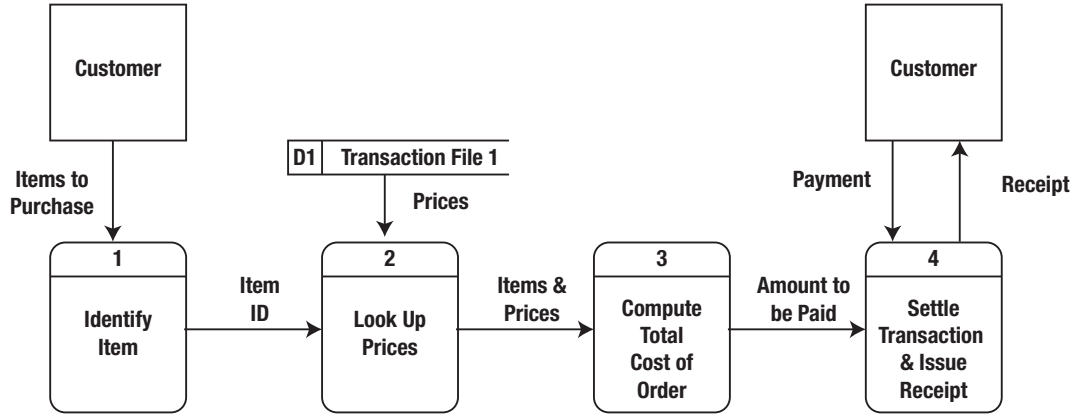
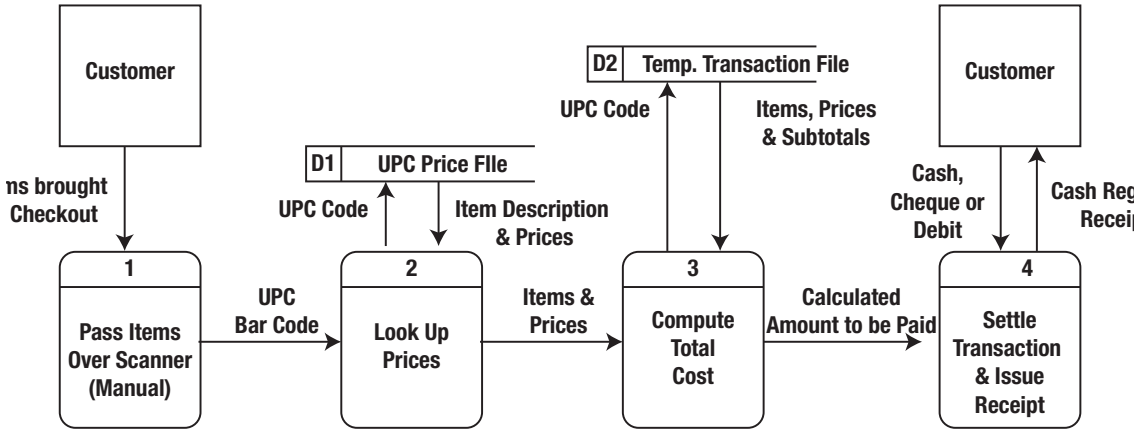
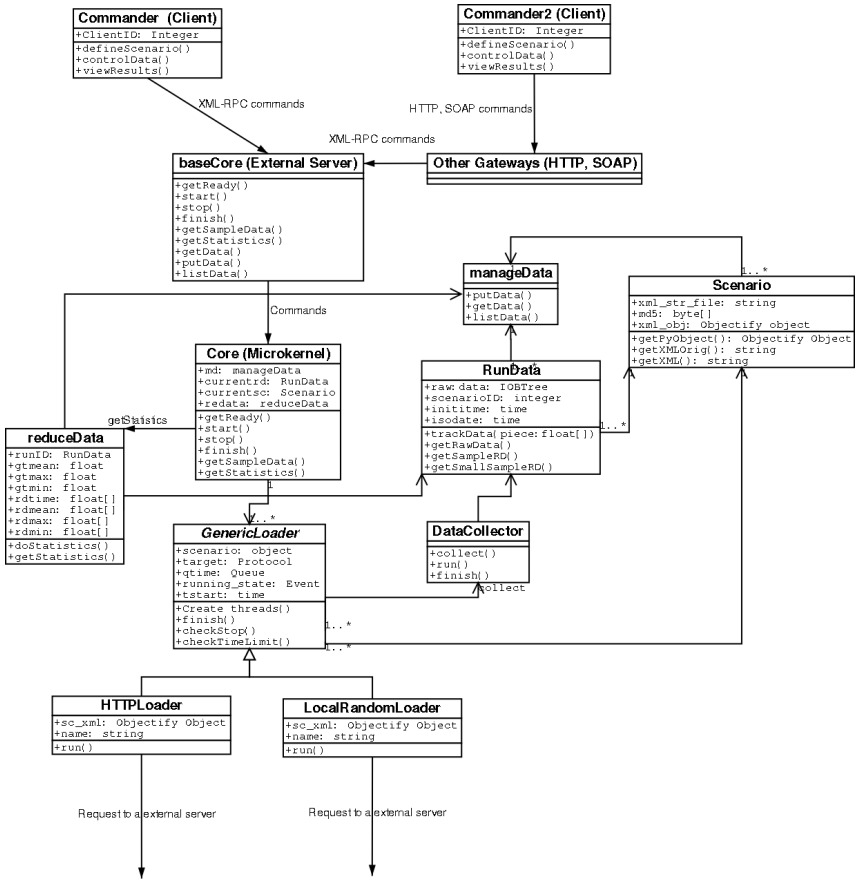
2. System Design

