Software Process Models

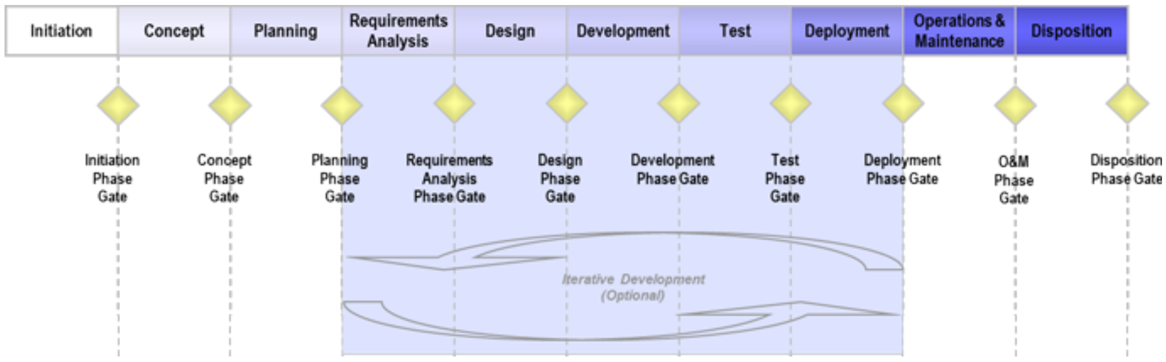
Software Process Models are used to explain the order of activities and how to move from one activity to the next. The software manager chooses and applies the best process models for a specific software development project. Process models have evolved over time. In this module, we discuss a variety of process models and how they have changed including Waterfall, Evolutionary (or Incremental), Spiral, Agile Development, Commercial off the Shelf (COTS) Integration, and Prototyping.

Definitions

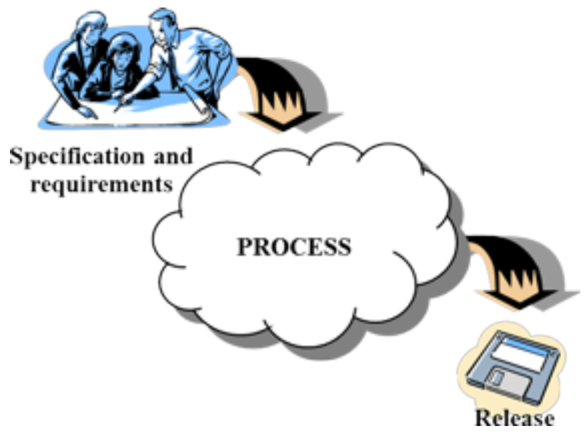
A **product life cycle model** is a series of phases or stages showing how the product will be conceived, developed, and disposed. For software products, the life cycle should:[1]

- Provide guidelines to organize, plan, staff, budget, schedule, and manage software project work over organizational time, space, and computing environments
- Outline the documents to produce for delivery to client
- Be the foundation for determining what software engineering tools and methodologies will be most appropriate to support the different life cycle activities
- Offer the framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle (Boehm 1981)
- Be the basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

The following image shows an example of a product life cycle.



A **software process model or development approach** is used in the life cycle and defines the sequence of activities and the criteria for transitioning from one activity to the next. It focuses is on "what" will occur. Examples of software process models include waterfall, evolutionary, prototype, spiral, and agile development.



A **methodology** defines "how" to proceed through each step and represents the products of each step. Examples of methodologies are real-time structured analysis and design, and object oriented analysis and design.

A **procedure** defines how you do something and may include the "who, what, when, and where".

A **guidebook** is a reference set of generic procedures or best practice guidance.

A **plan** is an ordered definition of the "who, what, when, and where". It defines a written strategy, the "why", and may include the tactics or "how" the tasks will be accomplished.

Module 5 will cover procedures, guidebooks, and plans.

