What Lies Ahead?

Challenge with Trends

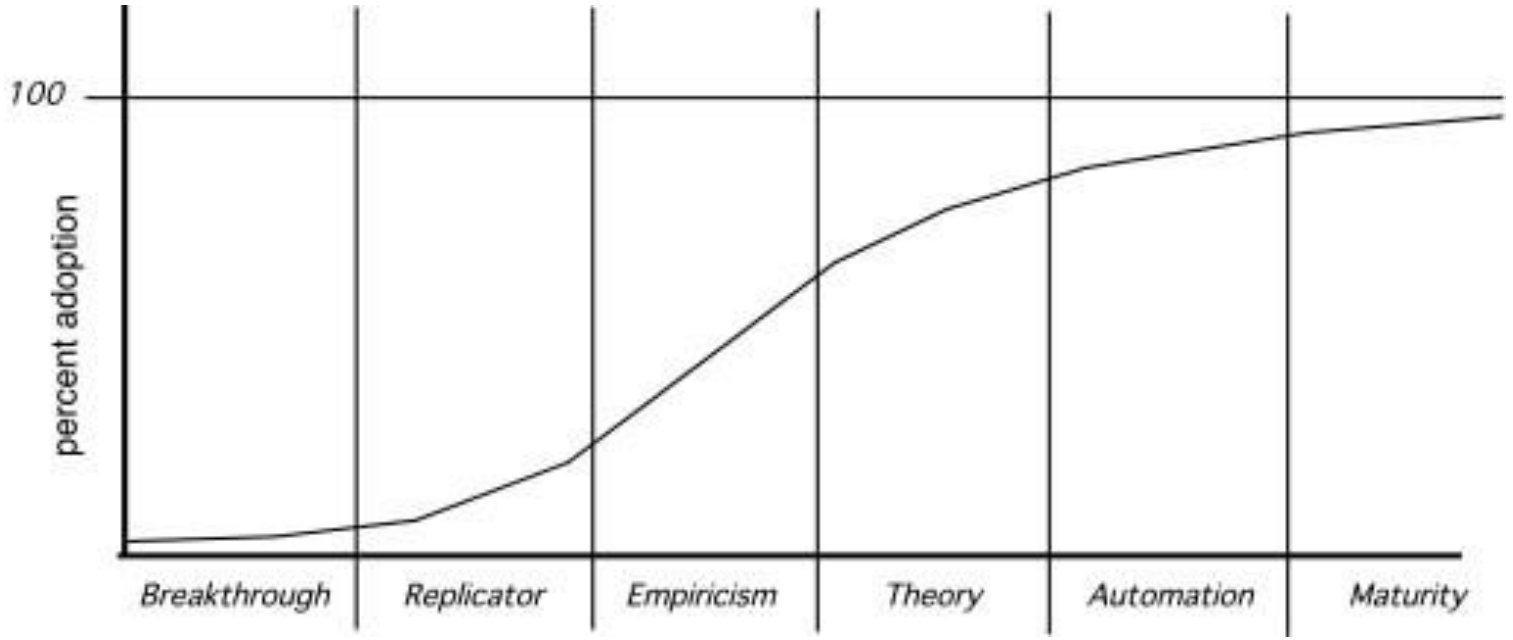
Challenges we face when trying to isolate meaningful technology trends:

- What Factors Determine the Success of a Trend?
- What Lifecycle Does a Trend Follow?
- How Early Can a Successful Trend be Identified?
- What Aspects of Evolution are Controllable?

Ray Kurzweil [Kur06] argues that technological evolution is similar to biological evolution, but occurs at a rate that is orders of magnitude faster.