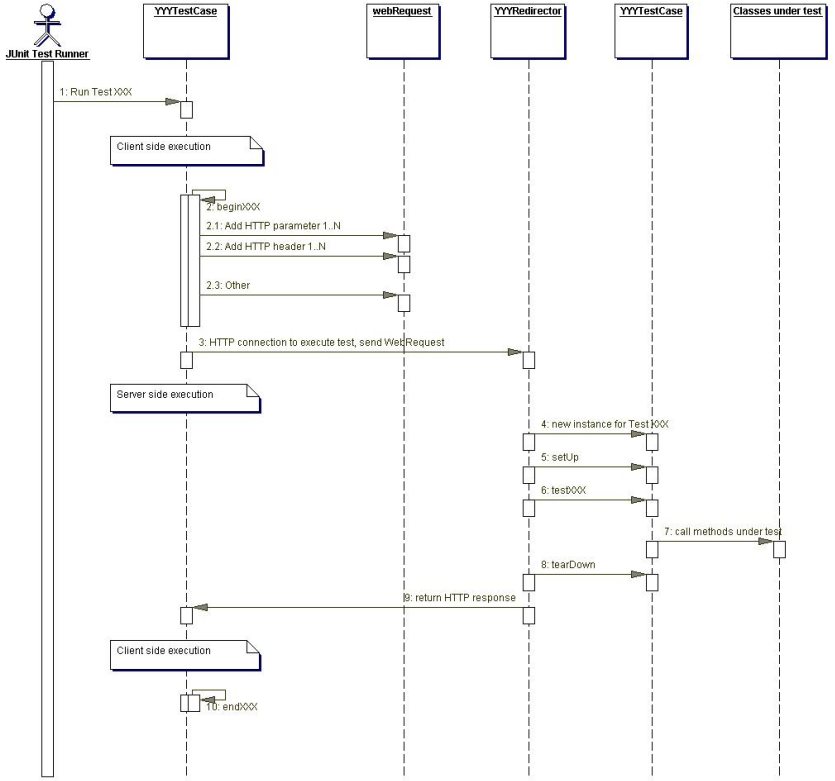




Dataflow diagrams (Logical vs. Physical)

