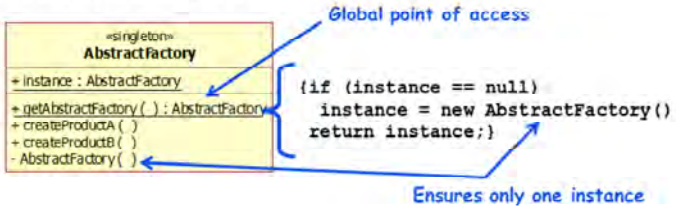
Creational Patterns: Singleton

Ensures a class has only one instance, and provides a global point of access to it

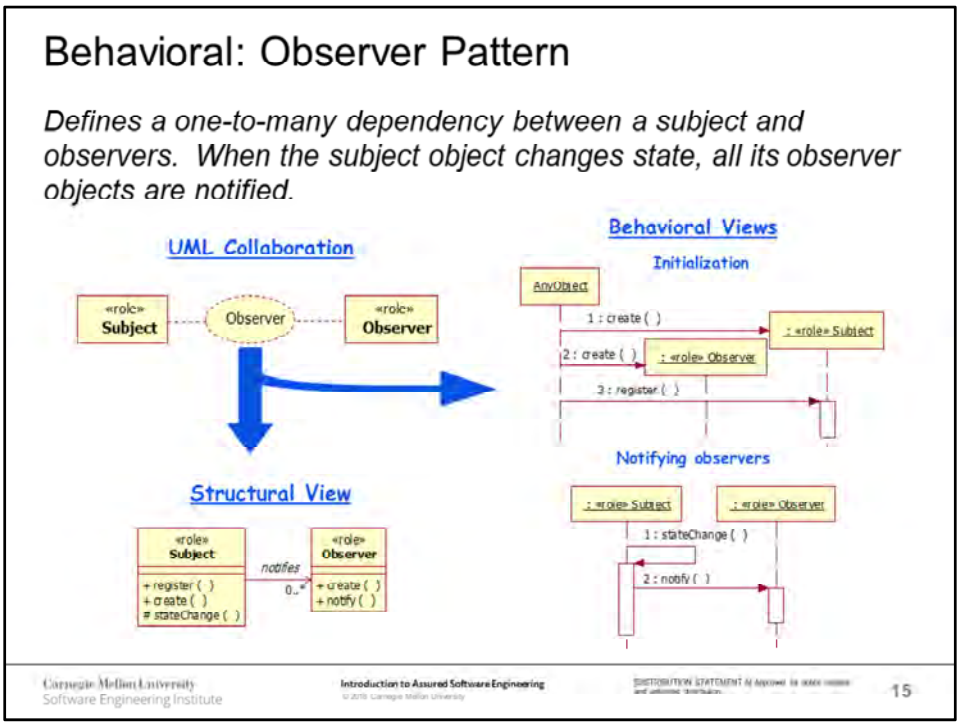
- A very popular pattern and commonly needed by other patterns
- Implementations typically use class-scope to provide global access
- Different than using a class with static variables and methods, as it is still a stateful object

Example – how do clients obtain the AbstractFactory?

- One strategy is a singleton



The instructor can point out that the getAbstractFactory method is a design pattern called "Factory Method".

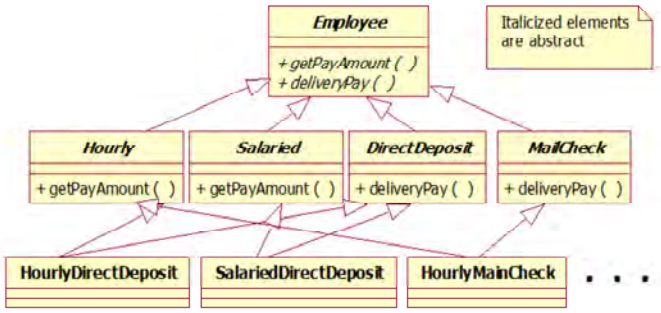


Make the point about documenting collaboration with a single structural view and multiple behavioral views.

Strategy Pattern Example – Inheritance

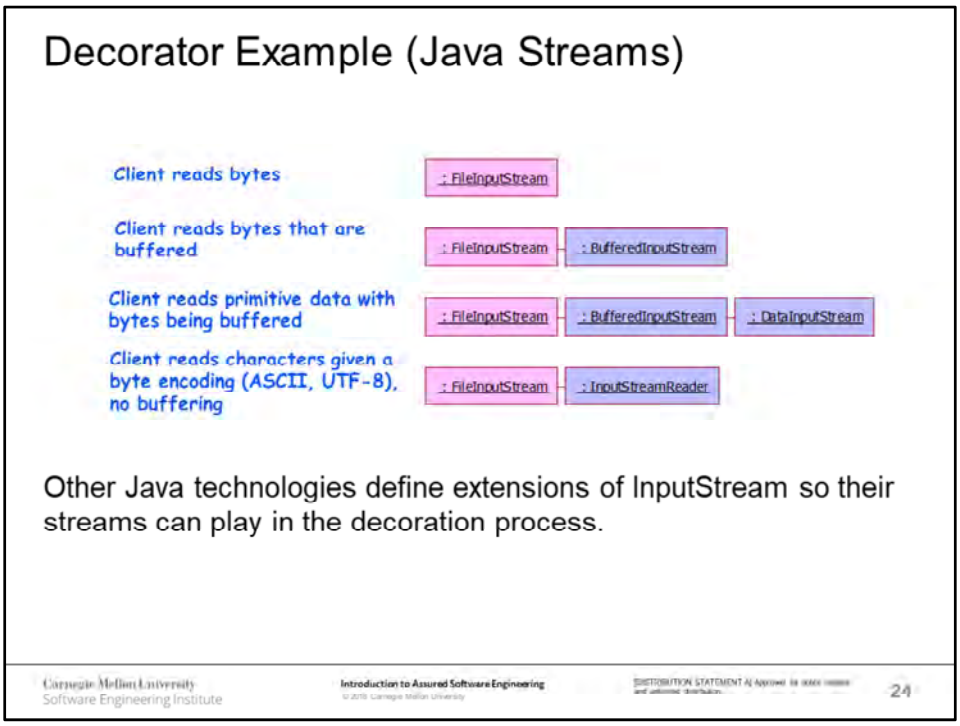
Behold the power of inheritance to really confuse a design!

- Still exposes all permutations

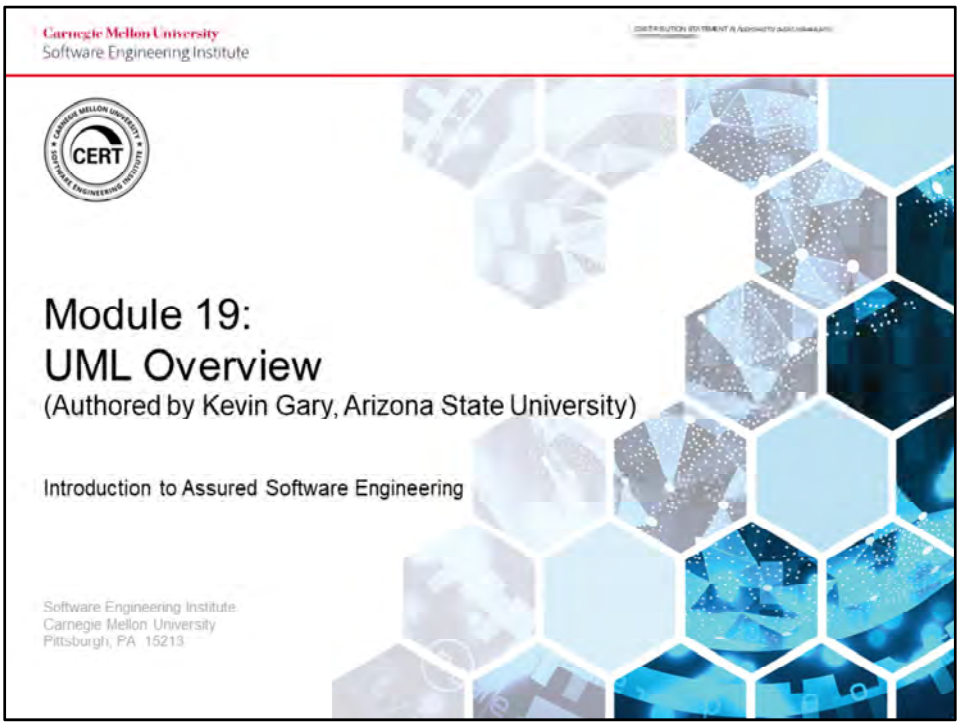


Notice the middle layer is a bunch of Abstract classes which only implement their particular method. The bottom layer has the concrete classes that, though inheritance, "reuse" the behaviors provided by the abstract classes.

This was the hot way to design in early days of OO – pre design patterns.

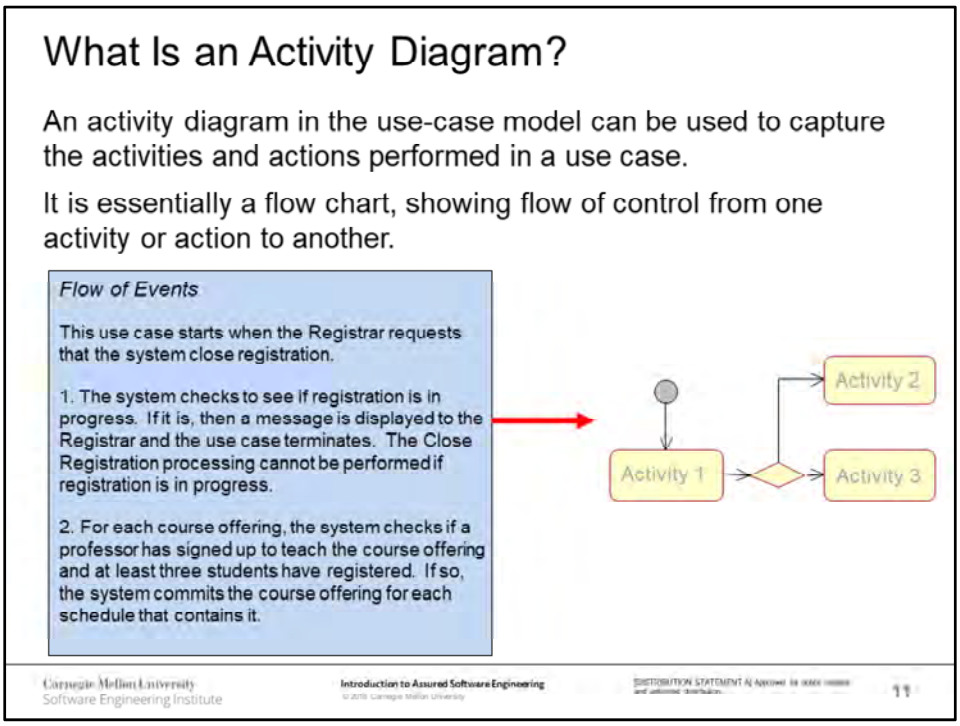


The classes above are invoking the readBytes() method on each other. In the first two, a client calls readBytes() form a file and it is either buffered or non-buffered. In the 3rd example, the client is reading primitive data, readInt(), readFloat() which causes the DataStream to read bytes looking for those primitive values from the file. The last example reads characters and Strings based on an encoding.



ASU site has resources, quizzes and exams that could be useful.

For more resources: https://softwareenterprise.asu.edu/curricular-modules



The workflow of a use case describes that which needs to be done by the system to provide the value the served actor is looking for.

It consists of a sequence of activities and actions that together produce something for the actor.

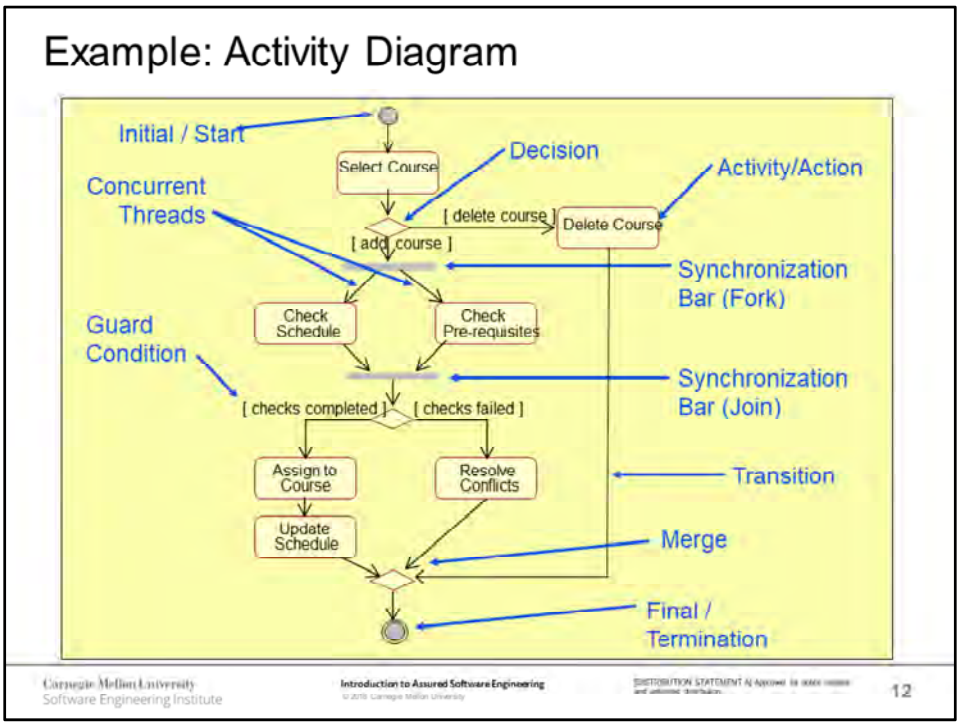
The workflow often consists of a basic flow and one or several alternative flows.

The structure of the workflow can be described graphically with the help of an activity diagram.

The goal of this section is to introduce the students to the concept of an activity diagram. You are not expected to teach them everything about this diagram at this time.

Activity diagrams can also be used to model the workings of an operation, an object, business modeling, or anything that involves modeling the sequential steps in a computational process.

This course focuses on using activity diagrams to model the flow of events in a use case



An activity diagram may include the following elements:

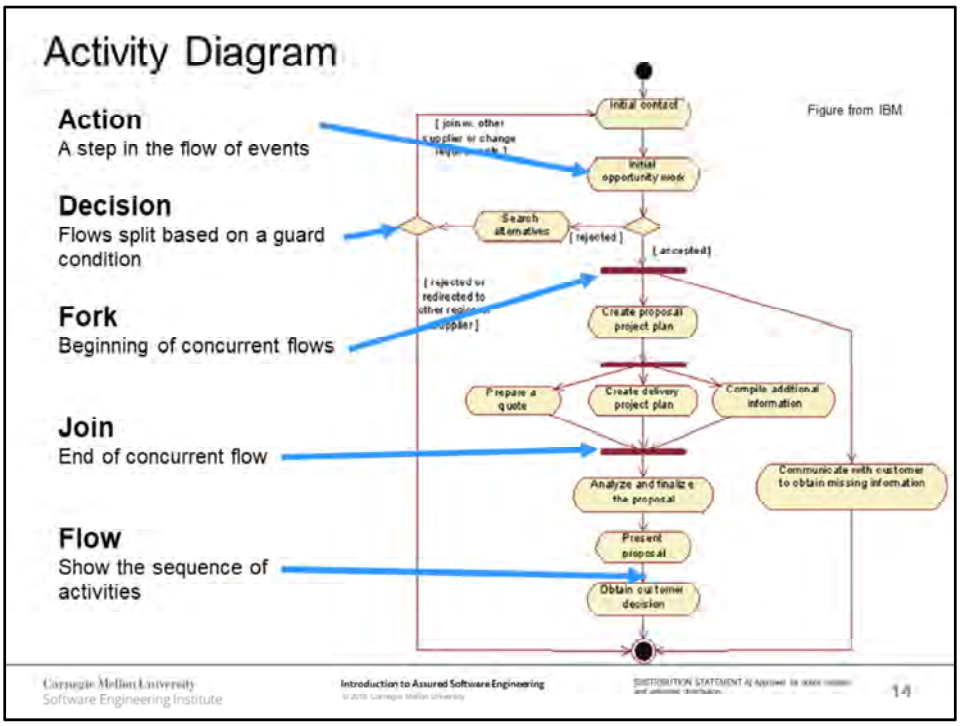
Activity/Action represents the performance of a step within the workflow.

Transitions show the activity/action that follows.

Decisions evaluate conditions defined by guard conditions. These guard conditions determine which of the alternative transitions will be made and, thus, which activities are performed. You may also use the decision icon to show where the threads merge again. Decisions and guard conditions allow you to show alternative threads in the workflow of a use case.

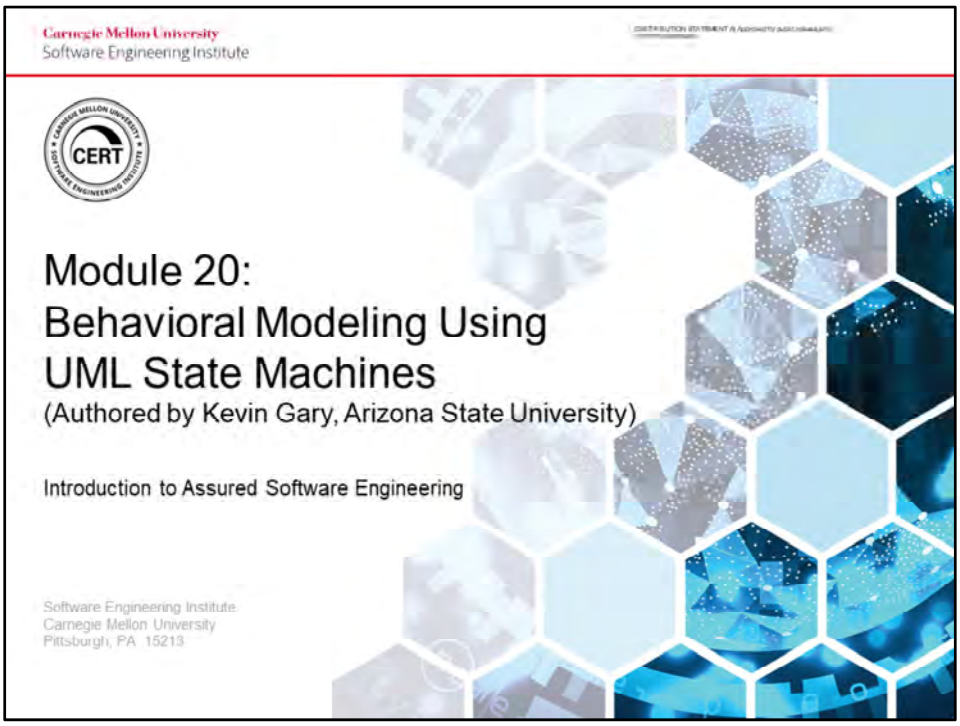
Synchronization bars show parallel sub-flows. They allow you to show concurrent threads in the workflow of a use case.

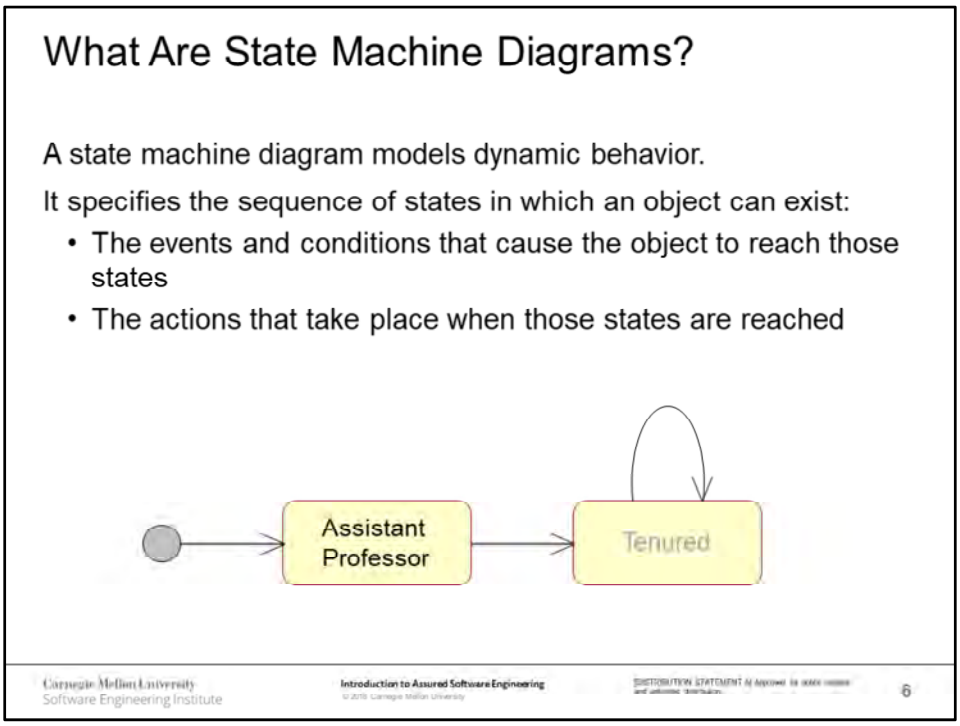
Walk the students through the activity diagram and explain each component (decision, fork, join, and so on).