- Evolution (whether biological or technological) occurs as a result of positive feedback—"the more capable methods resulting from one stage of evolutionary progress are used to create the next stage." [Kur06]

Technology Innovation Lifecycle



The Hype Cycle

Technology trigger*—a research breakthrough or launch of an* innovative new product that leads to media coverage and public enthusiasm

Peak of inflated expectations*—over-enthusiasm and overly optimistic* projections of impact based on limited, but well-publicized successes

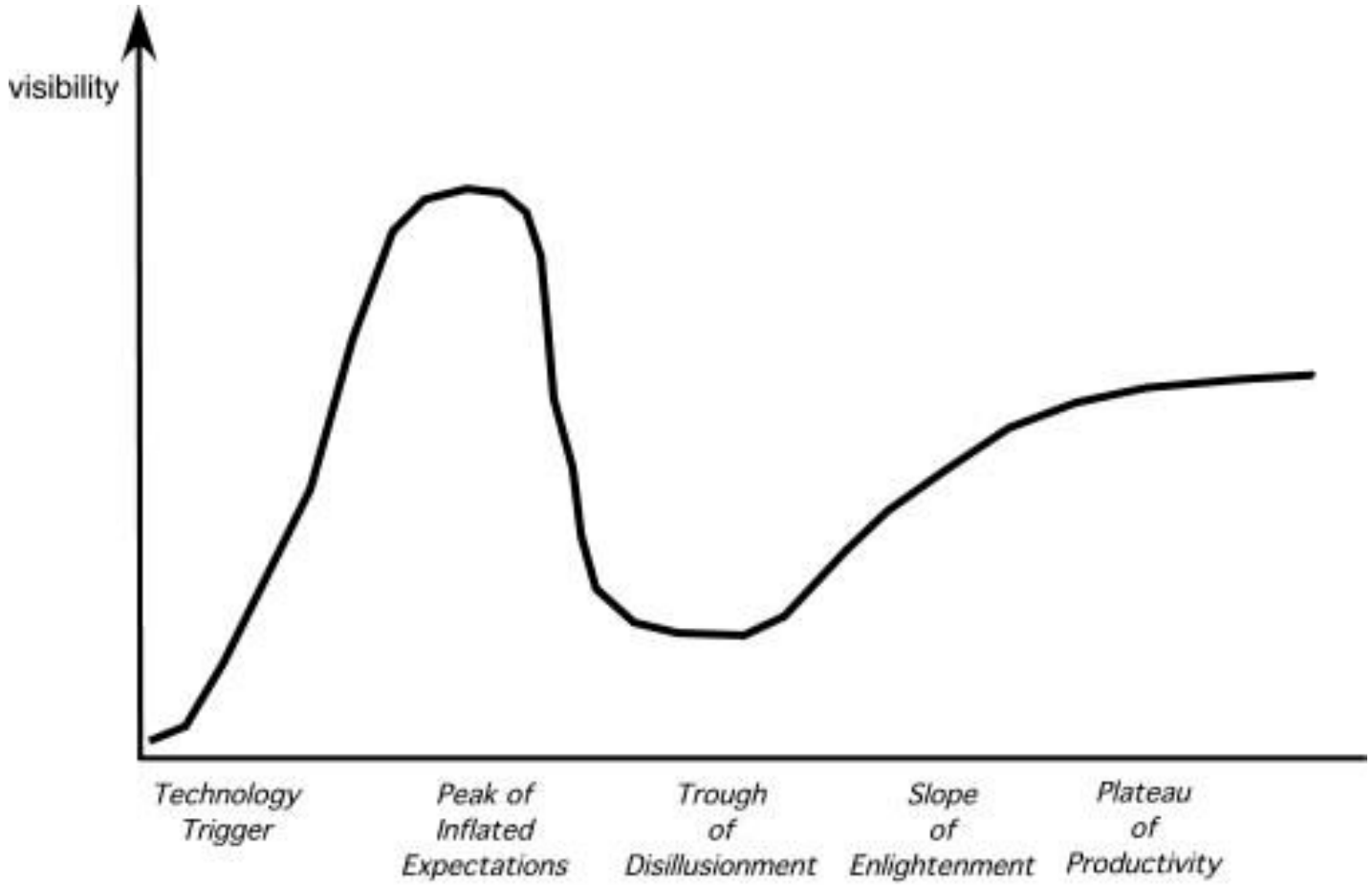
Disillusionment*— overly optimistic projections of impact are not met* and critics begin the drumbeat; the technology becomes unfashionable among the cognoscenti

Slope of enlightenment*—growing usage by a wide variety of* companies leads to a better understanding of the technology's true potential; off the shelf methods and tools emerge to support the Technology

Plateau of productivity*—real world benefits are now obvious and* usage penetrates a significant percentage of the potential market

Gartner Group [Gar08]

The Hype Cycle



What Lies Ahead?