3. Program Design

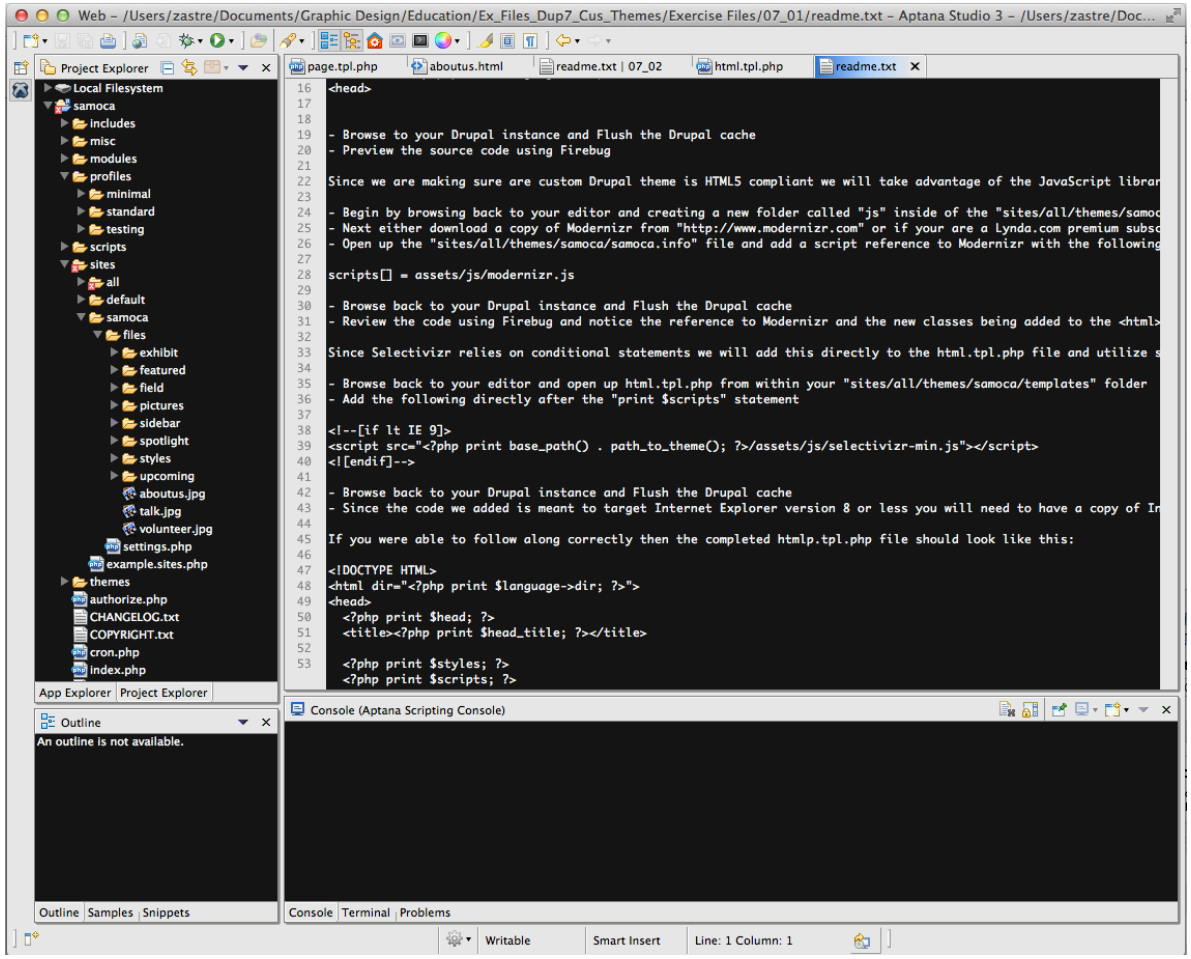




Class diagram example: static properties

Sequence diagram example: dynamic behavior

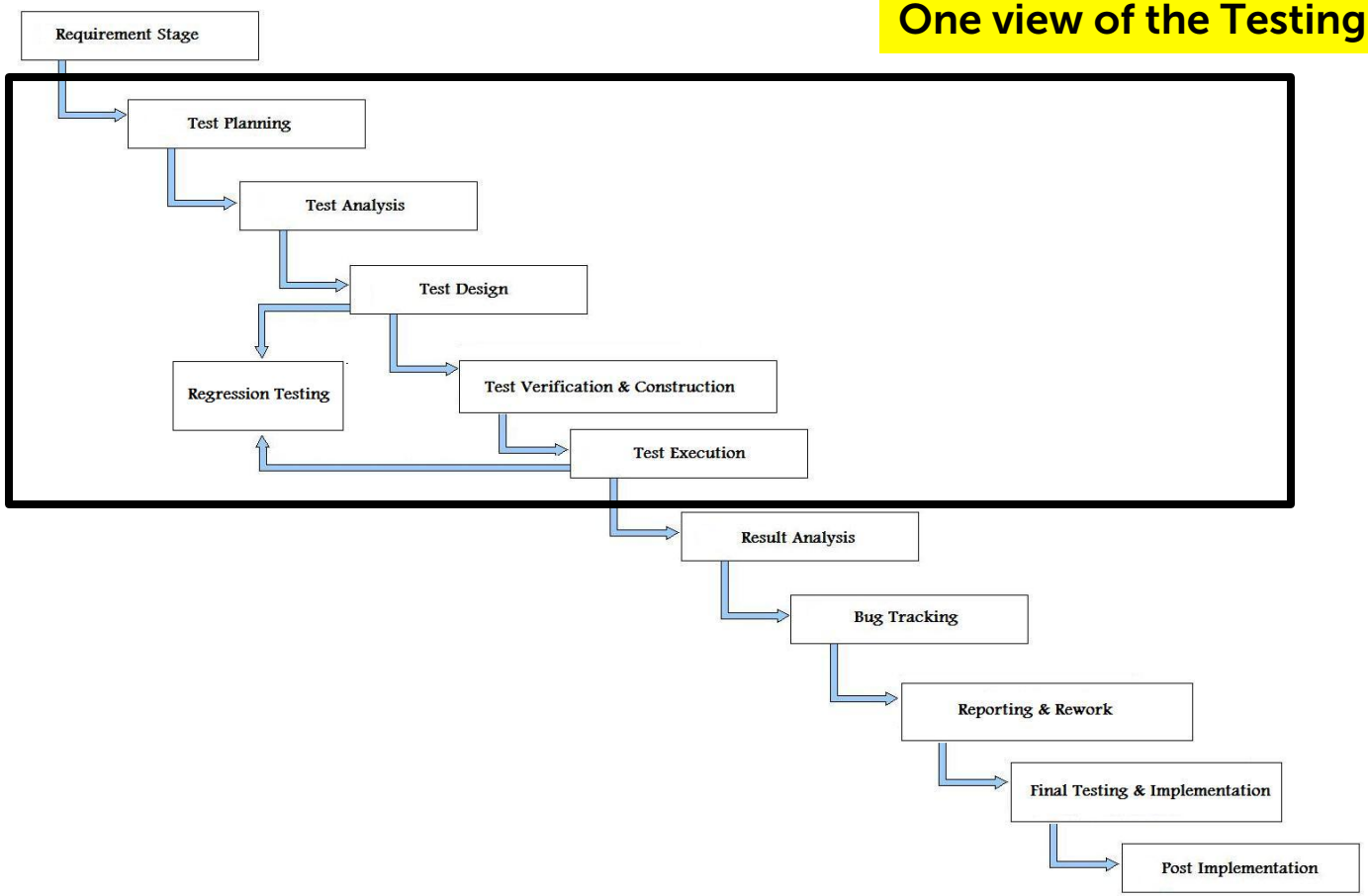
4. Writing the code



Screenshot of Aptana Studio 3

5. Testing

One view of the Testing sub-cycle



6. Deployment; 7. Maintenance

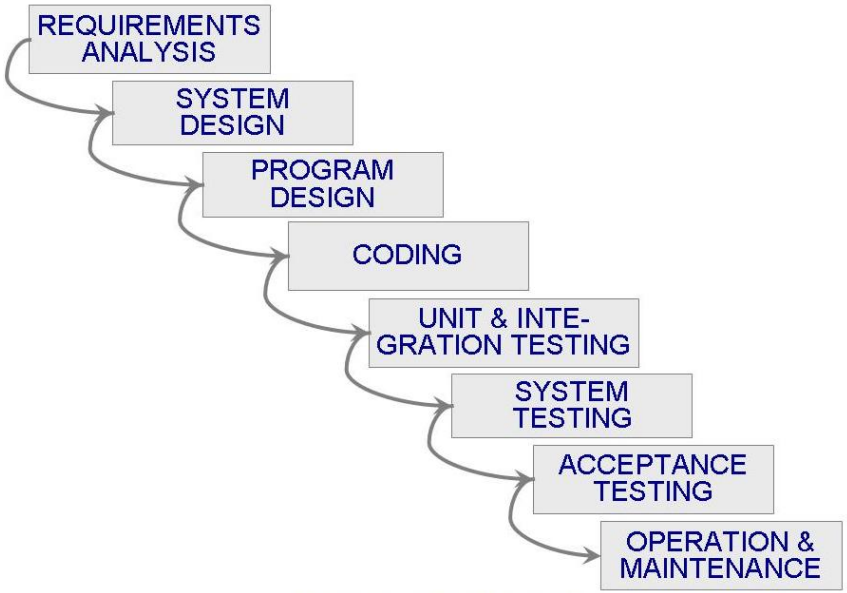
- Deployment:
  - May also involve testing system using client data, mockup of client's environment
  - Usually includes acceptance testing
  - Three actions: delivery, support, feedback
- Maintenance:
  - Fixing problems
  - Adding new functionality, extending existing functionality
  - Least glamorous but perhaps most important phase
  - (Iceberg model of visualizing effort!)
  - May also include software re-engineering (i.e., a rebuilding activity)

Some software process models

A. Waterfall model
B. V model
C. Prototyping model
D. Operational specification
E. Transformational model
F. Phased development: increments and iterations
G. Spiral model
H. Agile methods

A. Waterfall model

- Pure form of the waterfall model indicates a one-way flow of information
  - Data and details never move upstream
  - Model assumes that once system design is done, this phase of the process is not revisited again
- However, this is not an accurate reflection of actual development
  - Models are sometimes distinguished between being **prescriptive** and **descriptive**
  - Waterfall is meant to be prescriptive…
  - … yet it has not worked well in practice!