Common Software Process Models

"The lifecycle model you choose has as much influence over your project's success as any other planning decision you make."

-- *Steve McConnell, Rapid Development*

In reviewing some of the more common software process models, let us start with a review of the models no longer in use today so that we can better understand those that are in use today.

Historical Models

In this section of the module, we will address the historical models used in the software development process, including the "Code and Fix" approach, the Stagewise model (replaced by the Waterfall Model), and the Transform model.

Code and Fix Model

Code and Fix is the basic model used in the earliest days of software development. It contains two steps:

1. Write the code
2. Fix problems in the code

Its advantages are there is no overhead, anyone can use it, and there are the immediate signs of progress. The disadvantages are this method is only suitable if one person is developing the entire software product. Continually fixing the same code leads to poor structure and unmaintainable code, that is code that typically does not meet the user needs, and code that is hard to test since there is no preparation and planning for testing. Nevertheless, over 50 years after software development began, there are still software developers who practice code and fix!

Stagewise Model

Stagewise was introduced in the mid-1950s; it was largely based on hardware manufacturing. Software was developed following eight sequential and successive stages:

1. Operational Plan
2. Operational Specifications
3. Coding Specifications
4. Coding
5. Parameter Testing
6. Assembly Testing
7. Shakedown
8. System Evaluation

This model is no longer used; in its place, the Waterfall Model evolved.

Transform Model

The Transform model assumes the capability exists to automatically convert a requirement into a software program. The model then follows five steps to:

1. Formally specify the desired product (that is the initial understanding)
2. Automatically transforms the specification or requirement into code
3. Improves performance of code by adding optimization parameters to the transformation system (step 2)
4. Allows users to exercise the resulting product
5. Adjusts the specification (and repeat these steps) based on operational experience

Its advantages are the Transform Model eliminates the problem of "spaghetti code" because the code is regenerated to satisfy performance and user satisfaction; however, the disadvantages include that automatic transformation systems are only available in a limited domains usually some fourth generation (4GL) applications, and both reuse and Commercial off the Shelf (COTS) integration software are rarely supported. This model was initially intended in the Artificial Intelligence discipline and may have a place in the future but to date it has largely been excluded.

Waterfall Model

The Waterfall Model was introduced as a refinement to the Stagewise Model in 1970. It added feedback loops between steps and allowed prototyping to support requirements analysis and design. This approach to software development has been the basis for most software acquisition standards used in government and industry for over 40 years! Generally the guidelines to choose this model for a system:

- Developed in less than 12 months
- Requirements are known and stable
- Cannot practically break into builds (or releases)

- Added cost of developing support software (emulation, simulation, test) is more than 20% of total development cost
- Timely delivery of the entire system is the driver

The problem with Waterfall is its emphasis on elaborate documentation and the difficulty in allowing change once the system is being developed. For certain classes of software including interactive and end-user applications, the completion criteria does not work well. Although better than in the past, it is not a perfect match for Commercial off the Shelf (COTS) software integration and sometimes requires a forced work around.

Waterfall is the basis for the current process models that are often grouped under the larger category called Iterative Process Models, which includes Evolutionary (or Incremental), Spiral, COTS Integration, Agile Development, and Prototyping.

Evolutionary (or Incremental) Model

The Evolutionary or Incremental Model was created as a modification of the Waterfall Model. It allows for a subset of functionality to be developed and delivered; this first product is called Build 0 (or Release 1). Additional functionality is added to subsequent builds / releases. Each build goes through a complete cycle of the tailored software activities; the release product is a working product. As the next build begins, the current development build is tested and transitions to maintenance. The problems with Evolutionary are it can create "spaghetti code" if components of the early builds are continuously modified for later builds, and it assumes that the user's operational system will be flexible enough to accommodate unplanned evolution paths which is not always a valid assumption.

The characteristics of Evolutionary include:

Duration: Plan the build to span a period of time commensurate with higher level planning which ensures funds and resources are available by fiscal year or system event.