Activities describe graphically the flow of events of a use case. The flow of events consists of a sequence of activities that together produce something of value for the actor. The flow of events consists of a basic flow and one or several alternative flows.

- Actions: Represent the performance of an activity or step within the flow of events.
- Flow/Edge: Show what activity state follows after another.
- Decision/Merge Control which flow (of a set of alternative flows) follows once the activity has been completed, based on a guard condition. Decisions are used to show alternative threads in the flow of events of a use case.
- Forks/Joins: Show the beginnings and ends of parallel subflows. Forks and joins are used to show concurrent threads in the flow of events of a use case.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 20:
Behavioral Modeling Using
UML State Machines
(Authored by Kevin Gary, Arizona State University)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



A state machine diagram is typically used to model the discrete stages of an object's lifetime. They show the sequences of state that an object goes through, the events that cause a transition from one state to another, and the actions that result from the state change. State machine diagrams are closely related to activity diagrams.

Each state represents a named condition during the life of an object in which it satisfies some condition or waits for some event. A state machine diagram typically contains one start and multiple end states. Transitions connect the various states on the diagram. Like activity diagrams, decisions, and synchronizations may also appear on state machine diagrams.

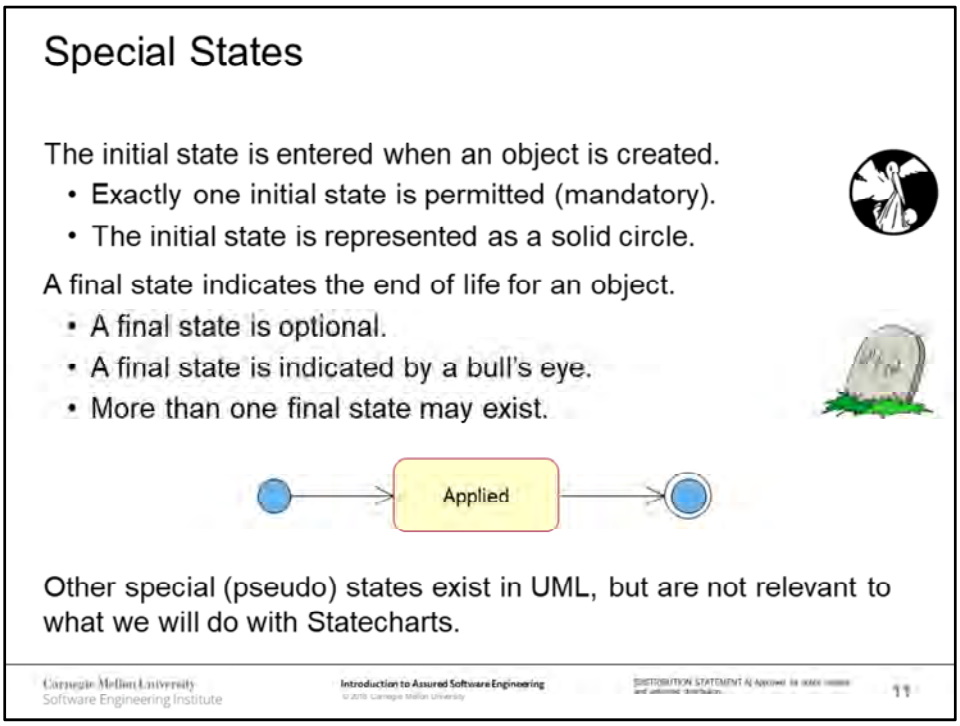
State machines are used to model the dynamic behavior of a model element, and more specifically, the event-driven aspects of the system's behavior. State machines are specifically used to define state-dependent behavior, or behavior that varies depending on which state the model element is in. Model elements whose behavior does not vary with the state, do not require state machines to describe their behavior. These elements are typically passive classes whose primary responsibility is to manage data.

Introduce the concept of state machines.

A state machine diagram shows a state machine, emphasizing the flow of control from state to state.

Using the previous example, J. Clarke was an associate professor before she achieved tenure and became a full Professor.

Another type of state machine that has been formalized in UML 2 is the Protocol State Machine. This is used to express usage protocols by defining rules on the invocation of operations or exchange of messages that a behavioral state machine or procedure may perform. This course does not cover these diagrams



The initial state indicates the default starting place for the state machine or sub-state. An initial state is represented as a filled black circle.

The final state indicates the completion of the execution of the state machine or the enclosing state. A final state is represented as a filled black circle surrounded by an unfilled circle.

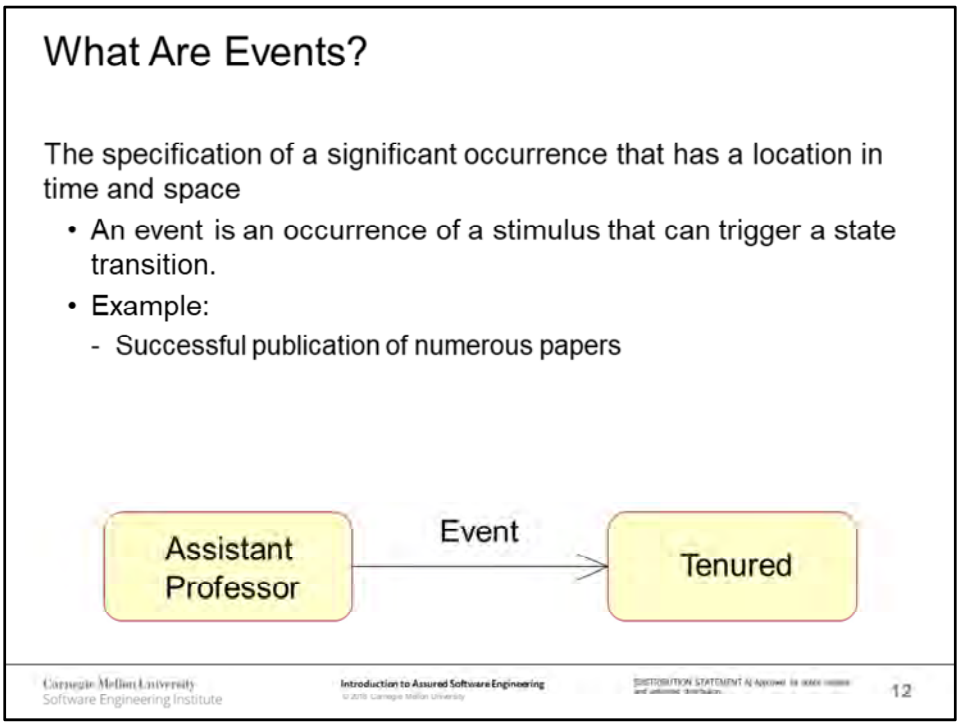
Initial and final states are actually pseudo-states. Neither may have the usual parts of a normal state, except for a name.

<u>Explain the need for start and final states.</u>

There is exactly one initial (start) state and 0..* final (end) states.

To emphasize why an initial state is mandatory, ask the students to think about how they would read a diagram without an initial state.

Note: Refer to the statement that "Only one initial state is permitted." This is not always true. When you have nested states, there can be an initial state within each nested state AND the one outside of them

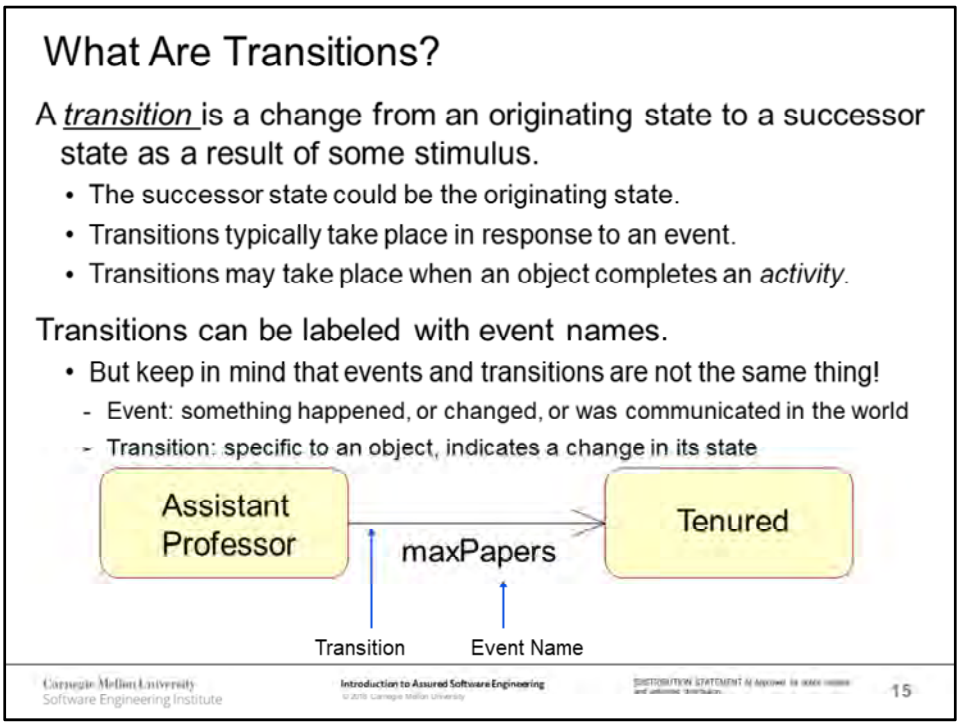


In the context of the state machine, an event can be defined as an occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

A signal or call may have parameters whose values are available to the transition, including expressions for the guard conditions and action.

It is also possible to have a triggerless transition, represented by a transition with no event trigger. These transitions, also called completion transitions, are triggered implicitly when their source state has completed its action. They are implicitly triggered on the completion of any internal 'do activity' in the state.

Explain events on a state machine



A Transition can be defined as:

A relationship between two states indicating that an object in the first state performs certain actions and enters a second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to “fire.” Until the transition fires, the object is said to be in the “source” state. After it fires, it is said to be in the “target” state.

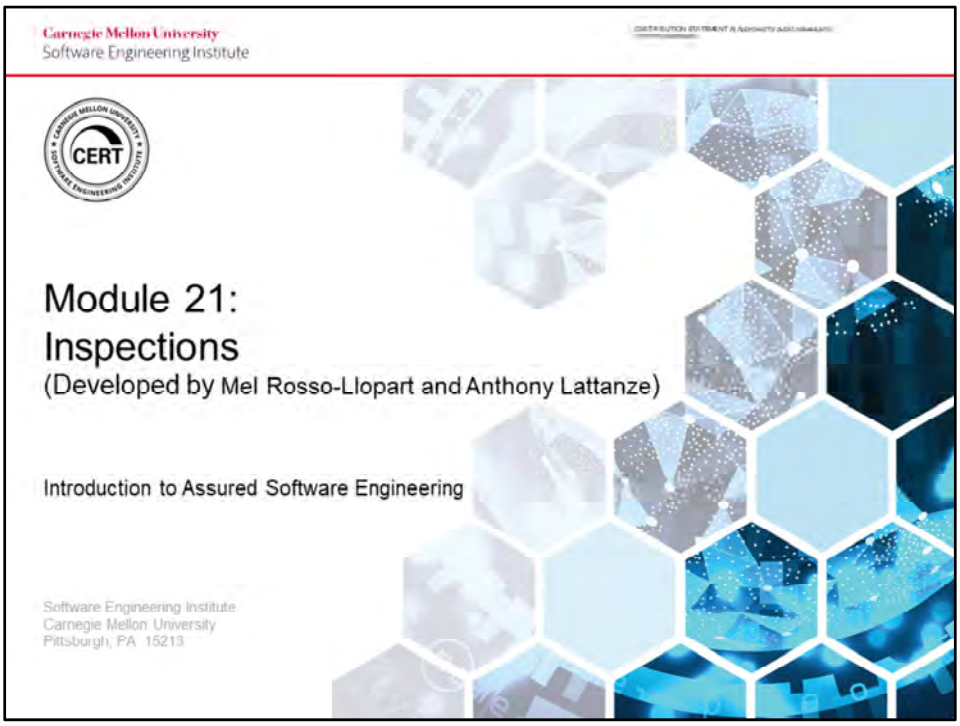
You can show one or more state transitions from a state as long as each transition is unique. Transitions originating from a state can not have the same event unless there are conditions on the event.

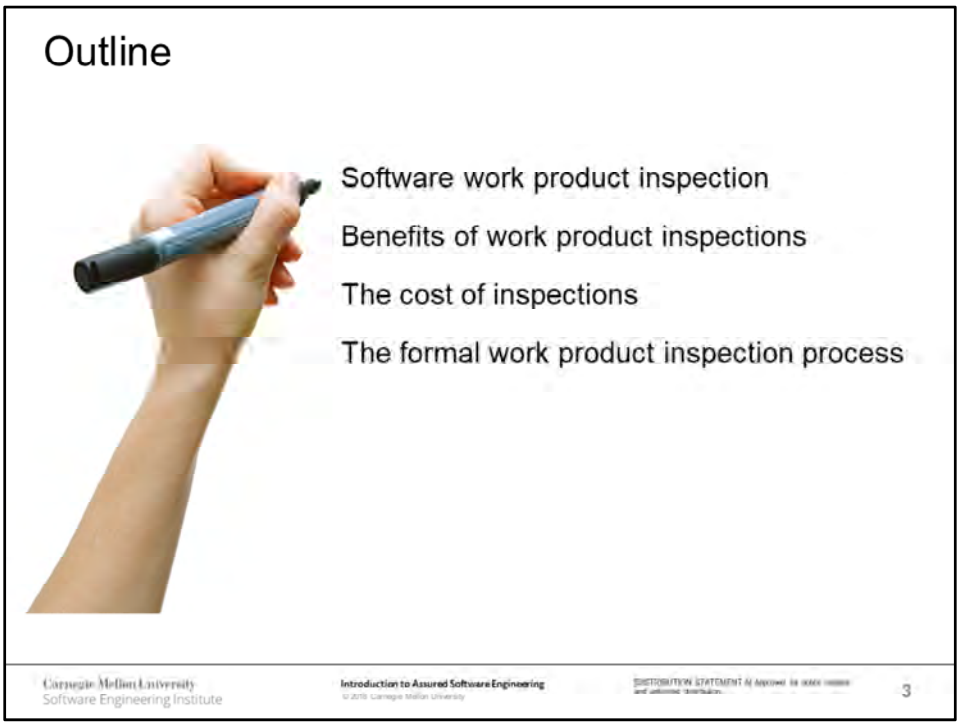
The icon for a state transition is a line with an arrowhead pointing toward the destination state.

Label each state transition with the name of at least one event that causes the state transition. You do not have to use unique labels for state transitions because the same event can cause a transition to many different states but it is recommended that they have unique labels.

Explain transitions on a state machine.

Transitions are not bi-directional. If transitions need to run both ways between two states, you can draw two different transitions going in opposite directions


Carnegie Mellon University
Software Engineering Institute
CERT
Module 21:
Inspections
(Developed by Mel Rosso-Llopart and Anthony Lattanze)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

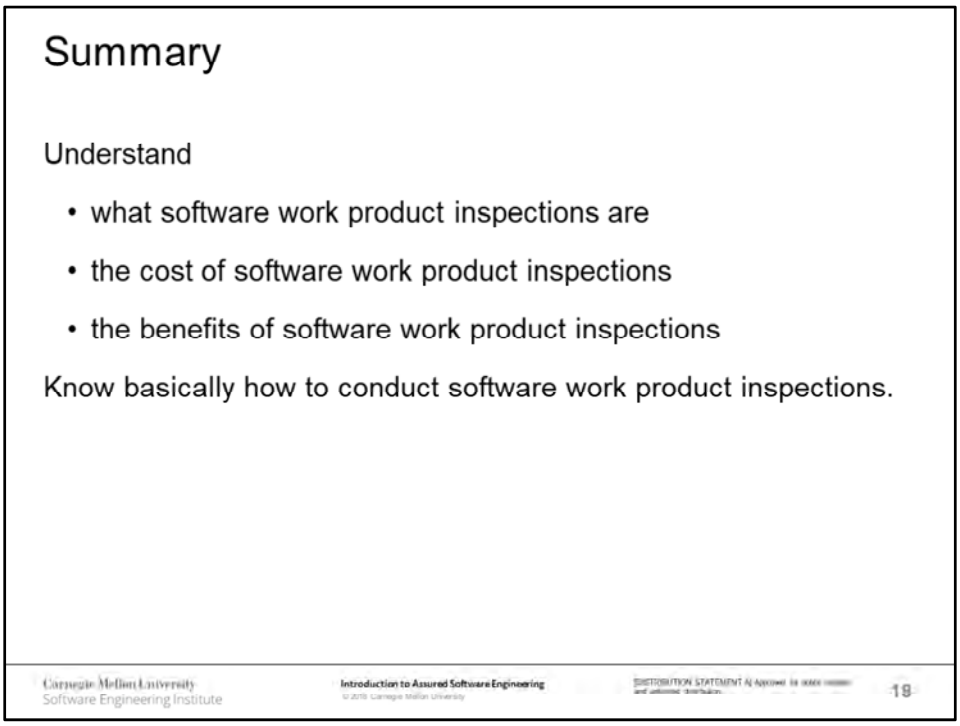


List the learning objectives for this particular session. For example:

The student will learn

- the importance of work product inspections
- how to organize work product inspection meetings
- techniques for reviewing work products
- what data to collect to measure effectiveness

Make sure that if you use sub-bullets, points are in parallel with the main point (as shown in the example above).



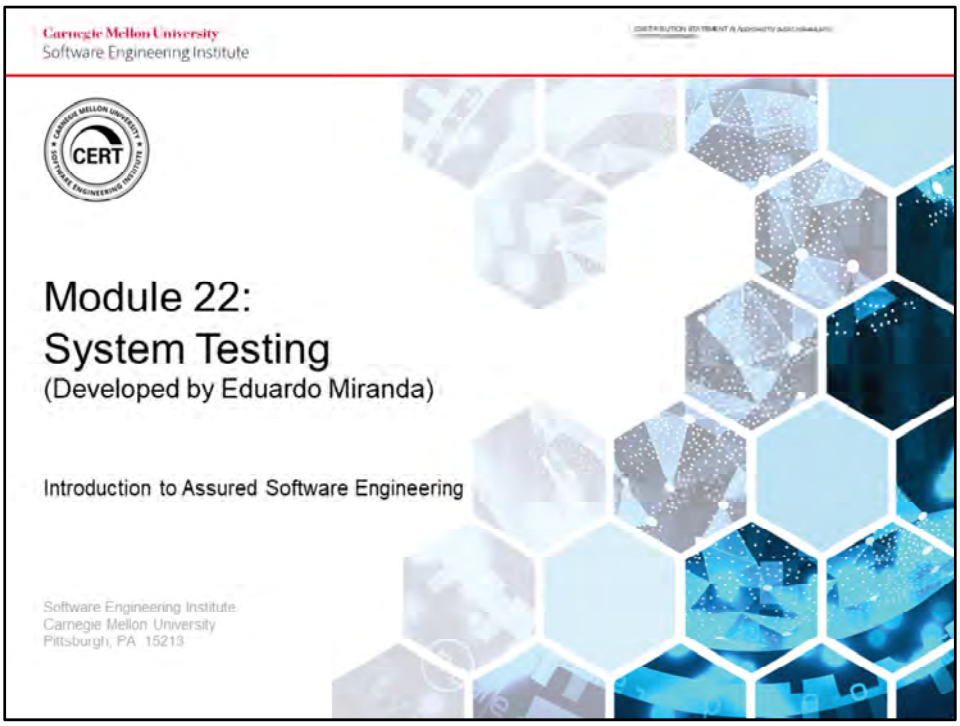
Summarize the main points of the talk. For example:

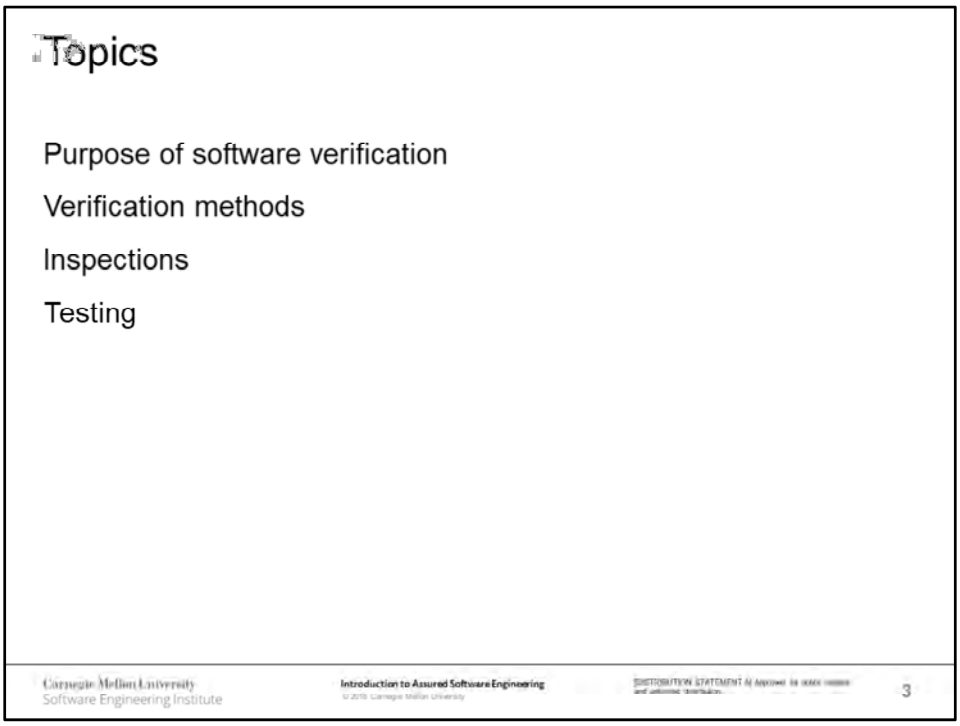
In this session we

- built a case for why work product inspections are important
- presented techniques for reviewing work products
- presented examples of how to organize inspection meetings
- discussed the type of data to collect from work product inspections and how to analyze it

Make sure that the summary is aligned with the lesson objectives.