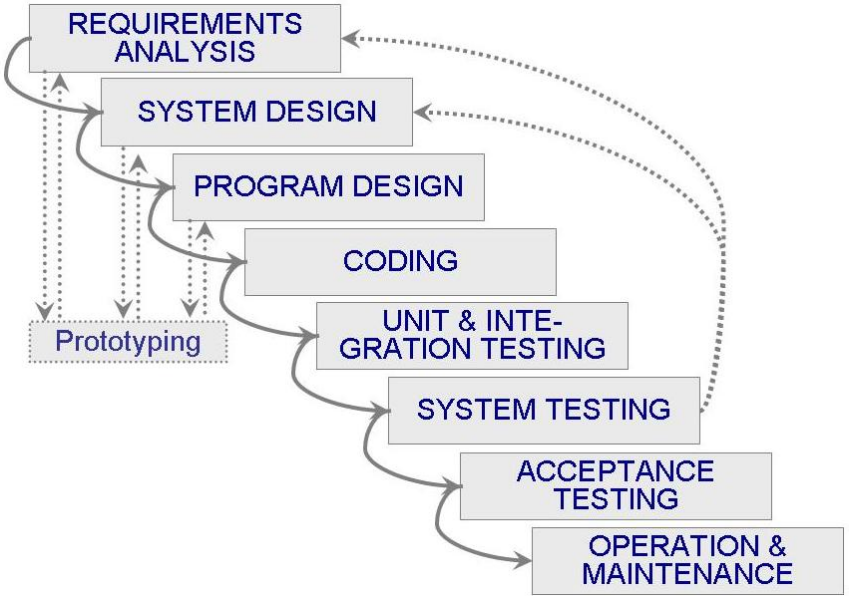


A. Waterfall model

- Provides no guidance how to handle changes to products and activities during development
  - Example: assumes requirements can be frozen, and that they are not ever modified when the customer sees a version of the system
- Views software development as a **manufacturing process** rather than as a **creative process**
- There are no iterative activities that lead to creating a final product
- Long wait before a final product
- (U.S. DoD story)

A. Modified Waterfall model

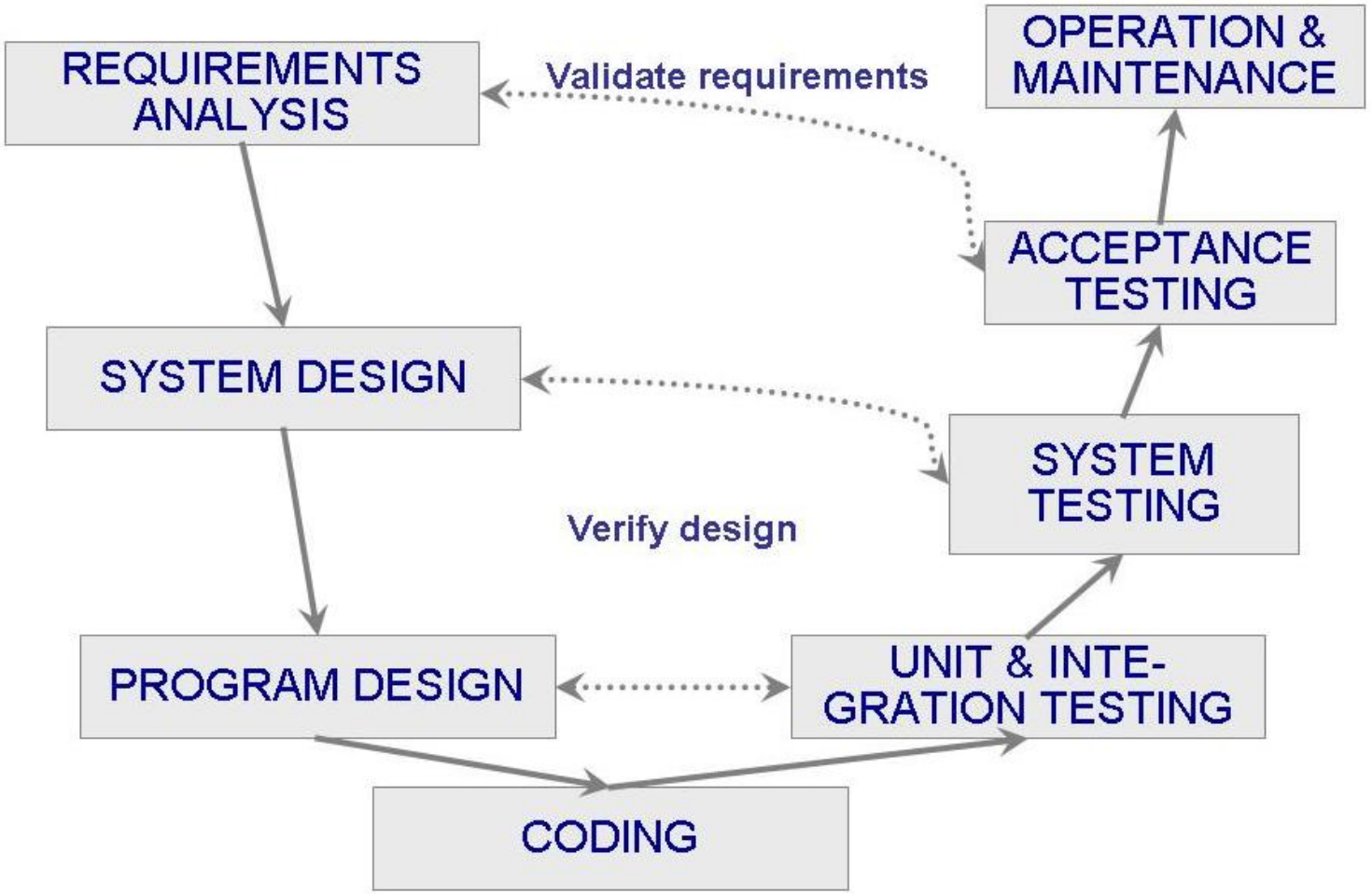
- Includes a prototyping element
- A **prototype** is a **partially developed product**
- Prototyping helps:
  - Developers **assess alternative design strategies** (design prototype)
  - **Users understand what the system will be like** (e.g., user interface prototype)
- Prototyping is useful for verification and validation
- The prototype is usually thrown away (i.e., prototype helps answer questions)