Some trends that will exacerbate Hardware-software-human factors integration problems are

- ***Complex, multi-owner systems of systems.*** Current collections of incompatible, separately-developed systems will cause numerous challenges.
- ***Emergent requirements*****.** Demands for most appropriate user interfaces and collaboration modes for a complex human-intensive system.
- ***Rapid change*****.** Specifying current-point-in-time snapshot requirements on a cost-competitive contract generally leads to a big design up front, and a point-solution architecture that is hard to adapt to new developments.
- ***Reused components.*** Reuse-based development has major bottom-up development implications, and is incompatible with pure top-down requirements-first approaches.
- ***High assurance of qualities.*** Future systems will need higher assurance levels of such qualities as safety, security, reliability/availability/maintainability, performance, adaptability, interoperability, usability, and scalability.

Another concern with Software Engineering

Do we stand on quicksand or the shoulders of giants?

- Have you ever found that a new method or practice is just the re-branding and regurgitation of old?
- Do you find every new idea about software development seems to be at the expense and in aggressive competition with everything that has gone before?
- Does it seem to you that following that latest software development trend has become more important than producing great software?
- Have you noticed how in their hurry to forge ahead people seem to throw away the good with the bad? It is as though they have no solid knowledge to stand upon.
- Many teams carelessly discard expensive process and tool investments, almost before they have even tried them.
- Every time someone changes their job they have to learn a new approach before they can get on with the real task at hand. People cannot learn from experience as they are forever starting over.

Ivar Jacobson, Ian Spencer "Why we need a theory for Software Engineering", 2009

Emerging Soft Trends

Connectivity and collaboration *(enabled by high bandwidth communication)* has already led to a software teams that do not occupy the same physical space (telecommuting and part-time employment in a local context).

Globalization leads to a diverse workforce *(in terms of language, culture,* problem resolution, management philosophy, communication priorities, and person-to-person interaction).

An aging population *implies that many experienced software engineers and* managers will be leaving the field over the coming decade. The software engineering community must respond with viable mechanisms that capture the knowledge of these aging managers and technologists

Consumer spending in emerging economies *will double to well over $9* trillion. There is little doubt that a non-trivial percentage of this spending will be applied to products and services that have a digital component—that are software-based or software-driven.

Review Session (L18)

Session 10

- Software Architecture is represented by
  - components, properties, and interactions
- Architecture is a graspable model representing early design decisions
- Architectural Style is transformation on the overall design
- Architectural Pattern is a rule on the style w.r.t. one aspect
- Some common architectural styles are
  - Data Flow, Data Centered, Call-n-Return, Layered, Object-oriented
- Architectural Design process uses ACD, archetypes, refinements
- Architecture is assessed using
  - Data and Control, or ATAM, or Complexity analysis

Architectural Design Summary(L10)

- A software architecture provides a uniform, high-level view of the system to be built
- It depicts
  - The structure and organization of the software components
  - The properties of the components
  - The relationships (i.e., connections) among the components
- Software components include program modules and the various data representations that are manipulated by the program
- The choice of a software architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative architectures
- Data design translates the data objects defined in the analysis model into data structures that reside in the software

(More on next slide)

Architectural Design Summary(L10)

- A number of different architectural styles are available that encompass a set of component types, a set of connectors, semantic constraints, and a topological layout
- The architectural design process contains four distinct steps
  1) Represent the system in context
  2) Identify the component archetypes (the top-level abstractions)
  3) Identify and refine components within the context of various architectural styles
  4) Formulate a specific instantiation of the architecture
- Once a software architecture has been derived, it is elaborated and then analyzed against quality criteria

Session 11 – Component Level Design

- Conventional view vs. OO view of component
- Component level design principles
- Component packaging principles
- Component design steps
- Component considerations for WebApp
- Component based software engineering
- Component qualification, Component adaptation, Component composition, Component update

Component Design Summary

Component-level design defines data structures, algorithms, interface characteristics, and communication mechanisms for each component identified in the architectural design.