Minimum Duration: Generally not less than 4 months since the overhead associated with planning, testing, documenting, and delivering a build will outweigh the quick turn-around if the duration is too short.

Maximum Duration: Generally not more than eight (8) to 12 months since customers will be dissatisfied if they have to wait too long for new features and problem resolutions.

Build Overlap: Planning activities and requirements for the next build can begin once the current software is developed and being formally tested. To minimize baseline control and merge problems, avoid overlapping builds prior to the software development phase.

Documentation: Evolving project technical information should be documented and available for build planning. After Build 0, adding appendices to documents can reduce cost.

Delivery: Testing should be complete and all deficiencies resolved before delivery. The Program Management Office (PMO) and customer often request early delivery that can create build overlap problems.

Planning: To accommodate evolving requirements or priority fixes identified during the build effort, re-planning will likely be necessary.

Metrics: Productivity data or metrics should be collected and tabulated. Productivity should increase significantly in later builds as the programmatic issues stabilize and the development team gains experience. Metrics data becomes more complex as code is created, modified, and deleted.

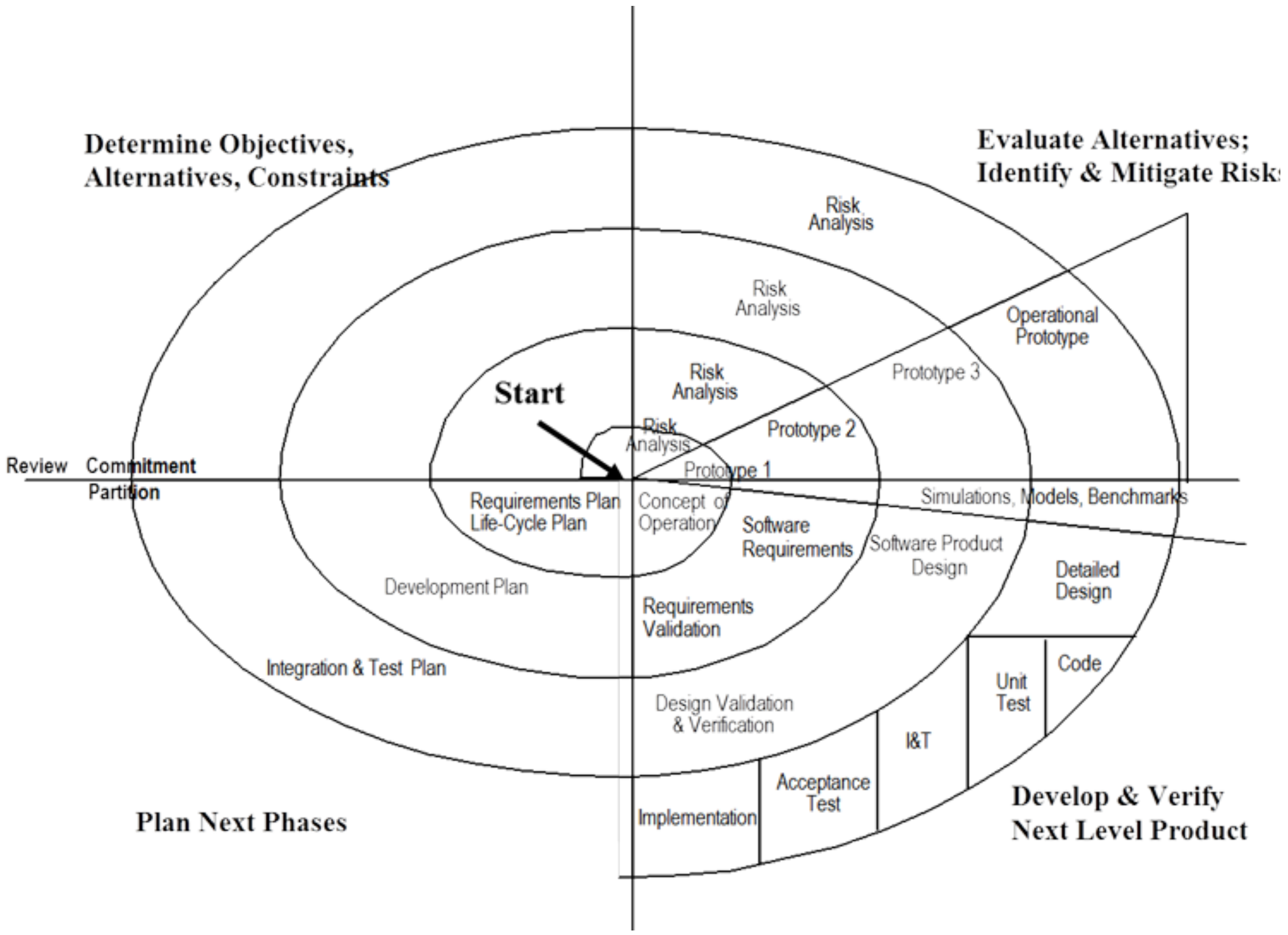
Spiral Model

Barry Boehm introduced the Spiral Model in mid-1980s; it is based on refinements to the Waterfall Model. Each cycle involves a progression that addresses the same sequence of steps for each level of elaboration. The iterations up until the last one produce a prototype or simulation and it is not until the last iteration that the product is developed and built. The Spiral Model is especially useful if the project is high risk. Its range of options accommodates the good features