B. V Model

- Another variant of the waterfall model
- Uses **unit testing** to verify **procedural design**
- Uses **integration testing** to verify **architectural (system) design**
- Uses **acceptance testing** to validate the **requirements**
- If problems are found during verification and validation, the activities on the left side of a "V" diagram can be re-executed before testing on the right side is re-enacted

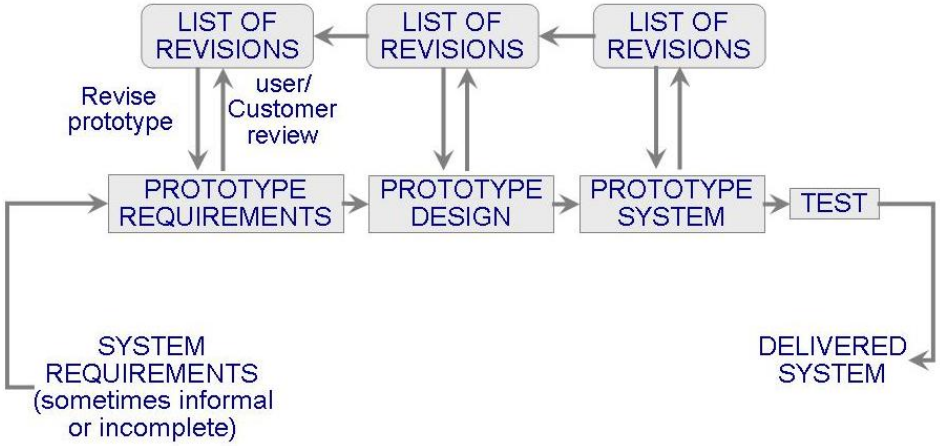
B. V Model


REQUIREMENTS ANALYSIS
Validate requirements
OPERATION & MAINTENANCE
ACCEPTANCE TESTING
SYSTEM DESIGN
SYSTEM TESTING
Verify design
PROGRAM DESIGN
UNIT & INTE-GRATION TESTING
CODING

B. V Model

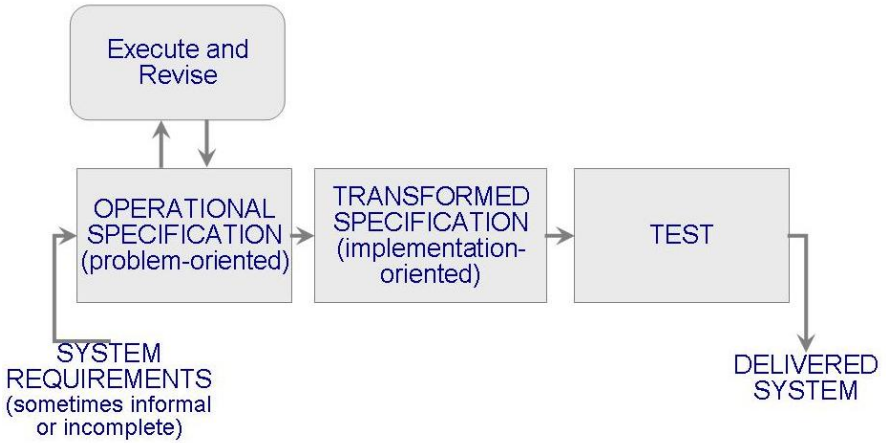
- The model makes more explicit the actual iteration present in software development
- Model was publicized/popularized through its adoption in the early 1990s by the German Ministry of Defense
  - Therefore many defense-industry participants use such a model as it is often a requirement in the tendering process
  - It has its opponents in the agile methods camp (but more about agile methods later)

C. Prototyping model



- Allows repeated investigation of the requirements or design
- Reduces risk and uncertainty in development as customer is able to verify each prototype
- If prototype is not thrown away after each iteration of the cycle, then this approach is something known as **tracer bullets**

D. Operational Specification model



- Requirements/specifications are expressed in some executable format (i.e., a specification language)
  - A flavour of this is also sometimes called "algebraic specification"
- The requirements are executed (either via a tool or by hand examination) and their implication evaluated early in the development process
- Aspects of functionality and design are – in effect – merged in this approach (unlike Waterfall where design and functionality are kept in separate phases).