Again, make sure sub-bullets are parallel.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 22:
System Testing
(Developed by Eduardo Miranda)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

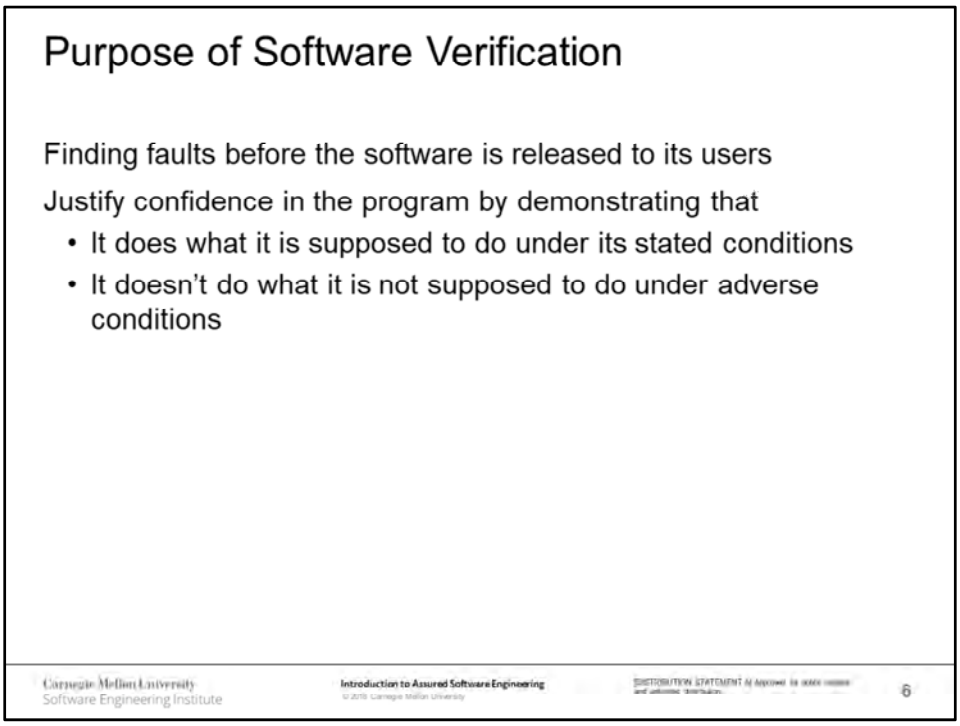


Instructor note: Some of the content of this module may be too advanced for your students. Use your professional judgment in deciding what to present to them, and what you can actually expect them to absorb.

This is the point where my colleagues normally start talking about big and terrible incidents, those that make the headlines news like: the Therac-25, a radiation therapy machine which in the 80s killed 2 and injured 4 people (The problem was traced to a seldom used sequence of key strokes – which had been masked in previous versions of the software by the presence of a hardware interlock). Or the Ariane 5, a rocket, that blew up in 1996 (integer overflow, a number too big to be represented by the data type selected turns from positive to negative) or the 1998 Mars Climate Observer destruction where a mistake in the use of English vs. Metrics units resulted in the destruction of the spacecraft.

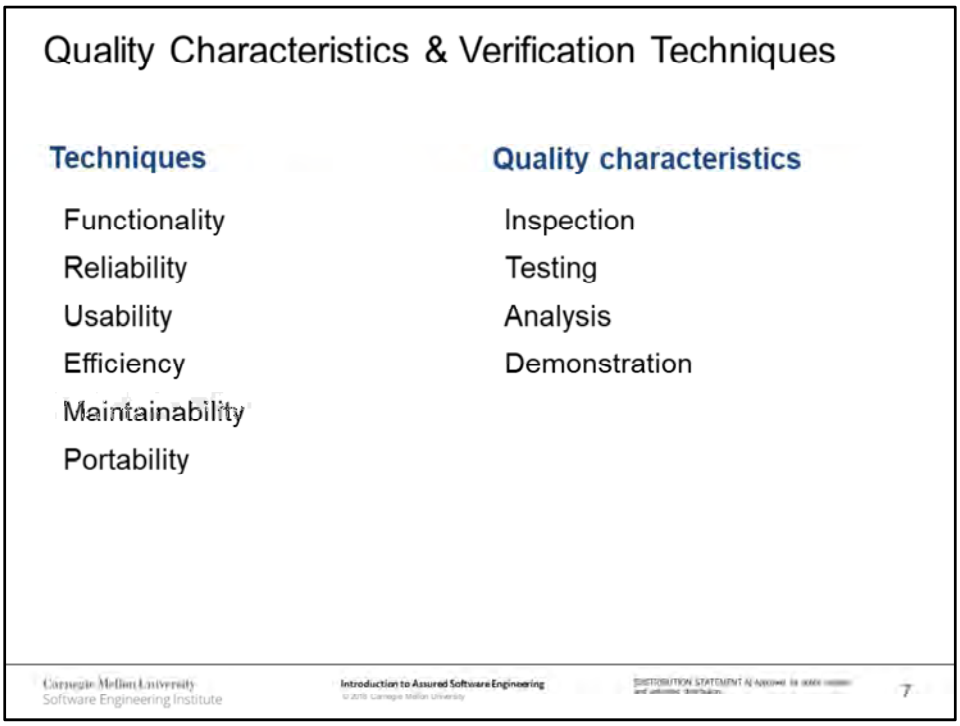
Thank goodness, these disasters do not happen every day, but if we look around there is plenty of bad software. These are just three examples of software failures.

The National Institute of Standards and Technologies in a 2002 reports estimates the national annual costs of an inadequate infrastructure for software testing in a range from $22.2 to $59.5 billion. Over half of these costs are borne by software users in the form of error avoidance and mitigation activities. The remaining costs are borne by software developers and reflect the additional testing resources that are consumed due to inadequate testing tools and methods.



So, in this talk we are going to look at software verification which is the process by which we try to eliminate bugs before a software system is released to its users and by which we gain confidence that the system works. That is: we want to be reasonably sure the software does what it is supposed to do and doesn't do what is not supposed to.

Notice that I said justify confidence and not prove, this is because in all but the most trivial systems we can not exhaustively test a system because of the sheer numbers of test cases required



The quality of a software system can be defined in terms of how well it satisfies a number of quality characteristics. Usually these quality characteristics fall in one of six categories. The International Standards Organization (ISO) in its standard ISO-9126 defined 6 quality characteristics:

- Functionality: Does the software provide the required functions
- Suitability, accuracy, interoperability, security, functionality compliance
- Reliability How reliable is the software?
- Maturity, fault tolerance, recoverability, reliability compliance
- Usability Is the software easy to use?
- Understandability, learnability, operability, attractiveness, usability compliance

Efficiency How efficient is the software?

Time behavior, resource utilization, efficiency compliance

Maintainability How easy is it to modify it?

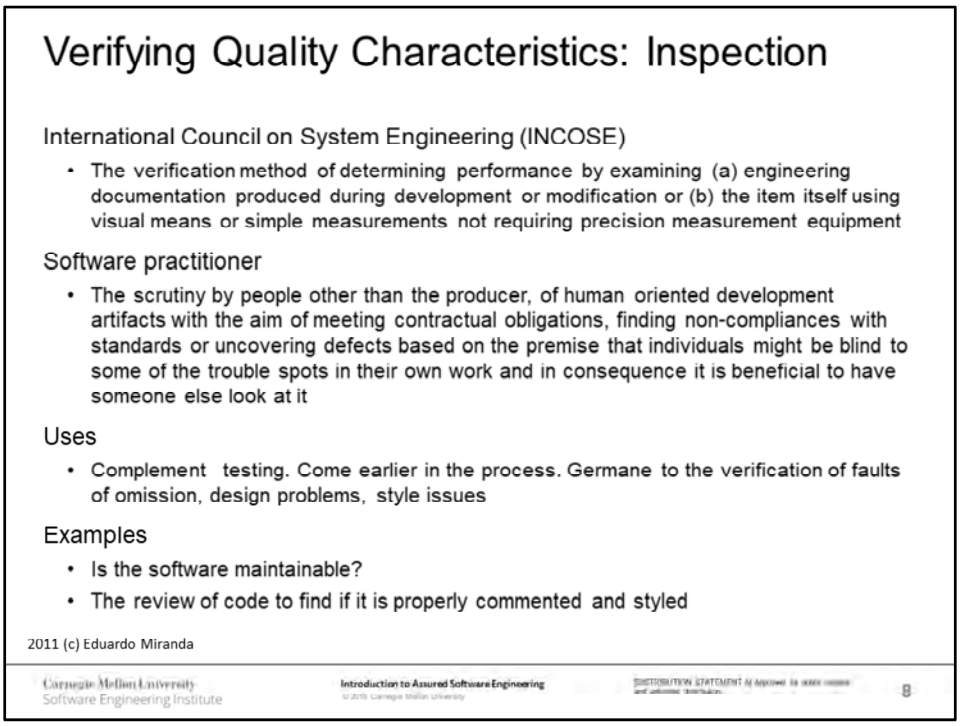
Analyzability, changeability, testability, maintainability compliance

Portability How easy is it to transfer the software to another environment?

Replace ability, environment? Adaptability, install ability, portability compliance

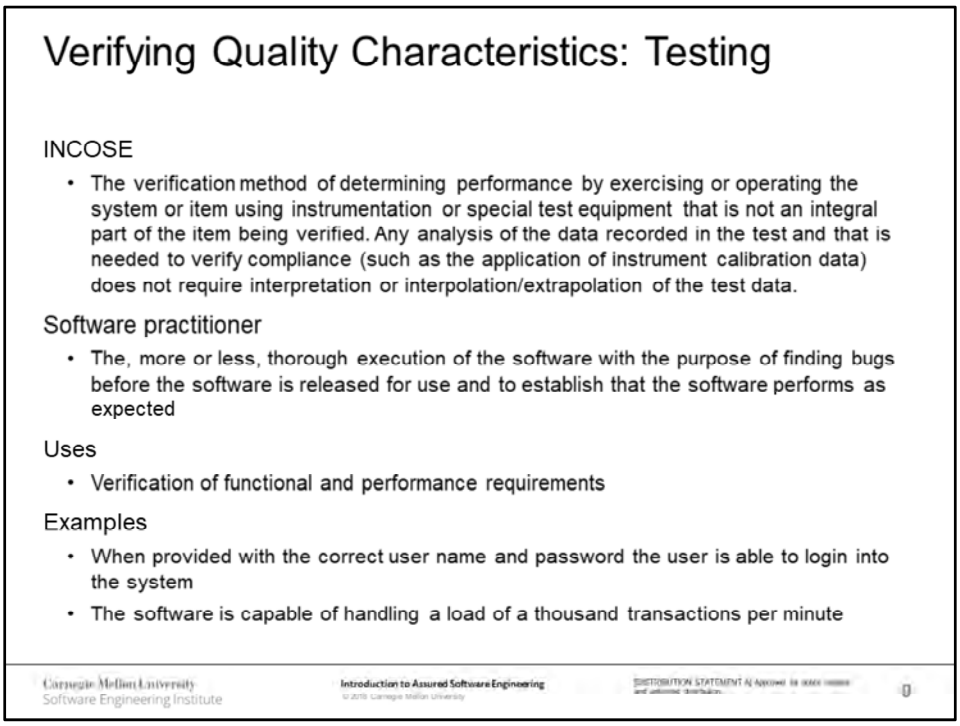
To verify these quality characteristics we will use diverse techniques, each which is categorized according to its nature

- Inspection
- Testing
- Analysis
- Demonstration

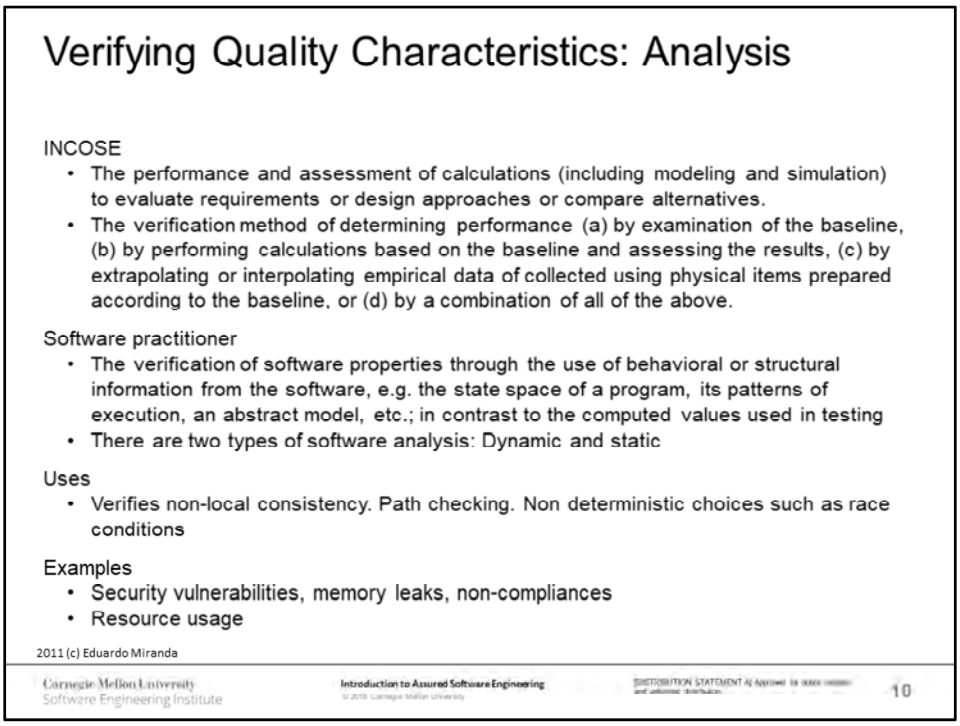


The examination by people other than the producer, of human oriented development artifacts, that is we are going to look at documents or at the source code of an application with the aim of meeting contractual obligations, finding non-compliance with standards or finding defects based on the premise that individuals might be blind to some of the trouble spots in their own work and in consequence it is beneficial to have someone else look at it.

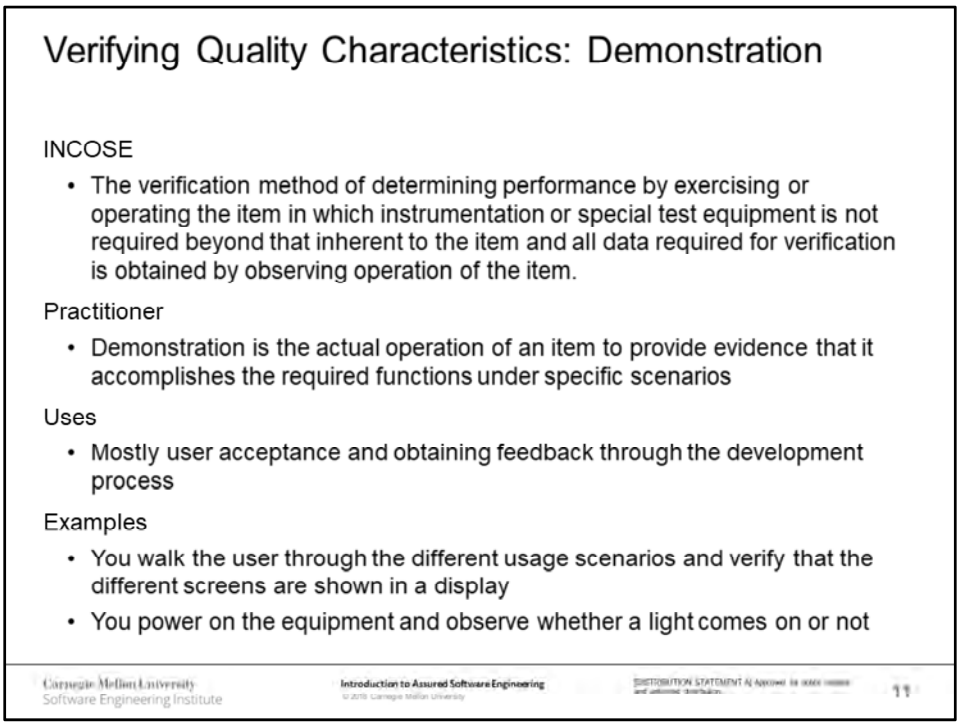
In the case of Vista, for example, two or three people could look at a clinical reminder and discuss its intent



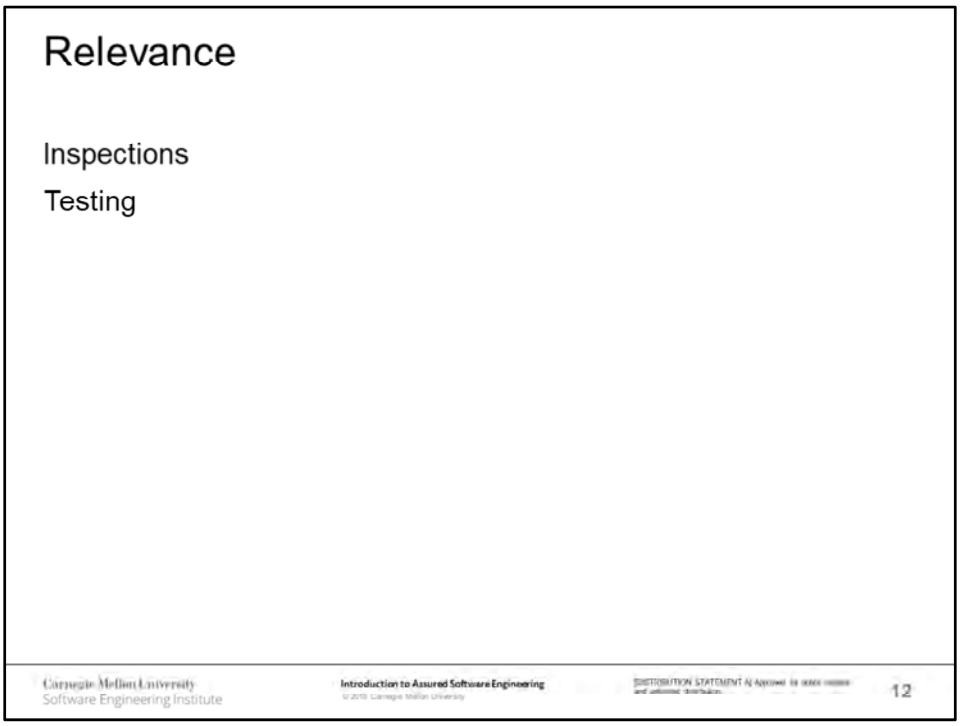
When we test something our purpose is to make it fail. As a matter of fact, a good test case is one that uncovers a fault.



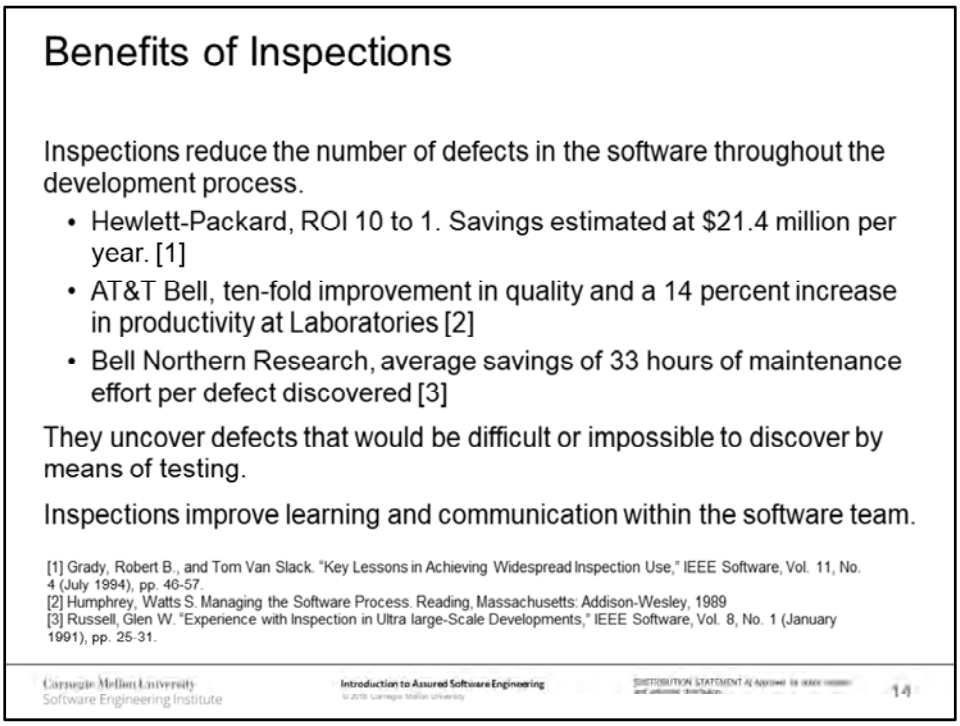
Analysis involves the creation of a model of the system and the verification using the model instead of the actual software. The advantage of analysis is that by omitting all the details that are not relevant to a specific verification it is possible to formally prove (or disprove) that a software exhibits (or not) a certain property.