of the other software process models while its risk driven approach avoids many of their difficulties. The Spiral Model focuses early attention on options involving software reuse, and it emphasizes eliminating errors and unattractive alternatives early. The problems with the Spiral Model are it is not applied extensively to contract software acquisition, relies heavily on risk-assessment expertise, it is often deemed as too expensive, and is not used as much as the more established models.

As a system advances, costs increase. Early iterations of system development are the cheapest requiring less time and money. It costs less to develop concepts of operations (CONOP) than requirements, less to develop requirements than design, less to develop the design than the software code, etc... Yet, as costs increase, risks decrease.

Here is a diagram of the Spiral process showing risk as the driving force. As you pass through each quadrant, you must:

- Determine the Objectives, Alternatives, and Constraints;
- Evaluate the Alternatives, and Identify and Mitigate Risks;
- Develop and Verify for the Next Level of the Product; and finally
- Plan the Next Phases



Barry Boehm in his classic book, Software Engineering Economics, uses the Spiral Model to successfully improve productivity at TRW over a five-year period.

	Round 0	**Round 1**	**Round 2**
Objectives	• Significantly increase software (SW) productivity	• Double SW productivity in five years	• User friendly system • Integrated SW Office Auto Tools
Constraints	• At reasonable cost • Within context of TRW culture • (Gov contracts, high tech, security)	• $10,000 per person investment • Within context of TRW culture • Preference for TRW products	• Support all project personnel and lifecycles • Portable • Deliverable to customers
Alternatives	• Mgmt: Proj Org, policy, planning • Personnel: staffing,	• Office: private/ modular/ ... • Comm: LAN/ Star/	• Stable, reliable service • OS: VMS/ AT&T Unix/ Berk Unix/