Component-level design occurs after the data and architectural designs are established.

Component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built.

The work product is a design for each software component, represented using graphical, tabular, or text-based notation.

Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps

Session 11 – User Interface Design

- Golden Rules of UI design
  - Place the User in Control
  - Reduce the User's Memory Load
  - Make the Interface Consistent
- User Interface Design Process
  - Interface Analysis
  - Interface Design Steps
  - Interface construction
  - Interface validation

Interface Design Summary

User interface design creates an effective communication between a human and a computer.

Proper interface design begins with careful analysis of the user, the task and the environment.

Based on user's tasks, user scenarios are created and validated.

Good user interfaces are designed, they don't happen by chance.

Prototyping is a common approach to user interface design.

Early involvement of the user in the design process makes him or her more likely to accept the final product.

User interfaces must be field tested and validated prior to general release

Session 12 – Testing Strategies

- Testing strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Verification (Are the algorithms coded correctly?) versus Validation (Does it meet user requirements?)
- General levels of testing are unit, integration, validation, and system
- Integration can be top-down, bottom-up, or sandwich
- OO software integration can be by thread, use, or cluster
- Validation is done by Alpha and Beta testing
- System Testing considers Recovery, Performance, Stress, Security, Deployment etc.

Summary – Testing Strategies

- Software testing must be planned carefully to avoid wasting development time and resources.
- Testing begins “in the small” and progresses “to the large”.
- Initially individual components are tested and debugged.
- After the individual components have been tested and added to the system, integration testing takes place.
- Once the full software product is completed, system testing is performed.
- The Test Specification document should be reviewed like all other software engineering work products.

Session 13 – Testing Techniques

- A good test is the one that “breaks” the software
- Main approaches to unit testing are white-box and black-box
- White-box testing is quantified using flow graph, basis paths, and cyclomatic complexity
- White-box testing verifies various loop constructs
- Black-box testing, aka behavioral testing, complements white-box with functional orientation
- Black-box techniques include equivalence partitioning, boundary value analysis, orthogonal array testing, model-based testing.
- Debugging occurs after successful testing; it is largely person-dependent
- Main debugging techniques are cause elimination, backtracking, and brute force

Summary of Testing Methods

- Software engineer's goal is to remove as many as possible early in the software development cycle.
- Testing can only find errors it cannot prove that a program is free of bugs.
- Two basic test techniques exist for testing conventional software, testing module input/output (black-box) and exercising the internal logic of software components (white-box).
- Test data is important for effectiveness of test case.
- Testing must be planned and designed.
- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Session 14 – Product Metrics

- Product measurement happens at various phases
  - Analysis
  - Design
  - Coding
  - Testing
- At each phase, metrics can focus on size/volume as well as quality parameters
- Function Points is a popular method of measuring size at analysis using information domain values
- Halstead's scientific measures created healthy debate
- Maintenance metrics can indicate both stability of software and ease of maintenance

Summary of Product Metrics

- Software engineers use product metrics to help them assess the quality of the design and construction the software product being built.
- Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively.
- Qualitative criteria for assessing software quality are not always sufficient by themselves.
- The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration.
- Then data are collected and metrics are computed and compared to pre-established guidelines and historical data.
- The results of these comparisons are used to guide actions w.r.t. work products arising from analysis, design, coding, or testing.