Demonstration is more positive than testing. Here we are trying to show somebody that what we did indeed does what is supposed to do.

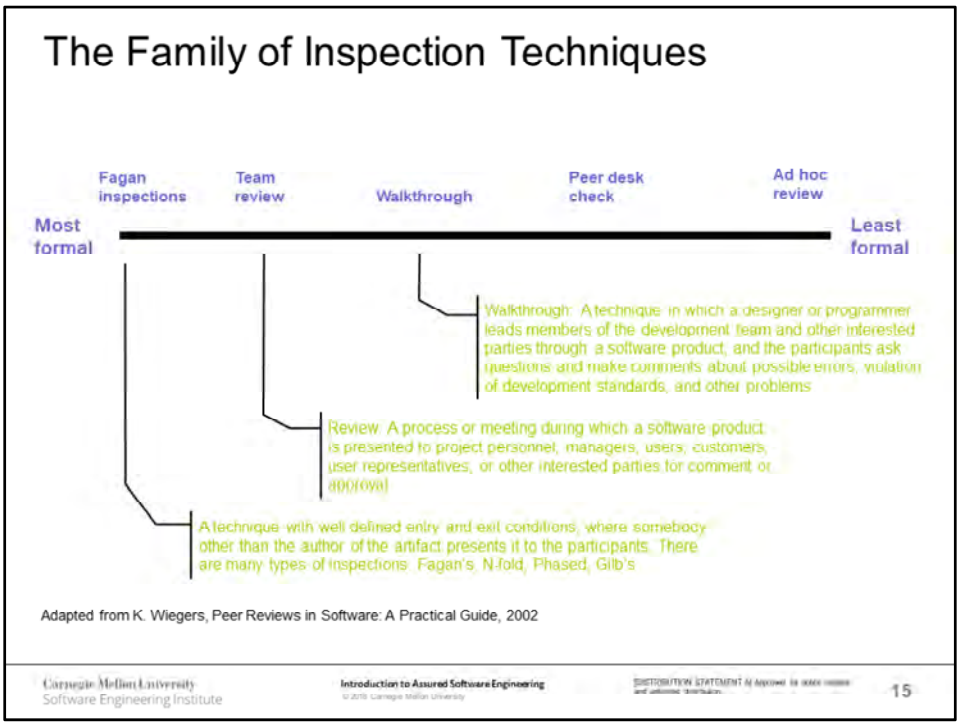


Of these four techniques I will concentrate in the two that I believe are more relevant to the audience.

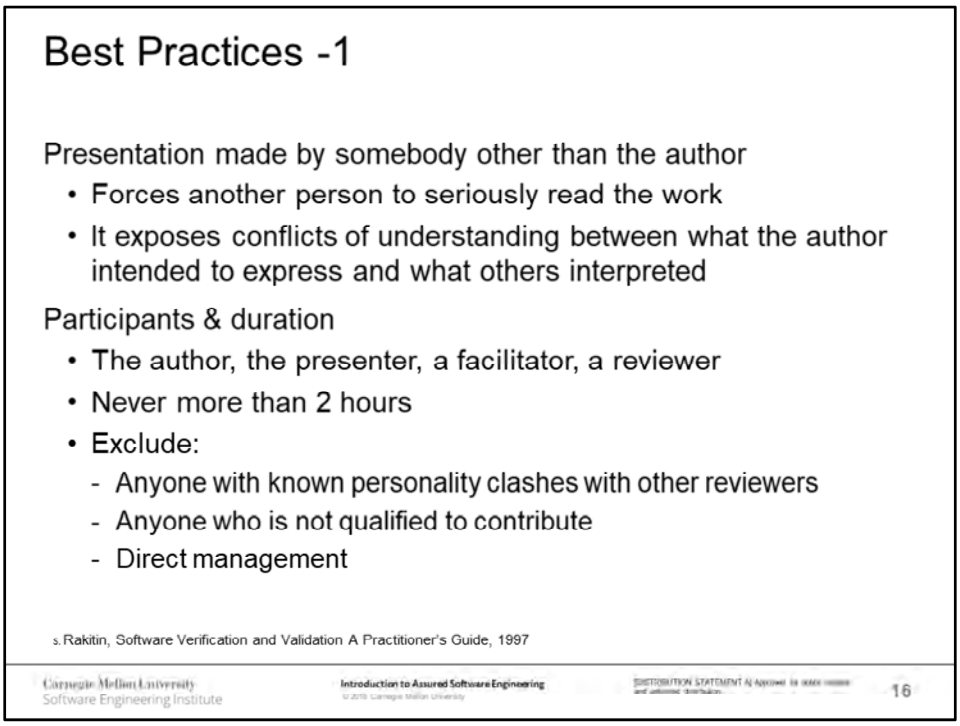


The benefits of inspecting software products have long been established, not only with regards to improving the quality of the artifacts but also as a vehicle for learning and communicating within the development team. Still despite its benefits inspections are not as widely practice as they should be.

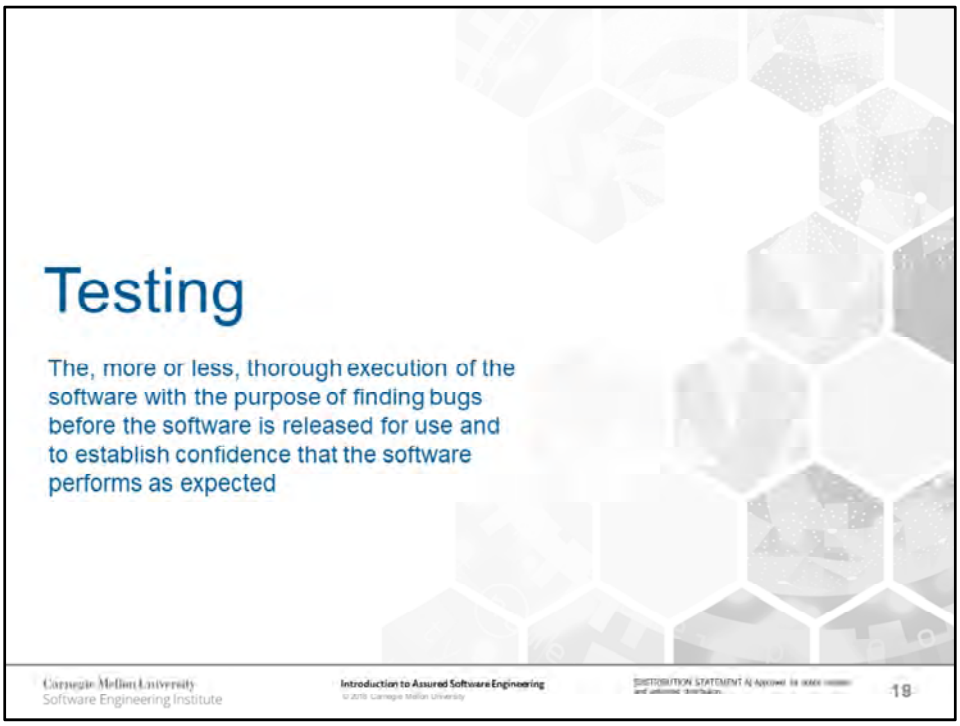
In my experience the reason why they are not more widespread is because they are like diets: they are not all that much fun and they require a lot of discipline.



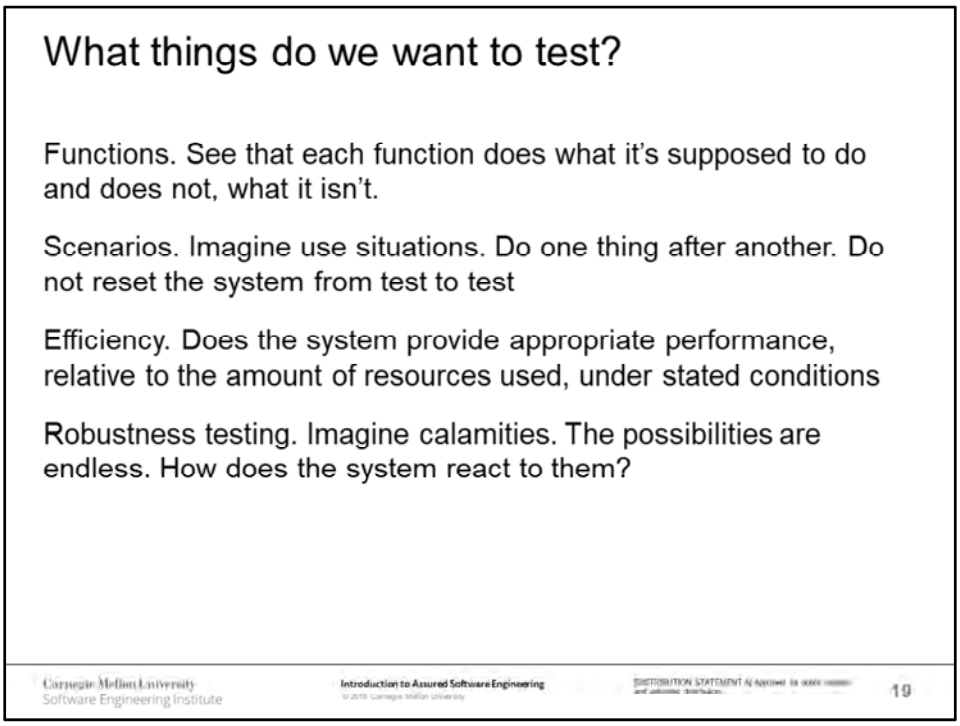
There are different types of inspections: They differ in their purpose, the artifacts inspected, the degree of formality (steps to follow and roles) required by the process, the existence (or not) of prescribed checklists



So what are the best practices in terms of inspections. Prevent people from showing off or settling scores by attacking the work of others



While inspecting consisted in reading design documentation and source code, testing requires the execution of the software under controlled conditions.



Functions. See that each function does what it's supposed to do and not what it isn't supposed to do

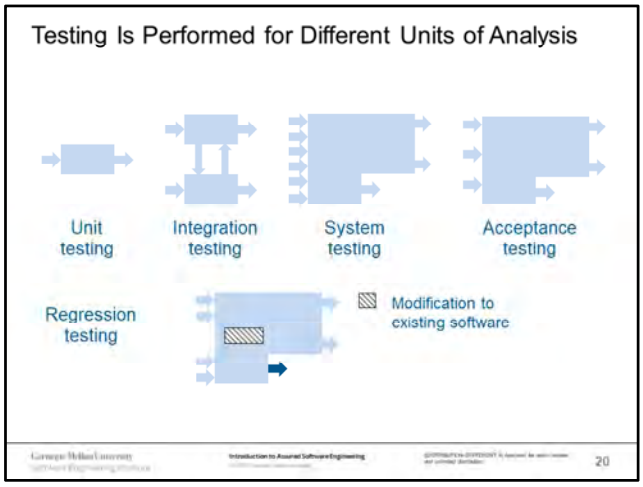
- Look for any data processed by the product. Look at outputs as well as inputs
- Decide which particular data to test with. Consider things like boundary values, typical values, convenient values, and invalid values
- Consider combinations of data worth testing together

Scenarios. Test to a compelling story. Do one thing after another

- Define test procedures or high level cases that incorporate multiple activities connected end-to-end
- Don't reset the system between tests.
- Vary timing and ordering of events

Efficiency. Does the system provide appropriate performance, relative to the amount of resources used, under stated conditions

- Performance testing. The testing to evaluate system's response time, throughput and resource utilization
- Load testing. Process of exercising the system by feeding it the largest specified task or workload
- Stress testing. Trying to break the system with the purpose of assuring that the system fails and recovers gracefully. This testing is performed by overwhelming the system's resources or by taking them away from it beyond the specified conditions
- Robustness Testing. Imagine calamities. The possibilities are endless. How will the system react to them?



Software is developed in units that are later assembled. Accordingly we can distinguish different levels of testing.

- Unit Testing - A unit is the "smallest" piece of software that a developer creates. It is typically the work of one programmer and is stored in a single file. Different programming languages have different units: In C++ and Java the unit is the class; in C the unit is the function; in less structured languages like Basic and COBOL the unit may be the entire program.
- Integration Testing - In integration we assemble units together into subsystems and finally into systems. It is possible for units to function perfectly in isolation but to fail when integrated. For example because they share an area of the computer memory or because the order of invocation of the different methods is not the one anticipated by the different programmers or because there is a mismatch in the data types. Etc.
- System Testing - A system consists of all of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer. System testing focuses on defects that arise at this highest level of integration. Typically system testing includes many types of testing: functionality, usability, security, internationalization and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more.
- Acceptance Testing - Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software and giving us their money. From the customer's point of view, they would generally like the most exhaustive acceptance testing possible (equivalent to the level of system testing). From the vendor's point of view, we would generally like the minimum level of testing possible that would result in money changing hands.

Typical strategic questions that should be addressed before acceptance testing are: Who defines the level of the acceptance testing? Who creates the test scripts? Who executes the tests? What is the pass/fail criteria for the acceptance test? When and how do we get paid?

The purpose of regression testing is to verify that the introduction of new functionality or fixes to the software does not affect things that should not be affected. It consists of the execution of the software, using previously passed test cases, while looking for differences in the results.

2. Re-executes some or all existing test cases to exercise code that was tested in a previous release or previous test cycle.

3. Performed when previously tested code has been re-linked such as when:

- Ported to a new operating system
- A fix has been made to a specific part of the code.
- A fix has been made to another part of the code, but this module had to be re-linked because the fix was in an underlying library that this module also uses.

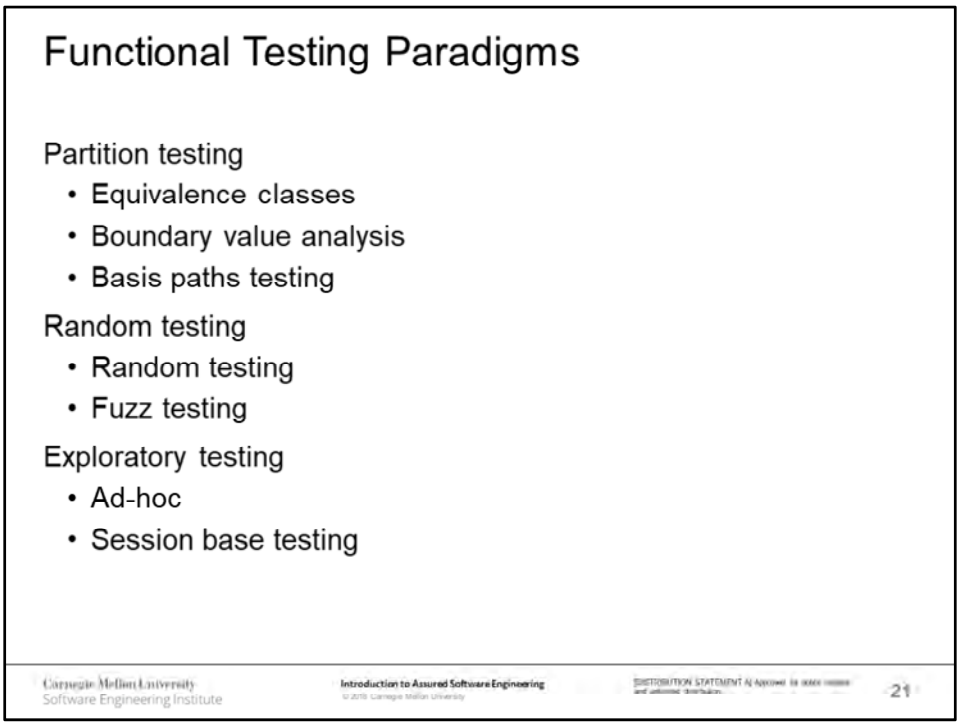
4. Chances are defects may not be fixed correctly or the code change may introduce new defects. Studies show that:

- The probability of changing the program correctly on the first try is only 50% if the change involves 10 or fewer lines of code.
- The probability of changing the program correctly on the first try is only 20% if the change involves around 50 lines of code.

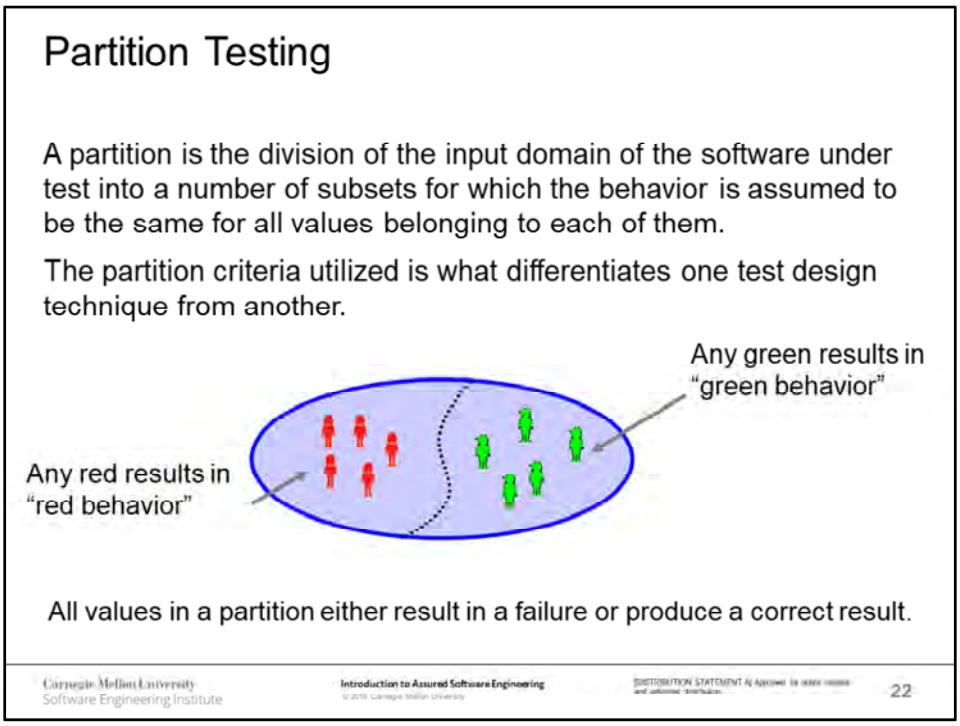
Not all systems are amenable to using these levels. These levels assume that there is a significant period of time between developing units and integrating them into subsystems and then into systems. In Web development it is often possible to go from concept to code to production in a matter of hours. In that case, the unit-integration-system levels don't make much sense. Many Web testers use an alternate set of

levels:

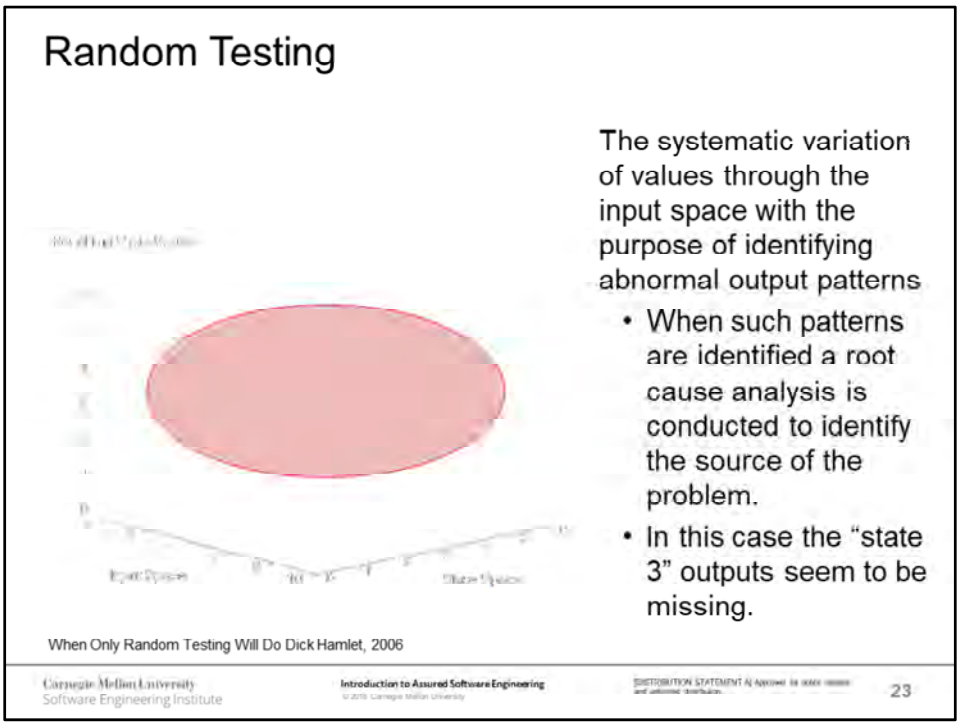
- Code quality
- Functionality
- Usability
- Performance
- Security