	Round 0	Round 1	Round 2
	incentives, training • Tech: tools, workstations, methods • Facilities: offices, communications	Concentrators/. • Terminals: Private/ Shared/ smart/...	• Host vs Target/ Fully portable
Risks	• May be no high-leverage improvements • Improvements may violate constraints	• May miss high leverage options • Workstation cost • TRW LAN price/performance • Extensive external surveys/visits	• Unix performance • User-unfriendly systems • Workstation/Mainframe compatibility • Survey of Unix using organizations
Risk Resolution	• Internal surveys, Literature search • Analyze cost model • Analyze exceptional projects	• TRW LAN benchmarking • Workstation price projections	• Workstation study • Requirements participation
Next Level Product (Risk Resolution Results)	• Single time-sharing system not feasible (security compartmenting) • Mix of alternatives can produce significant gains (factor of two in five years)	• Operations concept: private offices, TRW LAN, personnel terminals • Defer Operating System and tools selection	• Top level requirements specification • Host/target with Unix host • Unix based workstations • Initial focus on tools for early phases
Plan for Next Phase	• Six person task force for six months • More extensive surveys/analysis (internal, external, economic) • Develop concept of operation	• Partition effort into Software Development Environment (SDE), facilities, management improvements • Develop first- cut, prototype SDE	• Overall development plan ○ for tools ○ for LAN

Note that over the five-year period, it was highly likely that software productivity would improve (as it likely would continue to increase today) yet Barry Boehm found that the Spiral Model further reduced risk and yielded the productivity increase sought. It remains a very expensive model to implement and therefore is generally only used on very large, long-term, high-risk projects.

Commercial off the Shelf (COTS) Integration Model

In the early days of software development, the only commercial software included in the typical software system was the operating system and often the operating system was modified to meet the needs of a specific project.

Today, customers expect software products to use COTS whenever possible. Operating Systems, database management system products, communications packages, graphical user interface products are typically an integral part of developed software systems. Some developers and integrators are building systems where COTS products meet as much as 80% of the requirement. Using COTS products reduces the cost of software development but increases the amount of integration required. Traditional software size and cost estimation techniques do not adequately address this issue.

COTS Integration offers many advantages to customers including:

- Less development time and therefore potentially a lower cost
- Seeing what you get since you are using existing, working products
- High reliability products that were proven and field tested
- Vendor maintains the product
- Specialists, who focus on one product domain, develop the product
- Large variety of options to choose from

However, be aware:

- Sales reps promise everything but products seldom deliver
- Product must be generic to support broad customer base
- Product may not address unique requirements of your program
- Vendor may not provide information needed for Reliability, Maintainability, and Availability (RMA) analysis
- CMMI level or ISO 9000 compliance may be unknown
- Products may not be interoperable with other COTS products
- New releases occur frequently and asynchronously on the vendor's schedule
- Vendors may provide limited support for previous versions

A COTS solution is more that procuring, installing, and demonstrating requirements. All phases and activities of the traditional software development are still essential; some phases such as code and unit test are scaled down and other phases such as integration are scaled up. Requirements analysis, design, code, test, integration, formal test, configuration management (CM), and quality assurance (QA) are still essential.

Requirements analysis should be viewed from a new perspective. Project cost and risk can be reduced if requirements which cannot be allocated to COTS are candidates for change. The customer and developer must take a hard look at these requirements to ensure that they are really necessary or cost effective. COTS products often provide additional functionality beyond the business and system requirements; the customer and developer should consider adding requirements to allow for these functionalities, as users will eventually tap them anyway.

COTS product selection must be approached from a disciplined perspective. The naive software manager or customer might simply call several vendors, review the literature, and select and procure the product. The sophisticated manager will consider the hands-on evaluation of products, product to product interface testing, application to product interface testing, product ports to a variety of platforms, vendor and product considerations beyond functionality to ensure the checklist is used, and how to bound the search. These activities can be scaled up or down depending on the maturity of the products or the previous in-house experience.

Integration, that is COTS to COTS and COTS to application, is a major cost driver for the COTS solution. Standards based COTS products facilitate integration. Integration framework products are emerging and can simplify integration. Vendors and vendor partnerships offer suites of integrated products; however, selection becomes limited with this approach. Products with published application programming interfaces (APIs) can simplify integration but code development is required.