Example operational specification

Air-traffic control

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfed
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector
Put - adds an aircraft to a sector with no constraint checks
In-space - checks if an aircraft is already in a sector
Occupied - checks if a specified height is available

Example operational specification

Air-traffic control

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

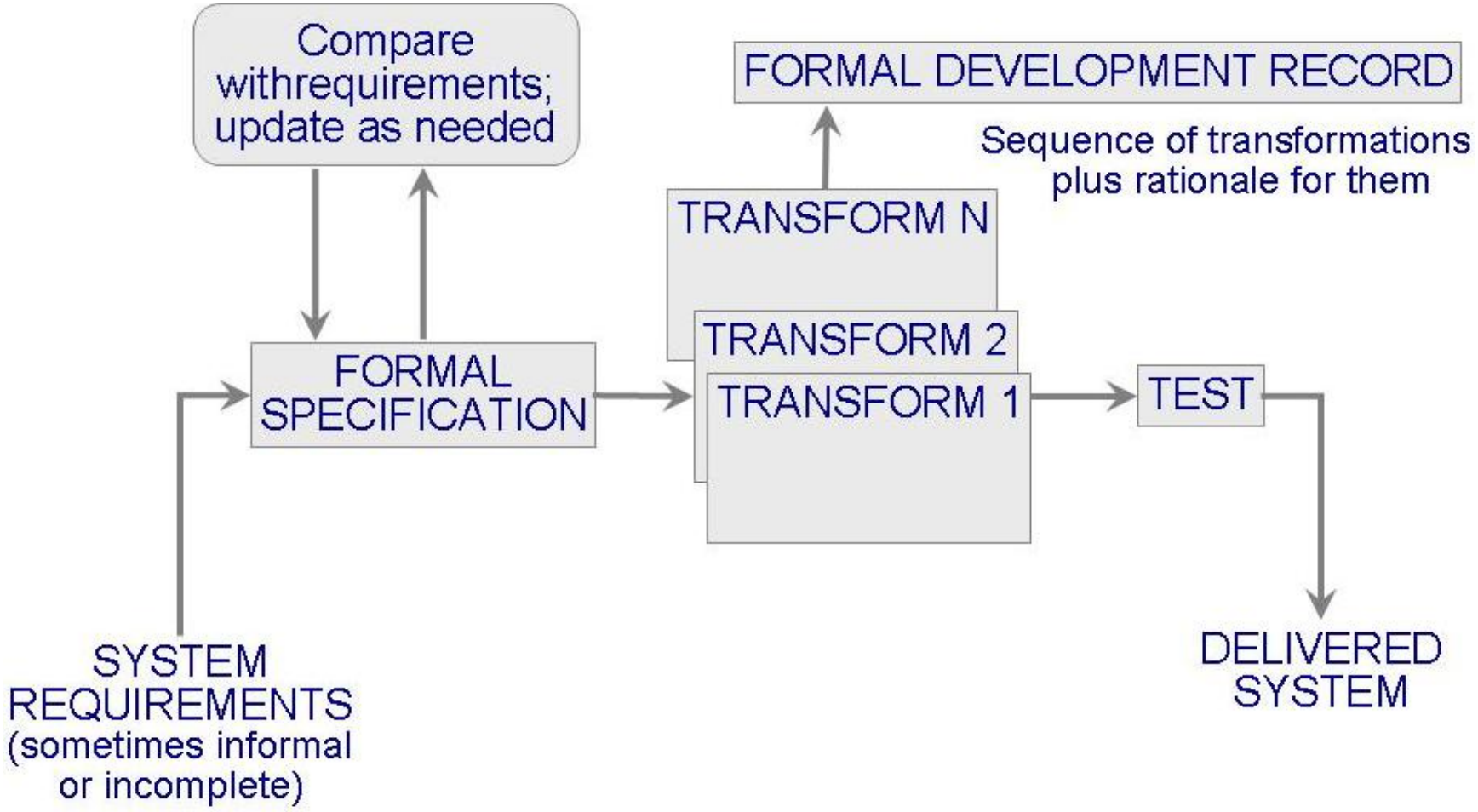
```
Enter (S, CS, H) =
    if      In-space (S, CS )  then  S exception (Aircraft already in sector)
    elsif   Occupied (S, H) then  S  exception (Height conflict)
    else    Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
     if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
     if      S = Create then Create  exception (No aircraft in sector)
    elsif   not In-space (S, CS) then S  exception (Aircraft not in sector)
    elsif   Occupied (S, H) then S exception (Height conflict)
    else    Put (Leave (S, CS), CS, H)

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) =  NO-HEIGHT  exception (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
     if CS = CS1  then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
     if      (H1 > H and H1 - H ≤ 300) or (H > H1 and  H - H1 ≤ 300)  then true
     else  Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS ) =
     if CS = CS1 then true else In-space (S, CS)
```