Session 14 – Process and Project Metrics

- Many metrics are common between process and project. Process metrics focus on process improvement while project metrics focus on tactical aspects.
- Metrics depend on approach to software size measurement
- Efforts were made to reconcile two popular approaches to software size : FP and LOC
- Quality is measured in terms of correctness, maintainability, integrity, and usability
- Organizations work towards establishing baselines in order to benefit from metrics

Summary of Process & Project Metrics

- Process and Project metrics are quantitative measures to gain insight into the efficiency of the software process and the projects conducted using the process framework.
- Software project management is primarily concerned with productivity and quality metrics.
- There are four reasons for measuring software processes, products, and resources
  - to characterize, to evaluate, to predict, and to improve
- *Process metrics* used to provide indictors that lead to long term process improvement
- *Project metrics* enable project manager to
  - Assess status of ongoing project
  - Track potential risks
  - Uncover problem are before they go critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software work products

Session 15 – Software Project Estimation

- Software project planning includes estimation, scheduling, risk analysis, planning for quality and change
- Estimation risk increases with Project Complexity, Project Size, The degree of structural uncertainty
- Scope & Feasibility be addressed before estimation
- It is recommended to use multiple methods of estimation & reconciliation
- Estimation is done using using decomposition or empirical models
- Decomposition can be problem-based or process-based
- The empirical models assume exponential growth of effort with increasing size

Summary of Estimation

- Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system.
- After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each.
- The final estimates are typically adjusted by taking project complexity and risk into account.
- Managers will not know that they have done a good job estimating until the project post mortem.
- It is essential to track resources and revise estimates as a project progresses

Session 16 – Software Project Scheduling

- A software project needs to meet time to market for success
- Basic principles of scheduling are compartmentalization, study of interdependencies, time allocation, effort validation, definition of responsibilities, outcomes and milestones
- Successful scheduling requires understanding of relation between people and effort
- A recommended distribution of effort across the software process is 40% (analysis and design), 20% (coding), and 40% (testing)
- Task networks represent interdependencies graphically, and enable identification of critical and near-critical paths.
- Schedule tracking can be done using qualitative methods or by earned value analysis
- Timeboxing is a technique whereby the project is not allowed to get “stuck” on a task

Summary of Project Scheduling

- To build complex software systems, many engineering tasks need to occur in parallel with one another to complete the project on time.
- The output from one task often determines when another may begin.
- Software engineers need to build activity networks that take these task interdependencies into account.
- Managers find that it is difficult to ensure that a team is working on the most appropriate tasks without building a detailed schedule and sticking to it.
- Sound project management requires that tasks are assigned to people, milestones are created, resources are allocated for each task, and progress is tracked.
- Earned value analysis is a quantitative technique for monitoring project progress

Session 17 – Risk Management

- Good governance ensures that IT investments have focus
- COBIT offers a framework for IT Governance
- Main characteristics of risk are uncertainty and loss.
- Software risks can be project risks, technical risks, or business risks. Also risks can be known, predictable, or unpredictable
- Risks can be handled by reactive or proactive(preferred) strategies
- Some predictable risk categories are product size, business impact, customer, process, environment, technology, staff
- US Air Force considers performance, cost, support, and schedule as risk components for software projects
- Risk table sorts risk by probability and impact
- Risk may be handled through mitigation steps.
- Risks are monitored for successful mitigation; in case of failure, contingency plan is triggered.

Summary of Risk Management

- Risks are potential problems that might affect the successful completion of a software project.
- Whenever much is riding on a software project, common sense dictates risk analysis
  - Yet, most project managers do it informally and superficially, if at all
- Risks involve uncertainty and potential losses. Risk analysis and management is intended to help a software team understand and manage uncertainty during the development process.
- The time spent in risk management results in
  - Less upheaval during the project
  - A greater ability to track and control a project
  - The confidence that comes with planning for problems before they occur
- Risk management can absorb a significant amount of the project planning effort…but the effort is worth it
- The work product is called a Risk Mitigation, Monitoring, and Management Plan (RMMM) or a set of Risk Information Sheets (RIS).

Thank You.