Focus on COTS requires more effort earlier in the life cycle, even in the pre-contract time frame. Vendor surveys, hands-on evaluations, prototypes of integration, mapping requirements to COTS products are all necessary activities surrounding potential COTS solutions. Bid and Proposal (B&P) budget must fund many of these activities, often during a time when schedule constraints are tight and correspondence with the customer is limited or prohibited.

Current Process Models

Agile Development Models

Agile Developments are very fast-paced and nimble. The Agile Development Model requires the programmer team and the business experts to closely collaborate; involves face-to-face communications, which takes precedence over written documentation; ensures frequent delivery (usually every two to six weeks) of new deployable business value; uses tight, self-organized teams; and aligns the code and team avoiding requirements crises.

The accepted differentiators between Agile Development and other process model approaches are that Agile Development Models are more adaptive and less predictive. Agile methods anticipate change. An Agile approach will deliver user value within schedule and cost. Agile methods are people-oriented rather than process-oriented. Agile methods require separation of design and focus on construction, anticipate requirements volatility, are highly collaborative, and require minimal planning.

Agile Development is **adaptive** rather than **predictive**

- Other process models use detailed planning and the interaction of activities within the plan to predict the outcome. This works very well when the end-state is known well enough to be planned in detail, requirements are stable, and cost and schedule are variables. Nature is to resist change.
- Agile methods use lightweight planning to define the outcome. This works very well when the end-state is not known well enough to be planned in detail. Cost and schedule are stable, and requirements are unstable. Nature is to anticipate change and limit rework.
- Lean, iterative development method uses in-process documents and formal reviews replaced by engineering models and demonstrations. Documentation is extra.

Agile Development values staying **within** cost and schedule

- Other engineering methods focus on satisfying requirements. This works very well when the end-state is known well enough to plan functionality. Requirements are stable, and cost and schedule are variables. The nature is to resist change.
- The agile methods define functionality that meets cost and schedule. This works very well when the end-state is not known well enough to be planned in detail. Cost and schedule are stable, and

requirements are unstable. Agile is risk and priority driven to produce high customer value early. The nature is to anticipate change and limit rework.

Agile Development anticipates **Requirements Volatility**

- The most typical source of issues with a development project is requirements changes, that is requirements creep: new, modified, and deleted requirements. If you cannot get stable requirements, you cannot have a predictable plan to develop the requirements. In Agile Development, you build only what you know you need. Why build for interoperability, scalability, and expandability unless you are sure it is required.
- Customer valuable functionality is demonstrated at each iteration within short, time and resource-boxed development cycles (2–6 weeks). Operational capabilities evolve.
- Test-driven development with continuous integration and test.
- Be prepared to stop at any time.

People-oriented rather than process-oriented

- Agile methods are more people-oriented (dependent) rather than process-oriented. Engineering methods define a process that will work well whoever happens to be using it. Agile methods rely on a high level of direct intra-team communication. Agile methods assert that no process will ever make up the skill of the development team, so the role of a process is to support the development team in their work.
- The fabric of each team is unique.
- The combination of characteristics of each project are different.
- Good teams are really greater than the sum of the individuals.
- Some organizations focus solely on a process where the people are replaceable parts. Based upon the assumption that software intensive systems development can be done like an assembly line. Based upon the assumption that trained, educated people are commodities. If you expect your developers to be plug-in compatible programming units, you do not treat them as individuals; this lowers morale and productivity. Good people look for job satisfaction and recognition and will find it elsewhere.

Highly Collaborative

- Agile Development requires continuous communications within teams, among teams, and with customer/users. Small, self-organizing, self-directed teams (usually less than 10 developers) have at least daily meetings.
- Customers are included in the team.

Separation of Design and **Focus on Construction**

- The engineering metaphor is based on the construction methodologies where System Design (Architecture through top level design) was/is considered design that requires creative and talented people. Detailed design and coding is typically considered construction.
- Agile Methodology focuses on construction, not design. The only compilation is construction. System development requires significantly higher investment of talented, creative people in design than other engineering disciplines such as mechanical, civil, or electrical.