All testing methods belong to one of the following categories

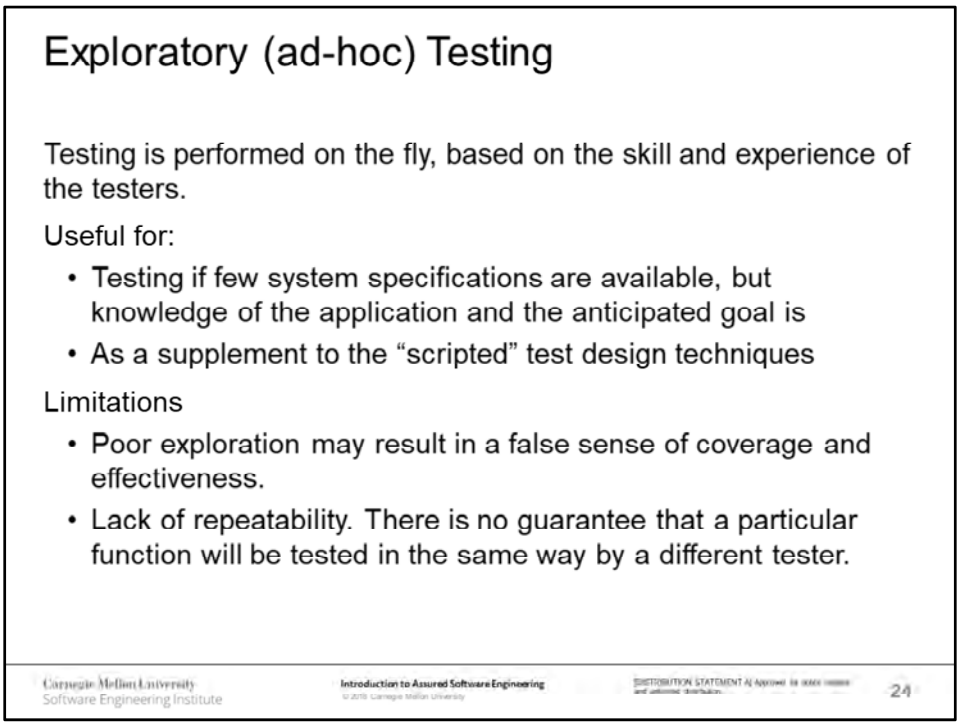


To keep down our testing costs, we don't want to write several test cases that test the same aspect of our program. A good test case uncovers a different class of errors (e.g., incorrect processing of all character data). Equivalence partitioning is a strategy that can be used to reduce the number of test cases that need to be developed. Equivalence partitioning divides the input domain of a program into classes. For each of these equivalence classes, it is hypothesized that the set of data must be treated the same by the module under test and should then produce likely answers, if it does not then the test fails.

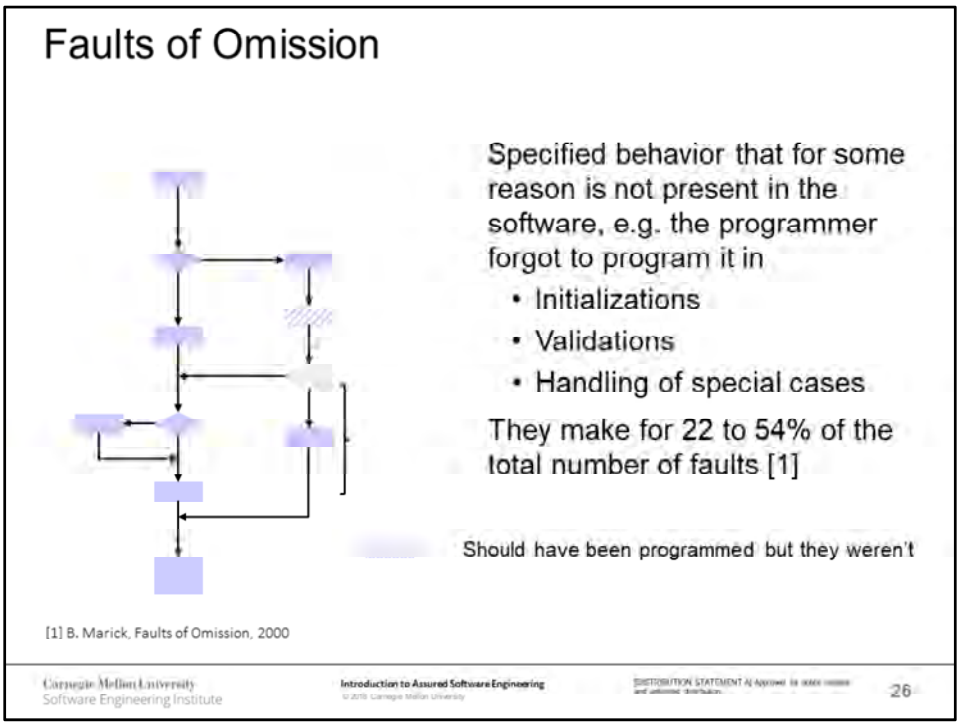


Random testing is a technique which systematically explores the input space of the software under testing. This use of the term random is very different from other disciplines where random testing means selecting a few cases just by "chance". To be effective and efficient random testing relies on the automatic generation of test inputs. The problem with random testing is how to verify that the results are those expected. This is known as the "oracle problem".

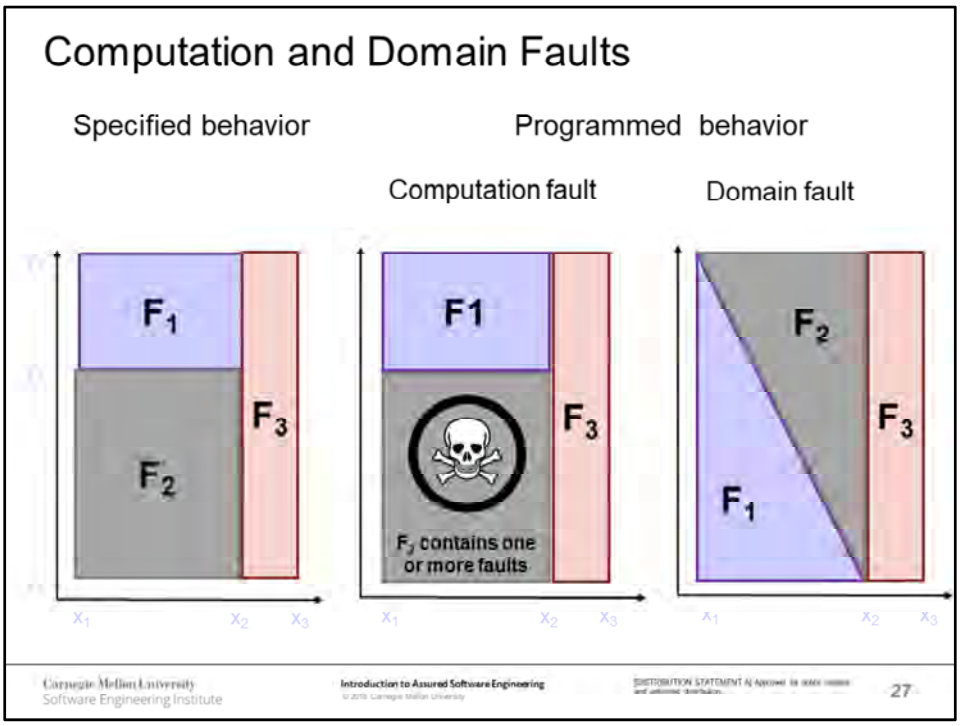
The solutions to the oracle problem fall into three categories: the use of a proxy to produce the correct results against the values produced by the new application will be checked. The proxy could be an existing system or a simple computational form. The second category is based on the recognition of patterns. Pattern recognition can take the form of curve fitting algorithms followed by a study of discontinuities in the output of the application or it can be in the form of assertions where a relation between variables is a specified and a violation of the assertion denotes a failure of the software. The breaking of a pattern can also be detected by visual inspection of a summary input like in the example shown. A third category consist in analyzing the states which result after the execution of the software with a certain data. A common example for this is fuzz testing in which the expected output from the test are that after processing the data the software exits normally. The software raises an exception indication an abnormal condition or the software crashes revealing an abnormal situation that it was not programmed to handle.



Important as a complement to other forms of testing



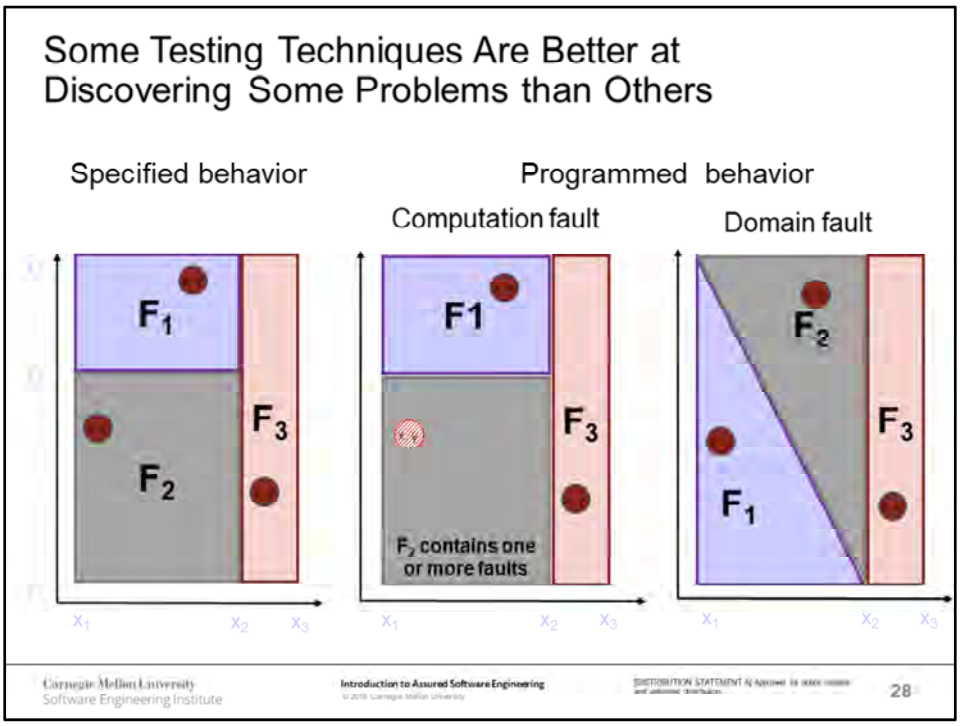
Faults of omission, as its name indicates, are things that were specified or at least expected and were left out.



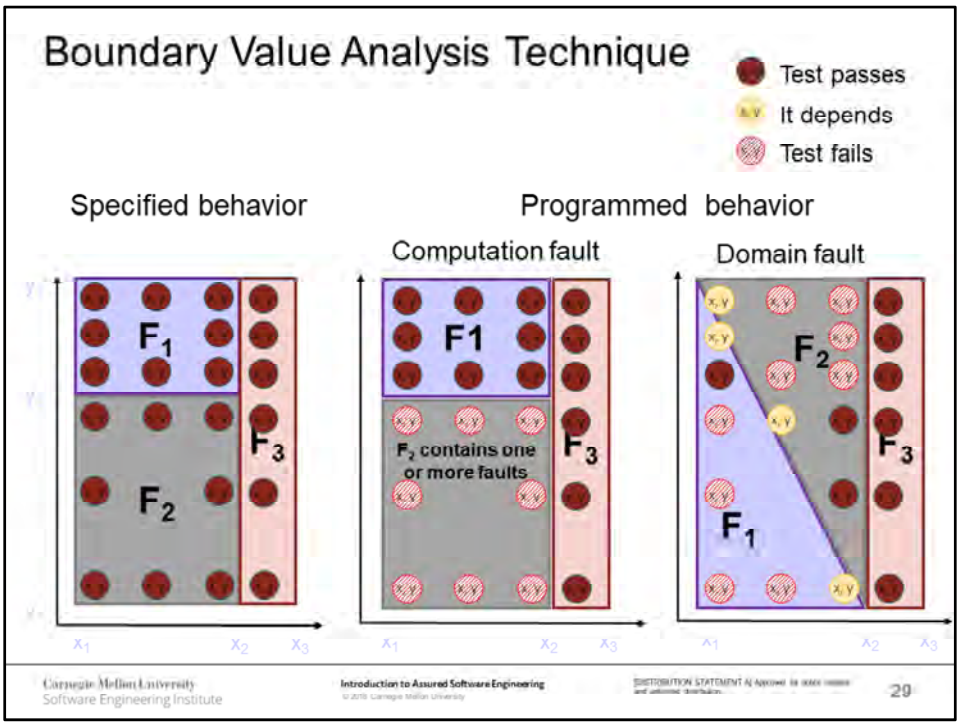
Faults can classified into computation faults and domain faults.

(1) Computation faults: The function containing an implementation fault is applied to the right data

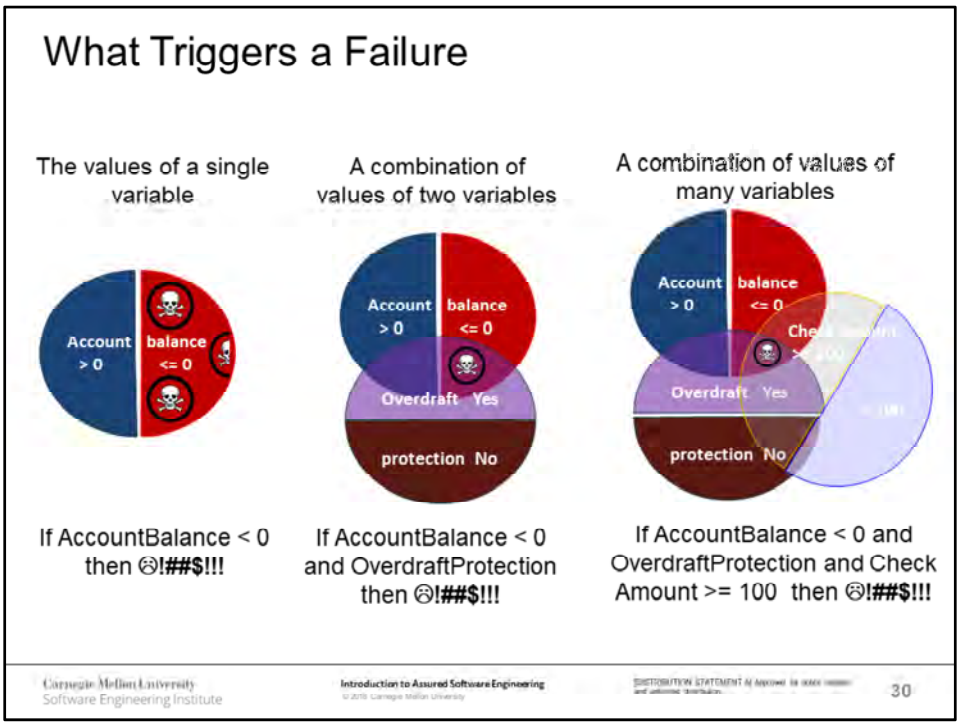
(2) Domain faults: The correct implementation of a function is applied to the wrong data



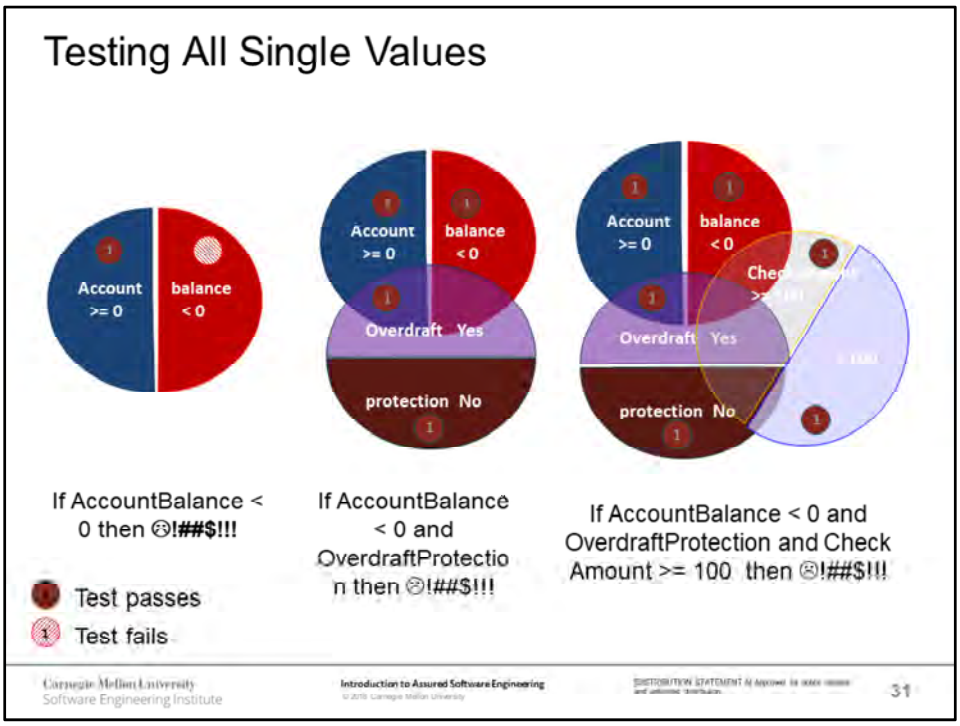
Let's say we test the different partitions with a single value per partition. The test suite will uncover the computation fault but miss the boundary shift.



BVA will find both the computation and the domain fault at the expense of a greater number of test cases.



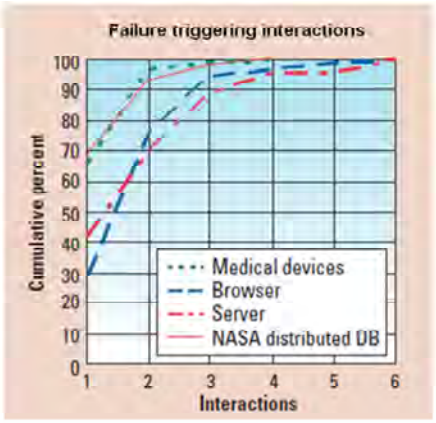
Imagine a program with have the three faults shown. In the first case any Account Balance < 0 will trigger the fault. The second fault will be only triggered if we execute the program with data where an individual has a negative balance and has Overdraft Protection. If this condition is not met during testing the fault will remain latent until the day the software is deployed and a customer that signed for Overdraft Protection gets withdraws more money than he deposits. In the last case three conditions need to be met for the failure to be triggered



In this case we have designed the test cases in such a way that all values of each variable are covered at least once. To do this we need two test cases for example: one whose values are Account Balance >=0, Overdraft Protection = Yes and Check Amount >=100 and another with Account Balance < 0, Overdraft Protection = No and Check Amount <100

These test cases will cover all equivalent values of all the variables involved but will find only one of the faults

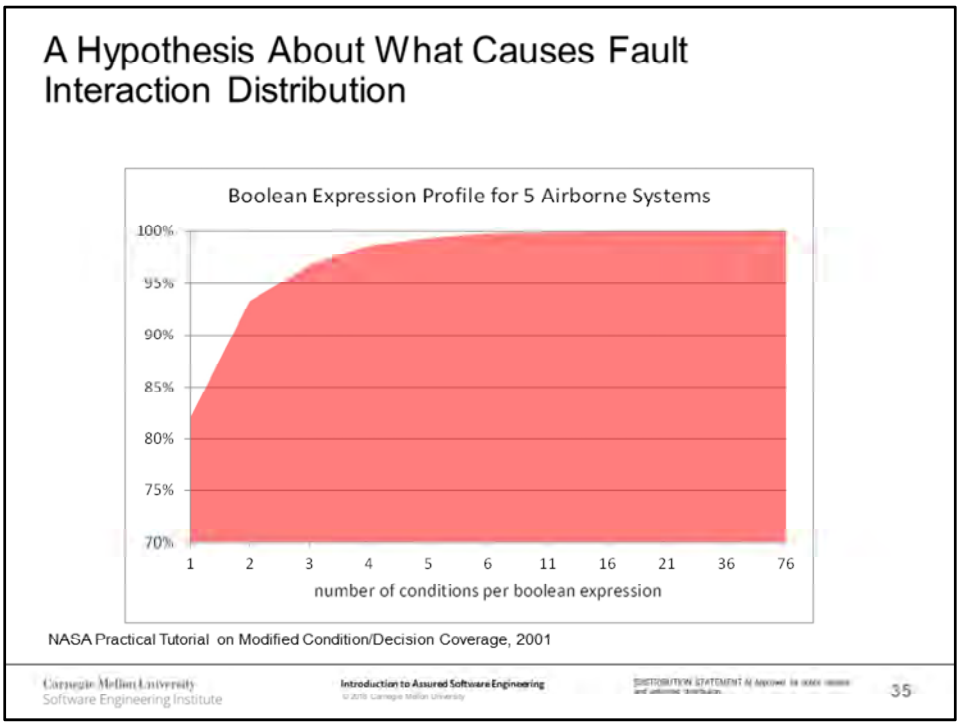
Do we need to test for all combinations of values?



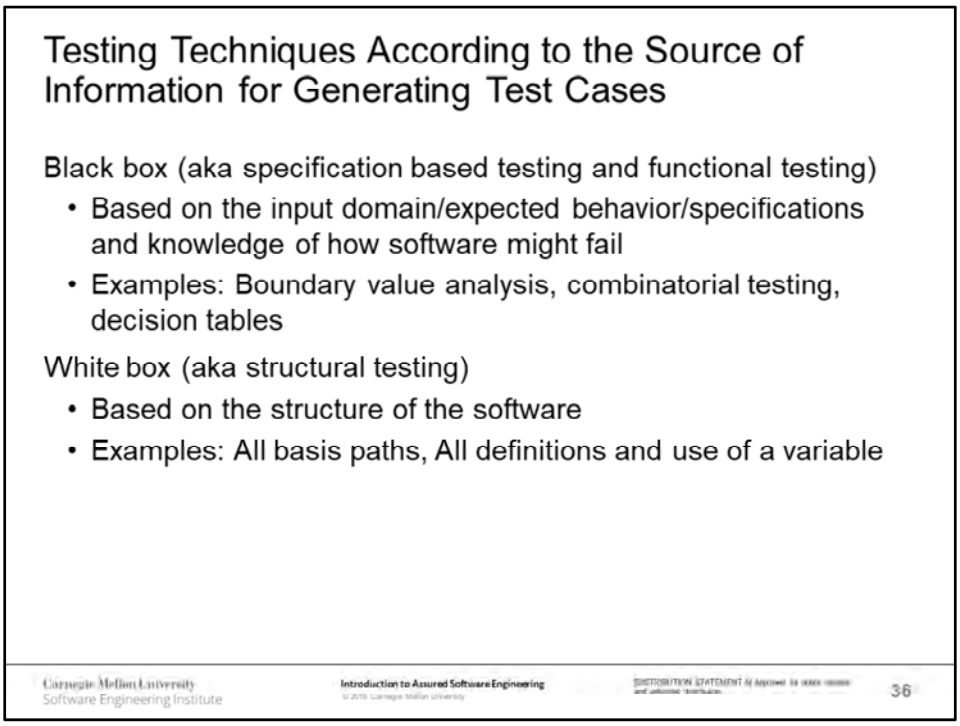
Combinatorial testing is based on empirical findings, there is no underlying "software physics". So while testing for 4 or 5 interactions is economically effective, and probably more thorough than what many organizations do today, there is no guarantee that it will find all faults.

Practical Combinatorial Testing: Beyond Pairwise, R. Kuhn, Yu Lei, R. Kacker, 2008

We all have been taught that exhaustive testing is impossible, but is it necessary?



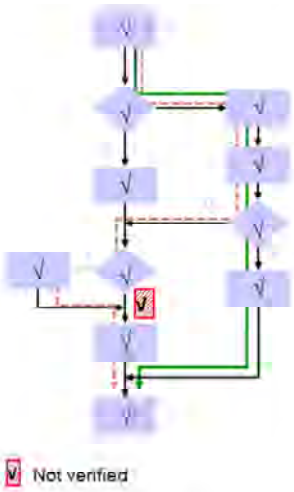
Miller, George A., 1956 – "The Magical Number Seven, Plus or Minus Two, Some Limits on Our Capacity for Processing Information"



Testing techniques are also classified according to the source of information from which the test cases are derived.

All systematic testing techniques (except random testing), are based on partitioning the input domain of the SUT. These techniques aim to produce partitions that exercise a particular feature of the SUT, such as branches in the code, or boundary values as described by the SUT specification. These techniques are based, in part, on the assumption that all inputs in a partition will be treated similarly by the SUT. That is, if one input value from a partition causes a SUT to fail, then it is assumed that the remaining inputs from the partition will also cause the SUT to fail, and vice versa for inputs that do not cause failure. If this is the case, then the partitions are considered to be homogeneous, or truly revealing. White box testing establish equivalence vis a vis the control or dataflow structure of the software. Notice that the criteria for designing test cases correspond to the adequacy criteria. That is the criteria can be used to measure the extent of testing as well as to generate test cases that satisfy it.

White Box Testing: Looking at the Code



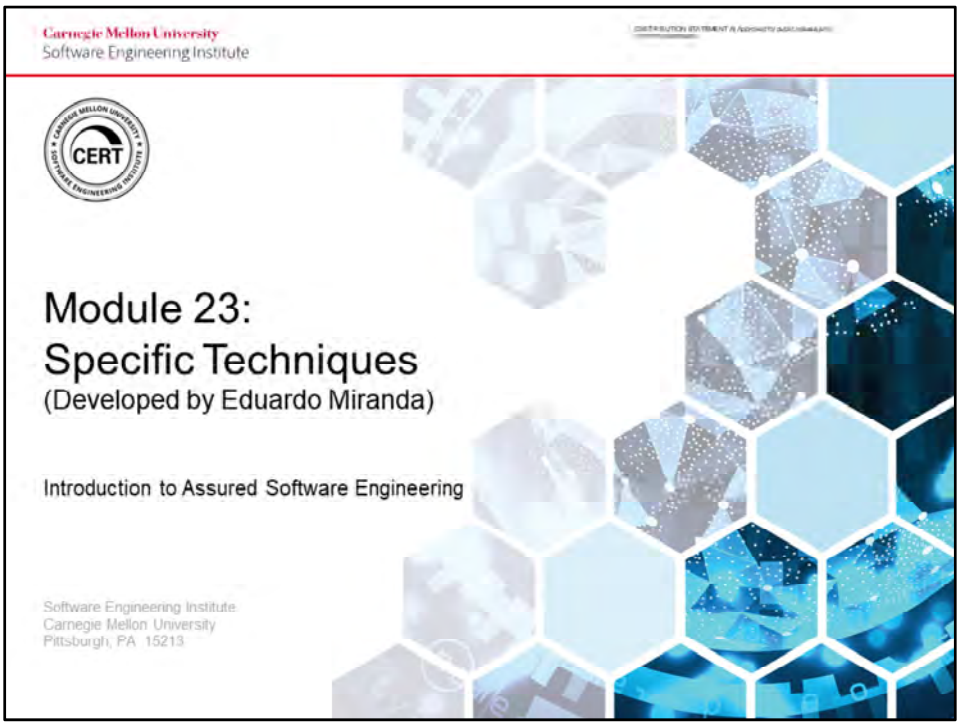
Verify that all tests are passed and:

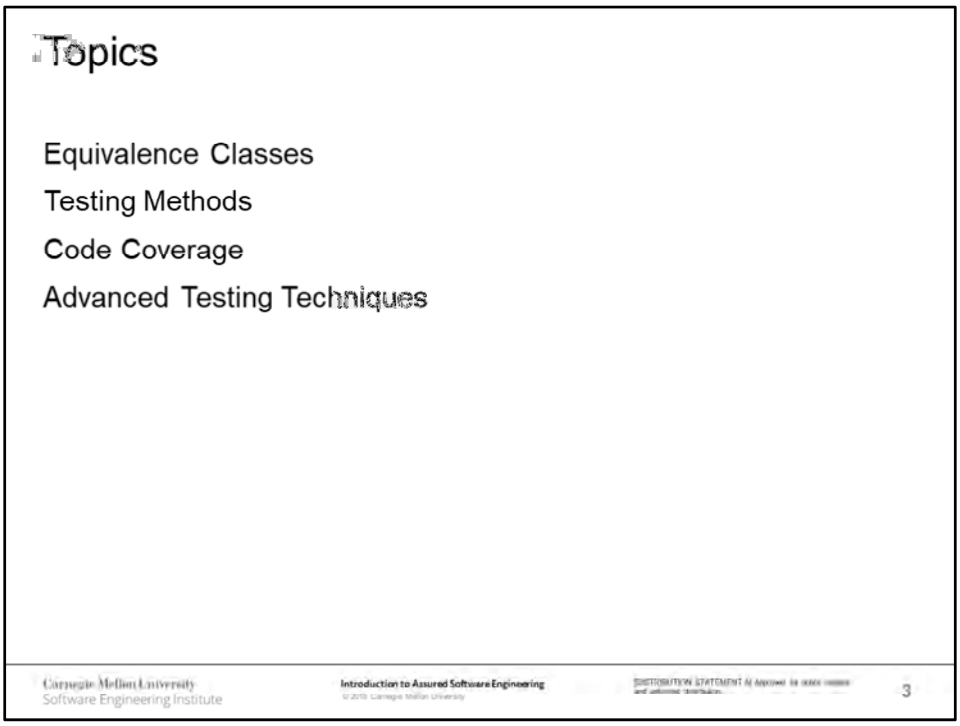
- All the code statements have been executed (statement coverage)
- Both branches of each condition have been executed (branch coverage)
- All paths (impossible in most but the simplest situations) have been traversed (path coverage)

The fact that all statements and all branches have been executed does not guarantee that the software is fault free since the coverage criteria can be met and still there might be some faults that were not triggered by the data.

To stimulate the program we need inputs, how do we choose them? If we base our selection we might miss unspecified behaviors implemented in the program. If we select values based on the structure we might miss required behaviors that were not implemented.

If the condition is not satisfied we will analyze the code to see what values must be provided as input to force the execution along the desired path


Carnegie Mellon University
Software Engineering Institute
CERT
Module 23:
Specific Techniques
(Developed by Eduardo Miranda)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Instructor note: This material is sophisticated and may be beyond what is expected for your students. You may have some advanced students who would appreciate it as an extra lecture or optional assignment.

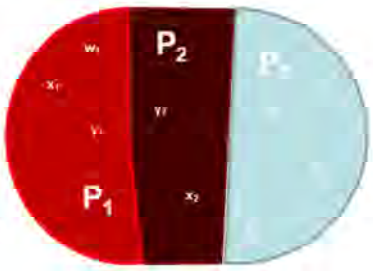
Equivalence Classes

Based on the specification or some other information it is possible to hypothesize that the input domain is made up of a number of partitions (e.g.: P1, P2 and P3) →

1. Values from the same partition exhibit the same behavior not the same result (e.g., they are calculated using the same procedure).
2. Values from different partitions exhibit different behavior.

Two situations

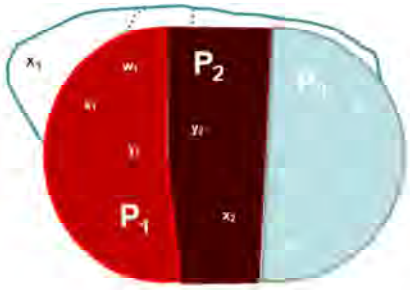
- The members of the partition are defined by enumeration.
- The partition can be defined by comprehension (intention, formula).



The idea is simple. You look at the problem and conclude that certain values ought to be treated the same by the software and as a result:

- If one value catches a bug, the others in the same category probably will too.
- If one test doesn't catch a bug, the others probably won't either.

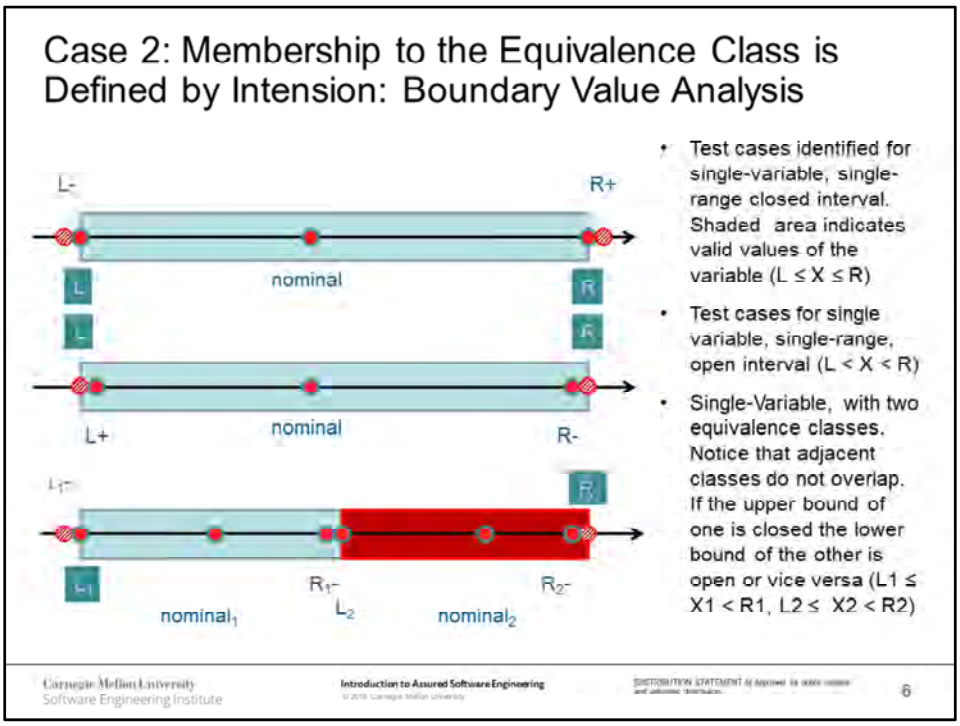
Case 1: Membership to an Equivalence Class Is Defined by Enumeration



Once the equivalence classes have been identified, create two test cases (if possible) for each equivalence class.

- If any or both test cases do not result in the expected values you can discard the equivalence hypothesis.
- Additional test cases for each equivalence class may give us more confidence, but the best we can hope for, without testing all members, is to disprove the equivalence hypothesis.

Many people says is sufficient to test just one value in a class. I prefer to test at least two values (if possible) to have a warmer feeling that the hypothesis might hold. You cannot be sure that it holds for all values without testing for them, but if you test for two and the test cases do not pass I will know that my hypothesis was not correct to start with.



From experience we know that faults tend to accumulate at the borders of an interval, this are typically called “off by one” faults. For example a programmer used a “less than condition” when he should have used a “less or equal” or when it should have started counting something from zero and instead started from one.

How many test cases do we need in order to be reasonably reassured that we have done a comprehensive testing job?

- If we wanted to test all possible combinations of the 34 switches in the panel we would need $2^{34} = 1.7 \times 10^{10}$ test cases.
- What if we suspected that all faults involved only 3-way interactions among the 34 switches? In this case we could do it with only 33 tests.
- What if we were 99% certain that all faults involved at most 4 interactions? In this case we could do the job with only 85 tests.

Adapted from Combinatorial Methods for Cybersecurity Testing, Rick Kuhn and Raghu Kacker, National Institute of Standards and Technology, 2009

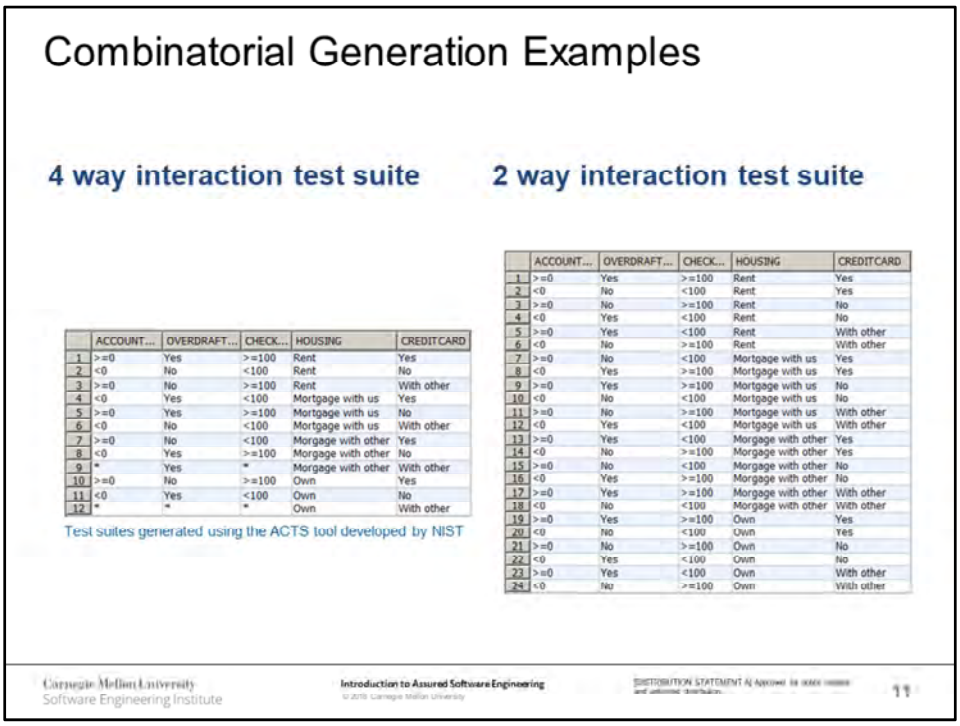
Carnegie Mellon University Software Engineering Institute | Introduction to Assured Software Engineering | 8

30,000 years of continued work assuming one minute per test

In general, the number of tests required for *t-way combinatorial testing of n parameters with v values apiece is proportional to* v^t *log n. So, the* number of tests needed for four-way testing is several times that required for three-way testing.

On the other hand, testing 30 parameters requires a modest increase over the number of tests needed for 20. For example, a system with 20 variables, five values each, requires 444 tests for three-way coverage but 3,019 tests for four-way coverage with IPOG. A much smaller penalty is incurred for covering more variables:

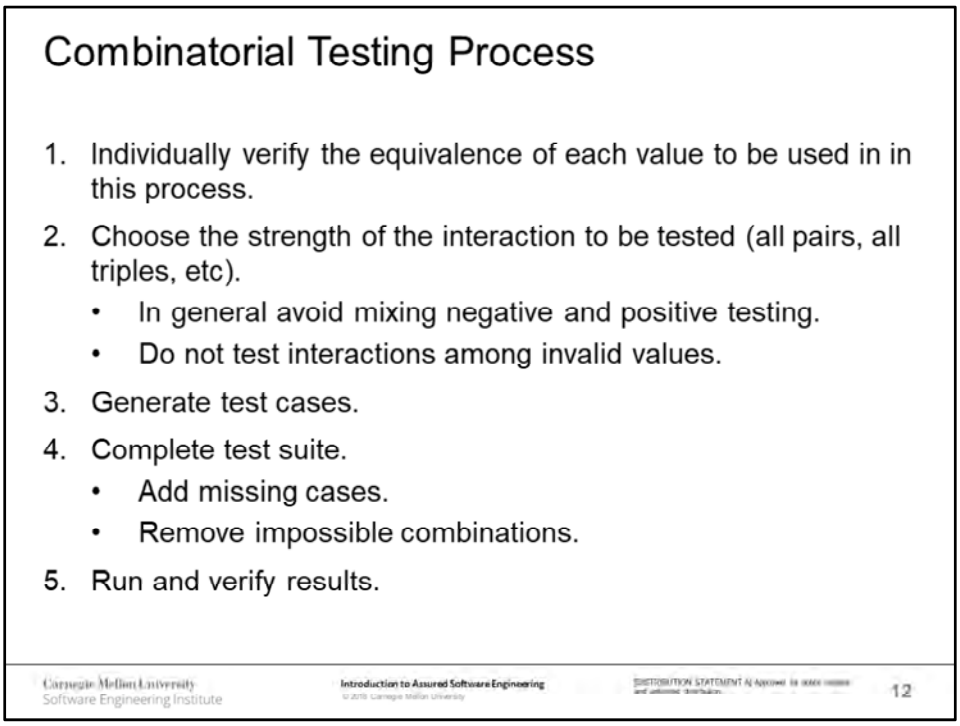
- increasing the number of variables to 30 requires 3,749 tests for four-way coverage, a 24 percent increase.