Minimal Planning

- A predictive project is often measured by how well it met its plan. Cost, schedule, requirements, and project quality are measured by conformance to plan.
- When the requirements are not well known, did the customer get more value than the cost put into it? A closer relationship is required between the Customer and Developer.
- Constrain iterations using any two of the following: cost, schedule, or functionality.
- A good predictive project will go according to plan and a good un-predictive project will go according to customer need.

Manifesto of the Agile Development Model

Please take a moment to read the Manifesto of the Agile Development Model. You should read the introduction and then click the 'Twelve Principles of Agile Software' to read the principles behind the manifesto.

Agile Development Model works best with small teams, often two (2) to five (5) people and certainly no more than 20–40 people. Popular approaches to Agile include: Extreme Programming (XP), SCRUM, Dynamic Systems Development Method (DSDM), Adaptive Software Development, Crystal, and Feature-Driven Development (FDD).

Features of Extreme Programming (XP) include:

- Programmer pairs
- Customer negotiates desired features for 2 week cycle
- Useful value every two weeks
- Team tests against customer desires
- Continuous testing
- Customer is a member of the team
- Continuous design improvement
- Documentation is outside of XP

Features of SCRUM include:

- Self-organizing integrated team typically 5–10 people including programmers, testers, systems administrators, configuration managers, and others
- Team meets at least once per day; multi-Scrum meeting once per day
- Customer continually defines prioritized list of features and a backlog is maintained
- Customer is a member of a team
- A team agrees to develop a set of features in a four (4) to six (6) week *Sprint*
- Useful value every *Sprint*
- Team tests against customer desires
- Continuous design improvement
- Documentation is negotiated

Features of Dynamic Systems Development Method (DSDM) include:

- Requirements are specified at a high level; assumes that nothing is built perfectly the first time, but that 80% of the proposed system can be produced in 20% of the time
- Joint Application Development (JAD) which involves creating a team with software developers, testers, customers, and users all with varying backgrounds to jointly determine the requirements and prototype the effort

- A surge teams of three (3) to seven (7) people with the necessary skills to deliver capability and who agree to develop a set of features in four (4) to six (6) week increments
- A wrapper for XP and Scrum
- Documentation is negotiated

Features of Adaptive Software Development include:

- Continuous adaptation of the process
- Repeating series of speculate, collaborate, and learning cycles
- Continuous learning and adaptation to the emergent state of the project
- Mission focused, feature based, iterative, time-boxed, risk driven, and change tolerant
- Evolved from rapid application development work

Features of Crystal Family include:

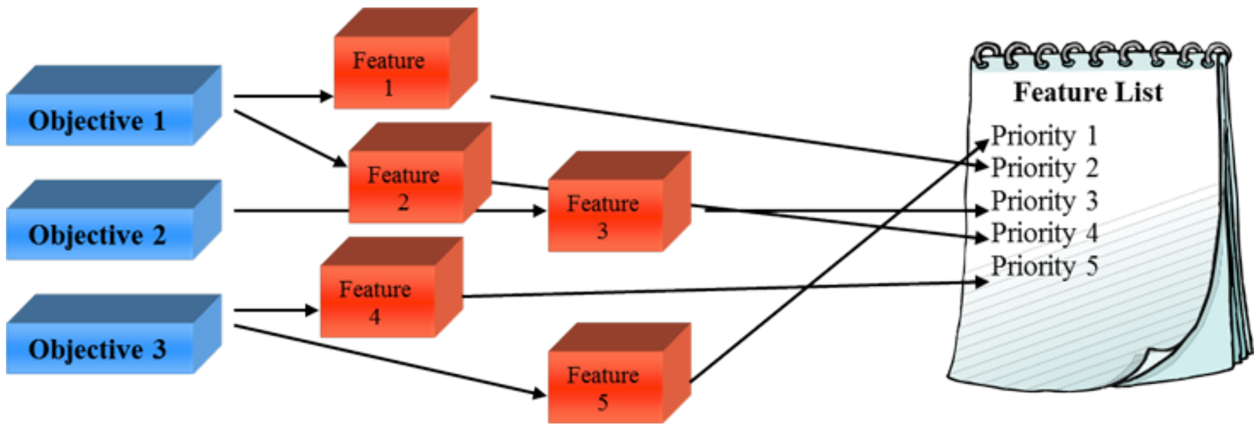
- Different project types require different methods. Crystal Family classifies projects by:
    - Number of development team staff
    - Amount of Risk
- Crystal methodologies are divided into color-coded bands. Each color has its own rules and basic elements