	ACCOUNT...	OVERDRAFT...	CHECK...	HOUSING	CREDITCARD
1	>=0	Yes	>=100	Rent	Yes
2	<0	No	<100	Rent	No
3	>=0	No	>=100	Rent	With other
4	<0	Yes	<100	Mortgage with us	Yes
5	>=0	Yes	>=100	Mortgage with us	No
6	<0	No	<100	Mortgage with us	With other
7	>=0	No	<100	Morgage with other	Yes
8	<0	Yes	>=100	Morgage with other	No
9	*	Yes	*	Morgage with other	With other
10	>=0	No	>=100	Own	Yes
11	<0	Yes	<100	Own	No
12	*	*	*	Own	With other

	ACCOUNT...	OVERDRAFT...	CHECK...	HOUSING	CREDITCARD
1	>=0	Yes	>=100	Rent	Yes
2	<0	No	<100	Rent	Yes
3	>=0	No	>=100	Rent	No
4	<0	Yes	<100	Rent	No
5	>=0	Yes	<100	Rent	With other
6	<0	No	>=100	Rent	With other
7	>=0	No	<100	Mortgage with us	Yes
8	<0	Yes	>=100	Mortgage with us	Yes
9	>=0	Yes	>=100	Mortgage with us	No
10	<0	No	<100	Mortgage with us	No
11	>=0	No	>=100	Mortgage with us	With other
12	<0	Yes	<100	Mortgage with us	With other
13	>=0	Yes	<100	Morgage with other	Yes
14	<0	No	>=100	Morgage with other	Yes
15	>=0	No	<100	Morgage with other	No
16	<0	Yes	>=100	Morgage with other	No
17	>=0	Yes	>=100	Morgage with other	With other
18	<0	No	<100	Morgage with other	With other
19	>=0	Yes	>=100	Own	Yes
20	<0	No	<100	Own	Yes
21	>=0	No	>=100	Own	No
22	<0	Yes	<100	Own	No
23	>=0	Yes	<100	Own	With other
24	<0	No	>=100	Own	With other

The use of a tool is necessary to generate test cases.



Test for equivalence classes first. If using BVA, test only for boundary values - do not include the nominal case as we will test for this in the next step.

Choose the strength of the interaction and create a test description containing only one nominal value for each equivalence class.

Generate test suite (automated step).

Complete test suite (manual or automated depending on the tool used).

Add missing “special cases”.

Remove impossible combinations by replacing the conflicting values with feasible ones not to affect the coverage required.

Unless required by the problem do not:

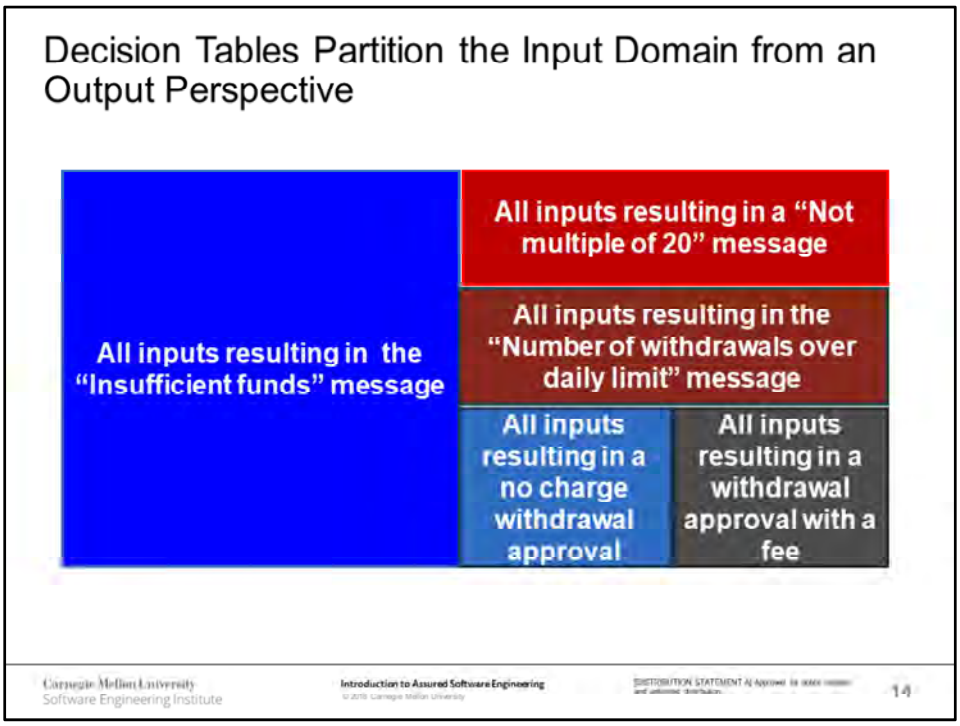
- Mix negative and positive testing
- Do not test interactions among invalid values

Decision Tables

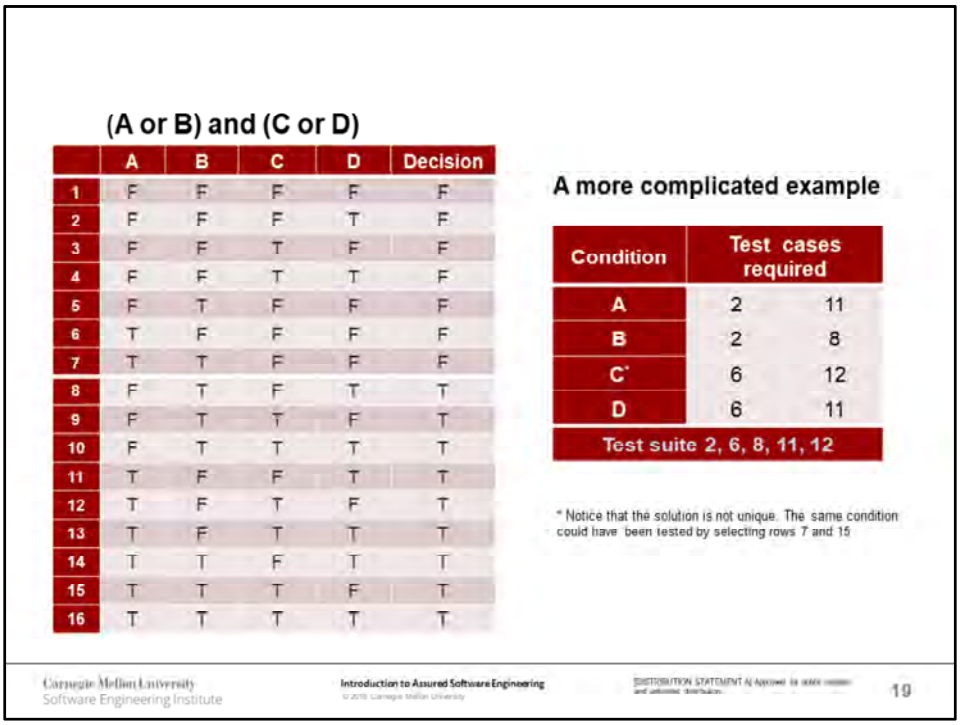
ATM withdraw decision logic

Conditions (Inputs)	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Requested amount <= Balance	No	Yes	Yes	Yes	Yes
Requested amount divisible by 20	*	No	Yes	Yes	Yes
Number of withdrawals	*	*	<= Free limit	<= Paid limit	> Paid limit
Actions (Expected results)					
Approve withdraw			X	X	
Insufficient funds message	X				
Not multiple of 20 message		X			
Charge withdrawal fee				X	
Number of withdrawals over daily limit message					X

The elements of a decision table are conditions, actions and rules. Each combination of conditions that result in a particular output or action is called a rule. Asterisks, dashes or white spaces in a rule are use to denote “Don’t matter values”.



We partition the input domain according to the expected results (actions on the table) and then we proceed as in the case of equivalence classes, generating at least one test case for each rule in the table, and one test case for each "don't matter conditions" value to verify that they really don't matter.



(A or B) and (C or D)

	A	B	C	D	Decision
1	F	F	F	F	F
2	F	F	F	T	F
3	F	F	T	F	F
4	F	F	T	T	F
5	F	T	F	F	F
6	T	F	F	F	F
7	T	T	F	F	F
8	F	T	F	T	T
9	F	T	T	F	T
10	F	T	T	T	T
11	T	F	F	T	T
12	T	F	T	F	T
13	T	F	T	T	T
14	T	T	F	T	T
15	T	T	T	F	T
16	T	T	T	T	T

This approach ensures that the effect of each condition is tested relative to the other conditions without requiring analysis of the logic of each decision (that is, if changing the value of a single condition causes the value of the decision outcome to change, then the single condition is assumed to be the cause for the change—no further analysis is needed).

Number of test cases growths linearly with number of conditions (n+1) instead of 2n required by multiple condition coverage.

Disadvantage is very laborious.

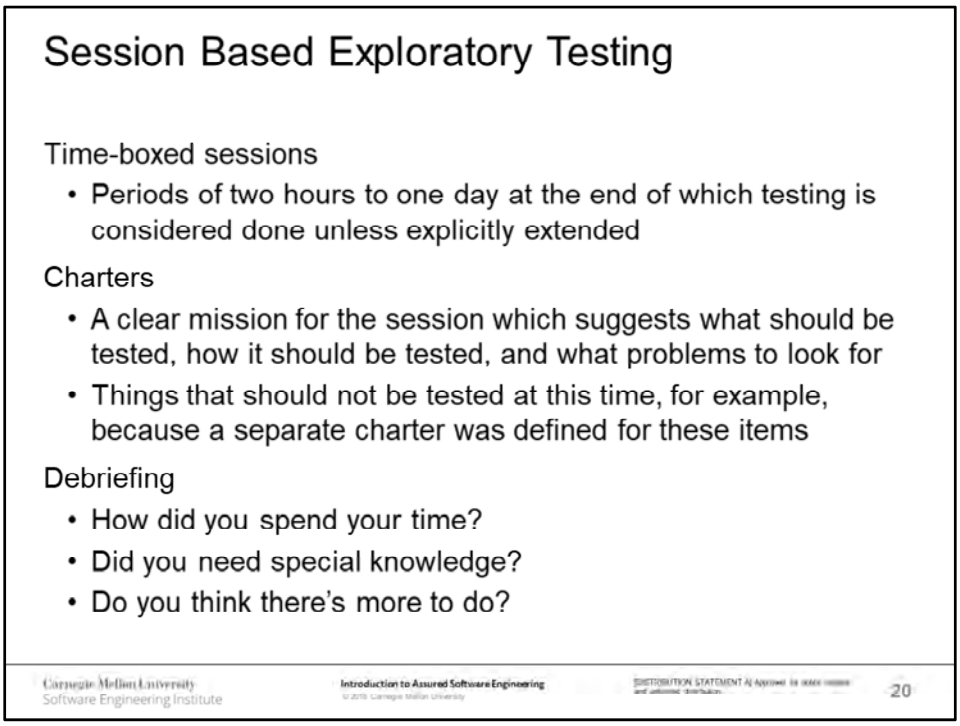
1) Choose a condition, find a pair of rows that differ in the decision outcome and in the value of the chosen condition alone (all the rest must remain the same)

Showing that a condition independently affects a decision's outcome by varying just that condition while holding all others fixed is commonly referred to as the unique-cause approach to MC/DC. This approach ensures that the effect of each condition is tested relative to the other conditions without requiring analysis of the logic of each decision (that is, if changing the value of a single condition causes the value of the decision outcome to change, then the single condition is assumed to be the cause for the change—no further analysis is needed).

Historically, the unique-cause approach has often been the only acceptable means of showing the independent effect of a condition. The unique-cause approach cannot be applied, however, to decisions where there are repeated or strongly coupled conditions; e.g., (**A *and B) or (A and C).***

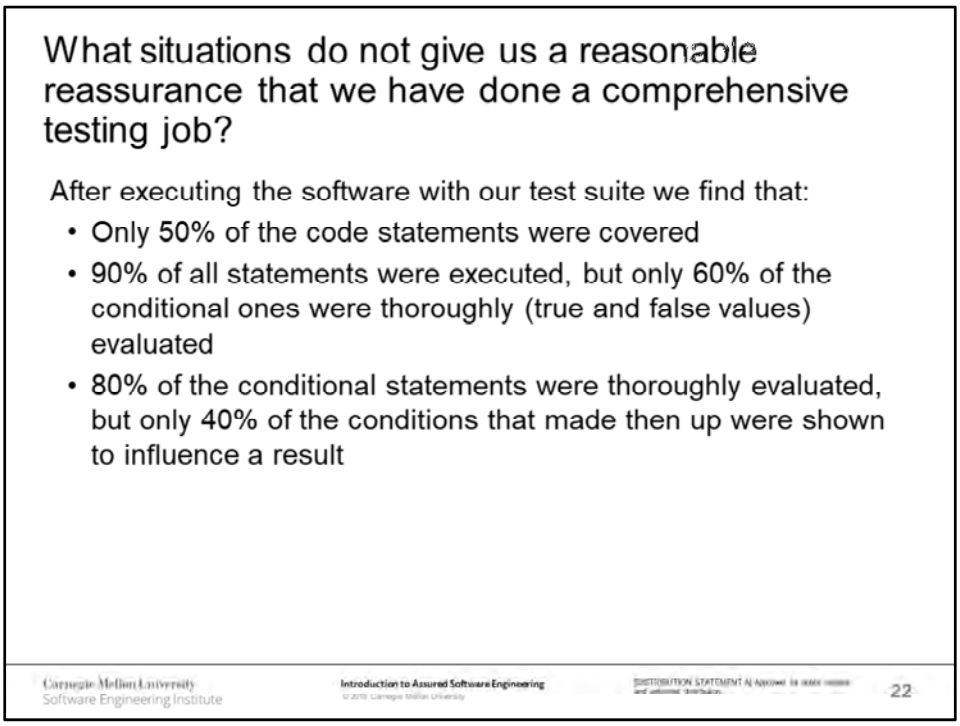
The unique-cause approach commonly is taught by presenting a truth table for an expression; for example, the decision **Z:= (A *or B) and (C or D) shown in Table 3. In the truth table approach, test cases*** that provide MC/DC are selected by identifying pairs of rows where only one condition and the decision

outcome change values between the two rows. In Table 3, the columns shaded in gray indicate the independence pairs for each condition. For example, test case 2 coupled with test case 10 together demonstrate the independent effect of **A, because A is the only condition that has changed value along** with the change in value of the outcome **Z. Although the truth table is a simple approach to showing the** independent effect of a condition, the truth table approach suffers from a number of limitations: (a) the truth table is unwieldy for large logical expressions; and, for a logical expression with *n inputs, only n+1* of the 2*n rows are useful; (b) the truth table addresses only one logical expression at a time; and, (c) the* truth table does not connect the inputs and outputs from the requirements-based tests with the source code structure.

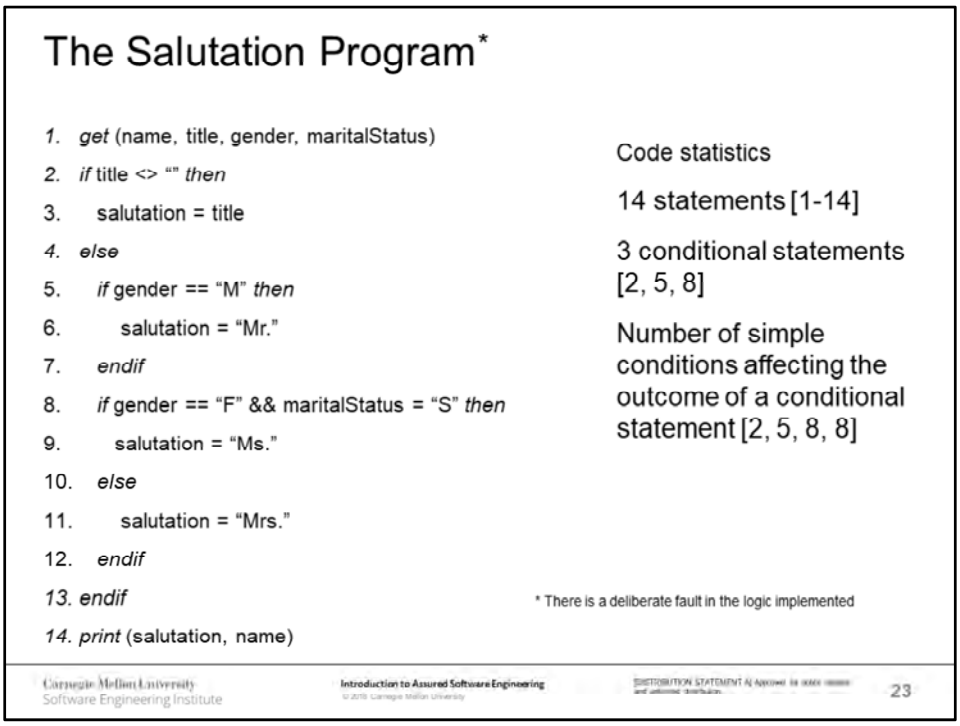


Unbounded exploratory testing tends to result in a waste of resources due to overlap (somebody testing something that was already tested), gaps (nobody checking something) and lack of visibility (how much is left, what has been accomplished). Exploratory Session-based testing is a method for managing testing effort by compartmentalizing testing activity into time-boxed called sessions.

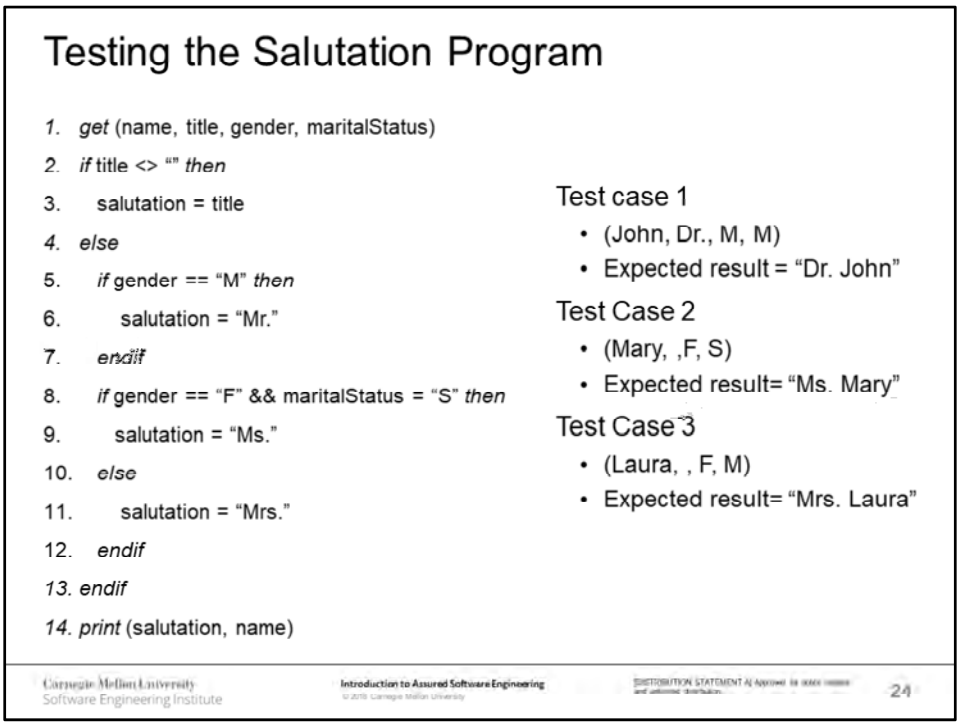
A session is governed by a charter, or a mission statement consisting of a paragraph or two to guide the tester on what to do in the session. It suggests what to be on the lookout for, what tools to use, and what areas of the product to cover.



Earlier in this presentation I asked the question: “How many test cases do we need to be reasonably reassured that we have done a comprehensive testing job?” This is a very difficult question to answer in the positive.



This program asks for the name, title, gender and marital status of a person and produces as output the appropriate salutation. The program contains a deliberate fault in its logic. The salutation for every man without a title will be "Mrs".



Execute the program with the following test cases.

100% Coverage Does Not Imply All Faults Have Been Exposed

Intended behavior

*If A **or** B then*

Z = Z + 1

else

Z = Z + 2

endif

Programmed behavior

*If A **and** B then*

Z = Z + 1

else

Z = Z + 2

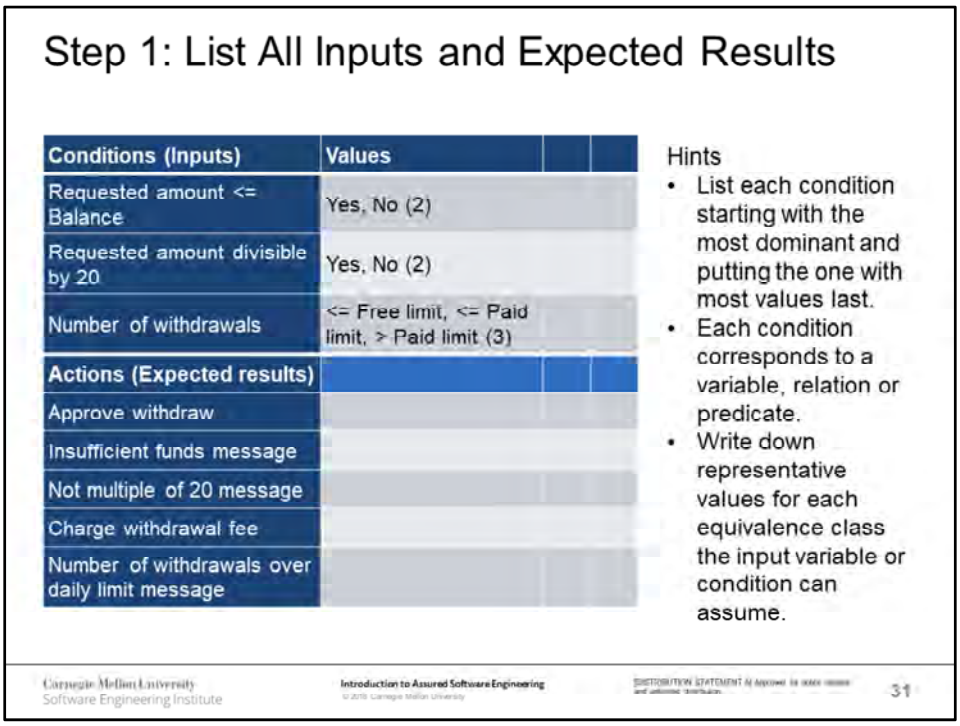
endif

Test suite

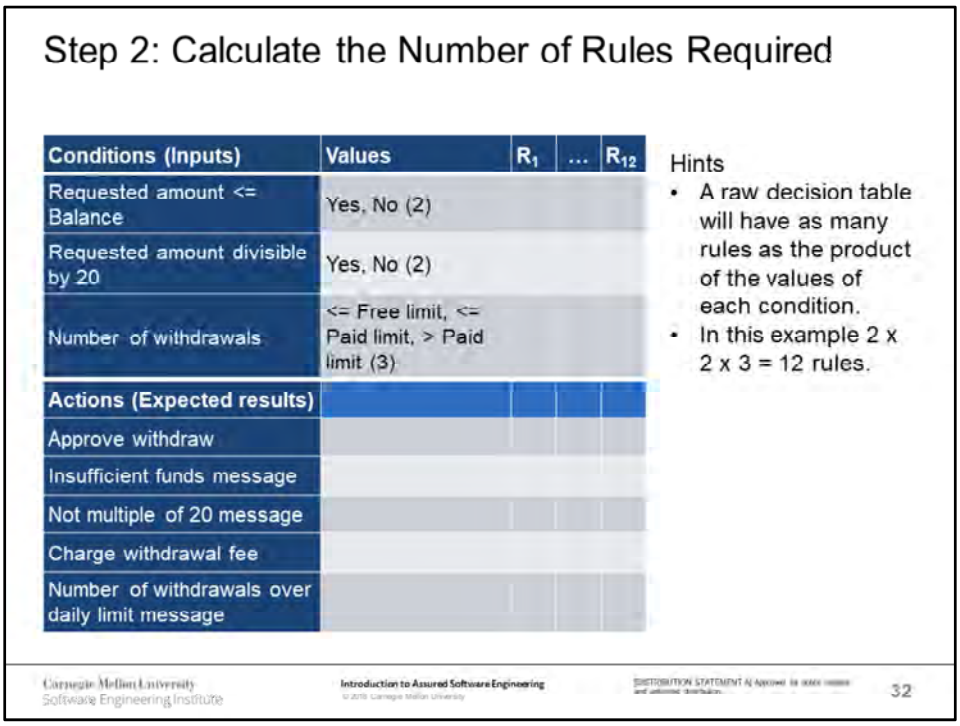
- A = True, B = True
- A = False, B = False

100% branch coverage without exposing the fault

|| = or



Decision table test design is a very useful technique.



Decision table test design is a very useful technique.

Step 3: Fill Columns With All Possible Combinations

Hints

- Write down each value of each condition in repeating sequences of length k = combinations left / by the number of values of the row's input.
- In this example:
  1st row, k=12/2 = 6; 2nd row, k=6/2= 3, 3rd row, k=3/3=1

Conditions (Inputs)	Values	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}
Requested amount <= Balance	Yes, No (2)	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Requested amount divisible by 20	Yes, No (2)	No	No	No	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes
Number of withdrawals	<= Free limit, <= Paid limit, > Paid limit (3)	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL
Actions (Expected results)													

Decision table test design is a very useful technique.

Step 4: Specify Expected Outputs for Each Combination

Hints

- Verify that there are no unmarked action rows (no rule triggers the associated action).
- Verify that the are no unmarked action columns (the rule does not trigger any action).

Conditions (Inputs)	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}
Requested amount <= Balance	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Requested amount divisible by 20	No	No	No	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes
Number of withdrawals	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL
Actions (Expected results)												
Approve withdraw										X	X	
Insufficient funds message	X	X	X	X	X	X						
Not multiple of 20 message							X	X	X			
Charge withdrawal fee											X	
Number of withdrawals over daily limit message												X

Decision table test design is a very useful technique.

Step 5: Reduce Test Combinations by Identifying Common Actions

Hint

- The "Insufficient funds" message is not influenced by the divisibility of the amount by 20 nor by the number of withdrawals. R_1 to R_6 can be consolidated into a single rule with "don't matter values".

Conditions (Inputs)	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}
Requested amount <= Balance	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Requested amount divisible by 20	No	No	No	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes
Number of withdrawals	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL	<= FL	<= PL	> PL
Actions (Expected results)												
Approve withdraw										X	X	
Insufficient funds message	X	X	X	X	X	X						
Not multiple of 20 message							X	X	X			
Charge withdrawal fee											X	
Number of withdrawals over daily limit message												X

Carnegie Mellon University Software Engineering Institute — Introduction to Assured Software Engineering — 35

Decision table test design is a very useful technique.

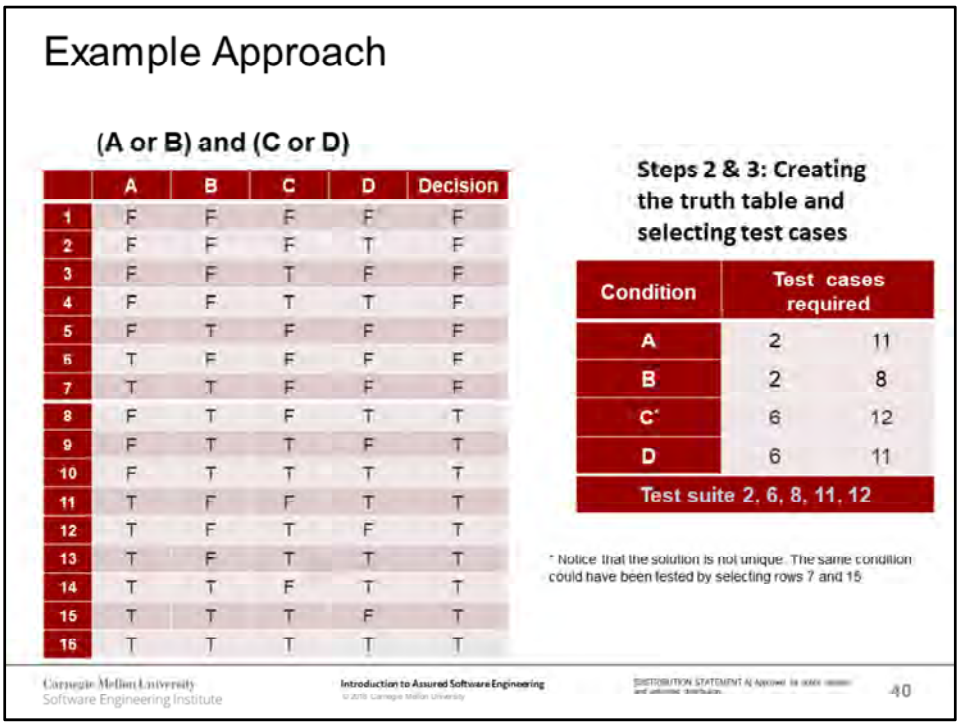
Step 6: Check Covered Combinations

Conditions (Inputs)	Values	R_1	R_2	R_3	R_4	R_5
Requested amount <= Balance	Yes, No (2)	No	Yes	Yes	Yes	Yes
Requested amount divisible by 20	Yes, No (2)	*	No	Yes	Yes	Yes
Number of withdrawals	<= FL, <= PL, > PL (3)	*	*	<= FL	<= PL	> PL
Actions (Expected results)						
Approve withdraw				X	X	
Insufficient funds message		X				
Not multiple of 20 message			X			
Charge withdrawal fee					X	
Number of withdrawals over daily limit message						X
Check sum	**12 =**	**6**	**+ 3**	**+ 1**	**+ 1**	**+ 1**

Hints

- For each column calculate the number of combinations it represents.
- An "*" stands for as many combinations as values the input has.
- The total number of combinations for each column is either 1 or the product of the "*" rows in it.
- Add up the total for each row and and compare with the result of step 2. It must be equal.

Decision table test design is a very useful technique.



	A	B	C	D	Decision
1	F	F	F	F	F
2	F	F	F	T	F
3	F	F	T	F	F
4	F	F	T	T	F
5	F	T	F	F	F
6	T	F	F	F	F
7	T	T	F	F	F
8	F	T	F	T	T
9	F	T	T	F	T
10	F	T	T	T	T
11	T	F	F	T	T
12	T	F	T	F	T
13	T	F	T	T	T
14	T	T	F	T	T
15	T	T	T	F	T
16	T	T	T	T	T

This approach ensures that the effect of each condition is tested relative to the other conditions without requiring analysis of the logic of each decision (that is, if changing the value of a single condition causes the value of the decision outcome to change, then the single condition is assumed to be the cause for the change—no further analysis is needed).

Number of test cases grows linearly with number of conditions (n+1) instead of 2n required by multiple condition coverage.

Disadvantage is that it is very laborious

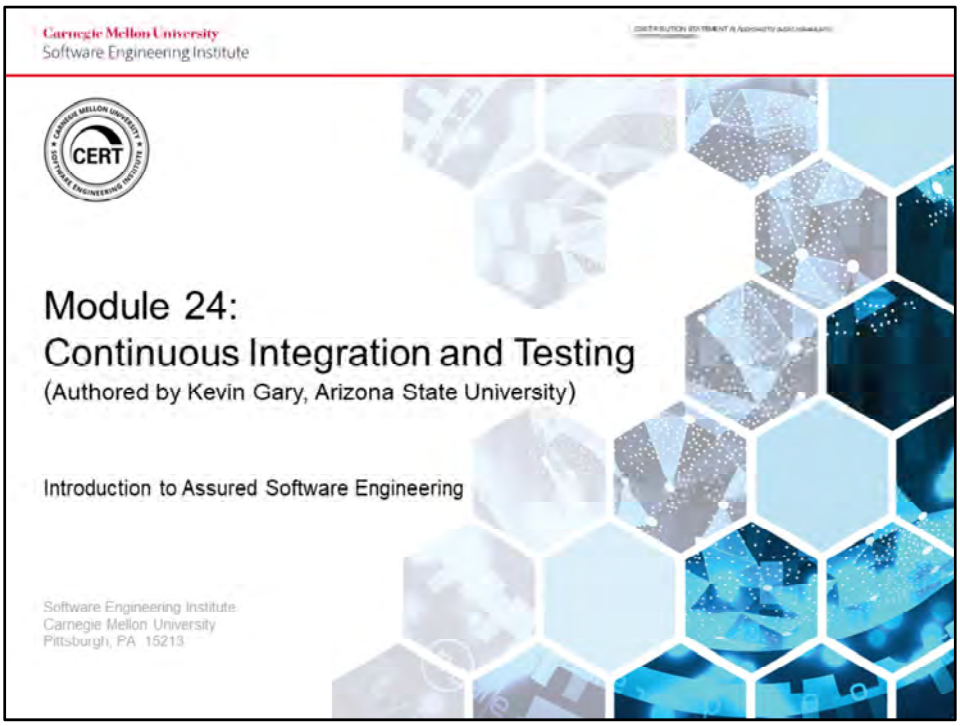
1) Choose a condition, find a pair of rows that differ in the decision outcome and in the value of the chosen condition alone (all the rest must remain the same)

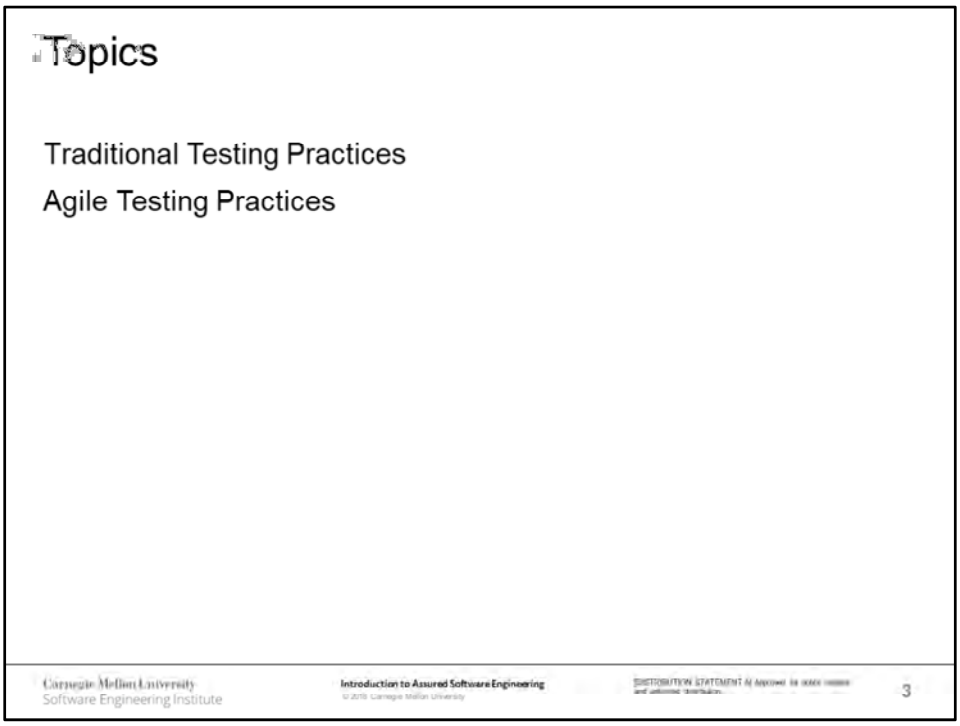
Showing that a condition independently affects a decision's outcome by varying just that condition while holding all others fixed is commonly referred to as the unique-cause approach to MC/DC. This approach ensures that the effect of each condition is tested relative to the other conditions without requiring analysis of the logic of each decision (that is, if changing the value of a single condition causes the value of the decision outcome to change, then the single condition is assumed to be the cause for the change—no further analysis is needed).

Historically, the unique-cause approach has often been the only acceptable means of showing the independent effect of a condition. The unique-cause approach cannot be applied, however, to decisions where there are repeated or strongly coupled conditions; e.g., (**A *and B) or (A and C).***

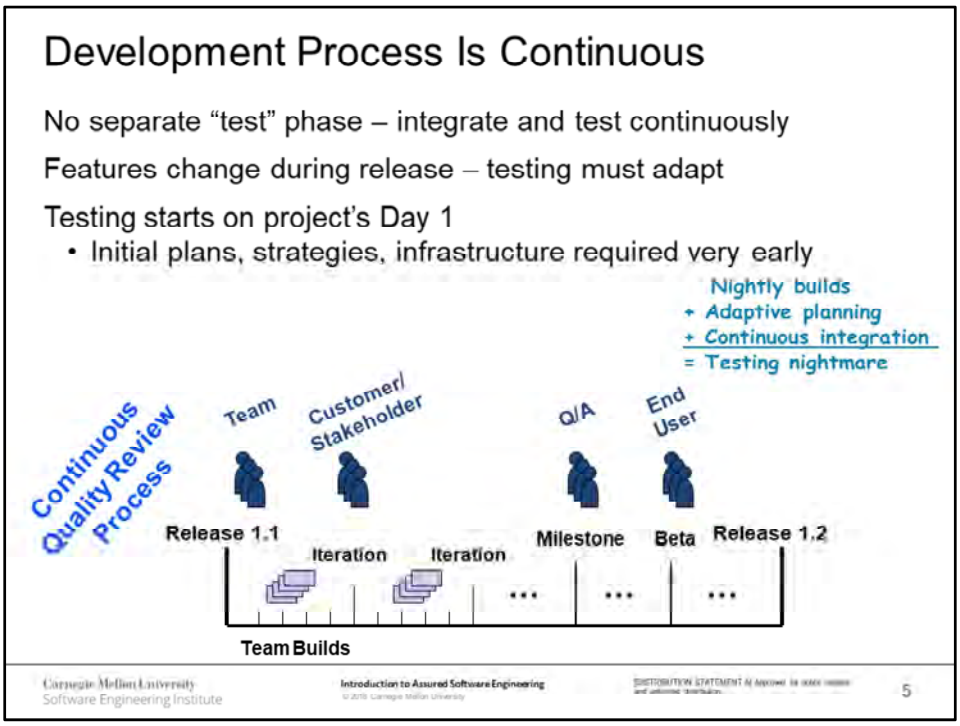
The unique-cause approach commonly is taught by presenting a truth table for an expression; for example, the decision **Z:= (A *or B) and (C or D) shown in Table 3. In the truth table approach, test cases*** that provide MC/DC are selected by identifying pairs of rows where only one condition and the decision

outcome change values between the two rows. In Table 3, the columns shaded in gray indicate the independence pairs for each condition. For example, test case 2 coupled with test case 10 together demonstrate the independent effect of **A, because A is the only condition that has changed value along** with the change in value of the outcome **Z. Although the truth table is a simple approach to showing the** independent effect of a condition, the truth table approach suffers from a number of limitations: (a) the truth table is unwieldy for large logical expressions; and, for a logical expression with *n inputs, only n+1* of the 2*n rows are useful; (b) the truth table addresses only one logical expression at a time; and, (c) the* truth table does not connect the inputs and outputs from the requirements-based tests with the source code structure.


Carnegie Mellon University
Software Engineering Institute
CERT
Module 24:
Continuous Integration and Testing
(Authored by Kevin Gary, Arizona State University)
Introduction to Assured Software Engineering
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



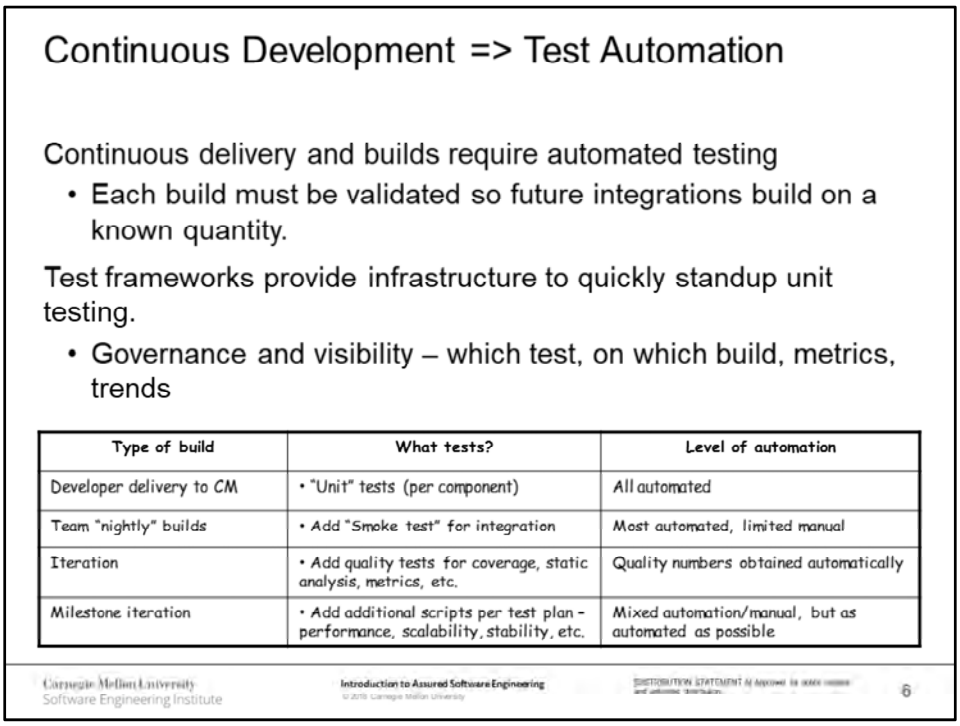
Instructor Note: This should easily be understood by the students. There are a lot of references for agile/XP. If the students are given a project to work on, it would be good for them to try an Agile approach.



Agile practices are an example of advances in development productivity.

Let's discuss some changes:

- Development changes to continuous integration and delivery of business value.
- Continuous integration exists on even $B projects – integrate, build, and test as early and often as feasible.



Quality organizations have changed their relationship with development. Quality used to be in a separate room, walled off so to speak.

- Send us the product when you think it is completed and we will tell you if it's good enough.

Modern quality organizations work with development – on the same team.

- Testing strategy, infrastructure, products, staff need to begin early – day one and evolve.
- Every "build" from development needs some form or quality review
- Cannot scale to this level without automation.

But, embedded systems require a blend of manual and automated testing – scripts plus human intervention.

XP Best Practices: Continuous Integration

What is Continuous Integration?

- Integrate & build the system several times a day
- Integrate every time a task is completed
- Let's you know every day the status of the system

Continuous integration and relentless testing go hand-in-hand.

By keeping the system integrated at all times, you increase the chance of catching defects early and improving the quality and timeliness of your product.

Continuous integration helps everyone see what is going on in the system at all times.

*If **testing** is good, why not do it all the time? (continuous testing)*
*If **integration** is good, why not do it several times a day? (continuous integration)*
*If **customer involvement** is good, why not show the business value and quality we are creating as we create it (continuous reporting)*

XP is Extreme Programming. There is a lot of reference material and many case studies on this. There is even an XP conference.