3-6	10-20	25-50	50-100	100-200	200-500	800+

- Each methodology is as light as possible and tuned to the project
    - Larger methodologies for larger teams
    - Denser methodologies for more critical projects
    - Interactive communication is the most effective
    - Weight is costly
- Teams define process within guidelines
- Delivery every three months

Features of Feature-Driven Development (FDD) include:

- Establish a set of objectives for the solution
- Turn objectives into *features*; create a prioritized *feature* list
- Establish an architecture within which to construct the solution; plan by *features*; build by *features*
- A feature is a capability that can be developed in four (4) hours to eight (8) days that is useful to the user and may also be necessary for infrastructure
- Project should organize to the nature of the project



Prototype Model

The Prototyping Model is usually a part of a software development project. It is not necessarily a separate model but generally is used in conjunction with one of the established models. It is important as it is used to avert risk and helps with analyzing requirements. Prototyping provides a scaled down version with limited performance of the product. Prototyping is often used early in the life cycle with users' direct involvement. Prototyping:

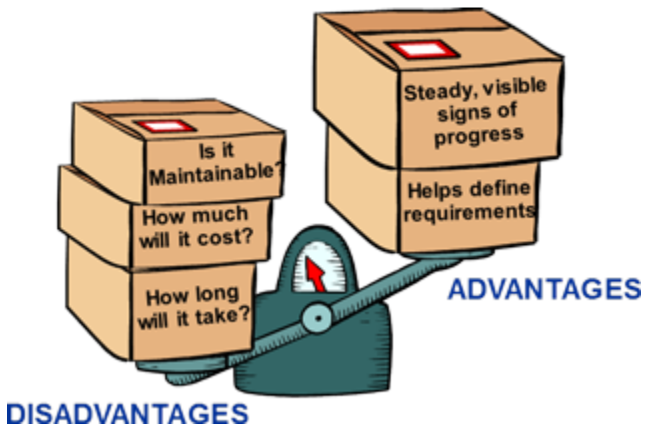
- Assists customers in identifying what they want
- Helps ensure potential problem areas have solutions
- Provides checkpoints to demonstrate observable progress
- Provides the customer a quick, usable capability to see what the system will look like

Generally, you begin with most visible part of the system, demonstrate it to the customer, continually update the prototype based on feedback, then stop and document when customer is satisfied. There are three types of Prototypes:

Throwaway Prototype (or Rapid Prototyping) is used during early software engineering to pinpoint requirements. It reduces risk by allowing experimentations with different implementations. It is developed without respect to standards or future maintenance. The prototype model may become the starting point to simplify requirements; however, most importantly it is discarded after preliminary design and only the requirements and algorithms survive.

Working Model Prototype normally does not lead to a larger software development effort. This prototype is used to examine performance characteristics and human machine interfaces (HMI) during the development that are not usually part of the delivered system.

Evolutionary Prototype becomes part of the delivered system. The tailored development practices apply. This should be the least common form of prototyping as it can become very expensive to incorporate an evolutionary prototype into a developing system.


Steady, visible signs of progress
Helps define requirements
ADVANTAGES
Is it Maintainable?
How much will it cost?
How long will it take?
DISADVANTAGES