Air-traffic control

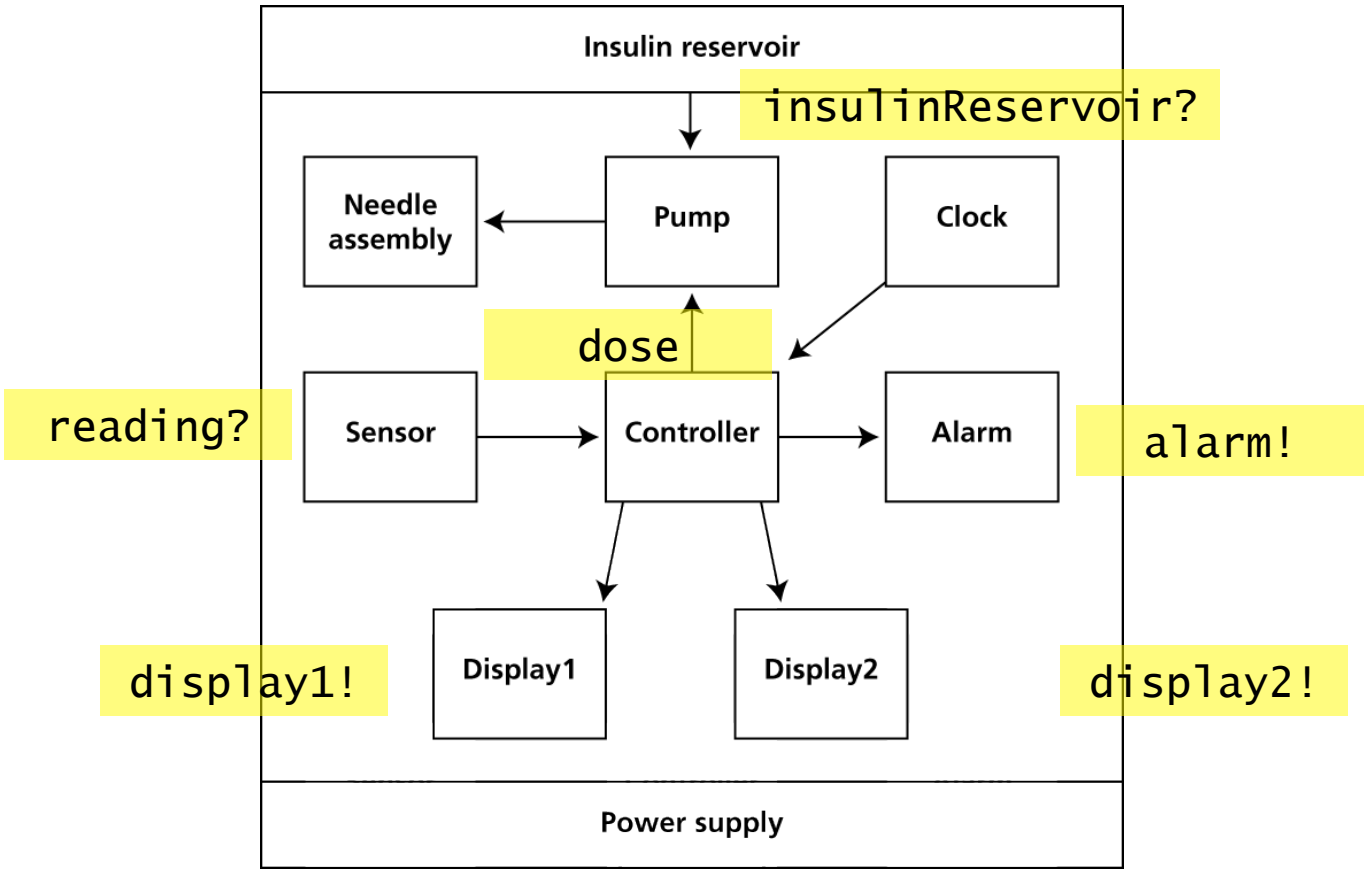
E. Transformational model

- Fewer major development steps
- Applies a series of transformations to change a specification into a deliverable system
  - Change data representation
  - Select algorithms
  - Optimize
  - Compile
- Relies on using a specific **formalism**
- Sometimes this model is referred to as **formal specification**
  - These permit specific transformations to be applied to equations in the formal specification

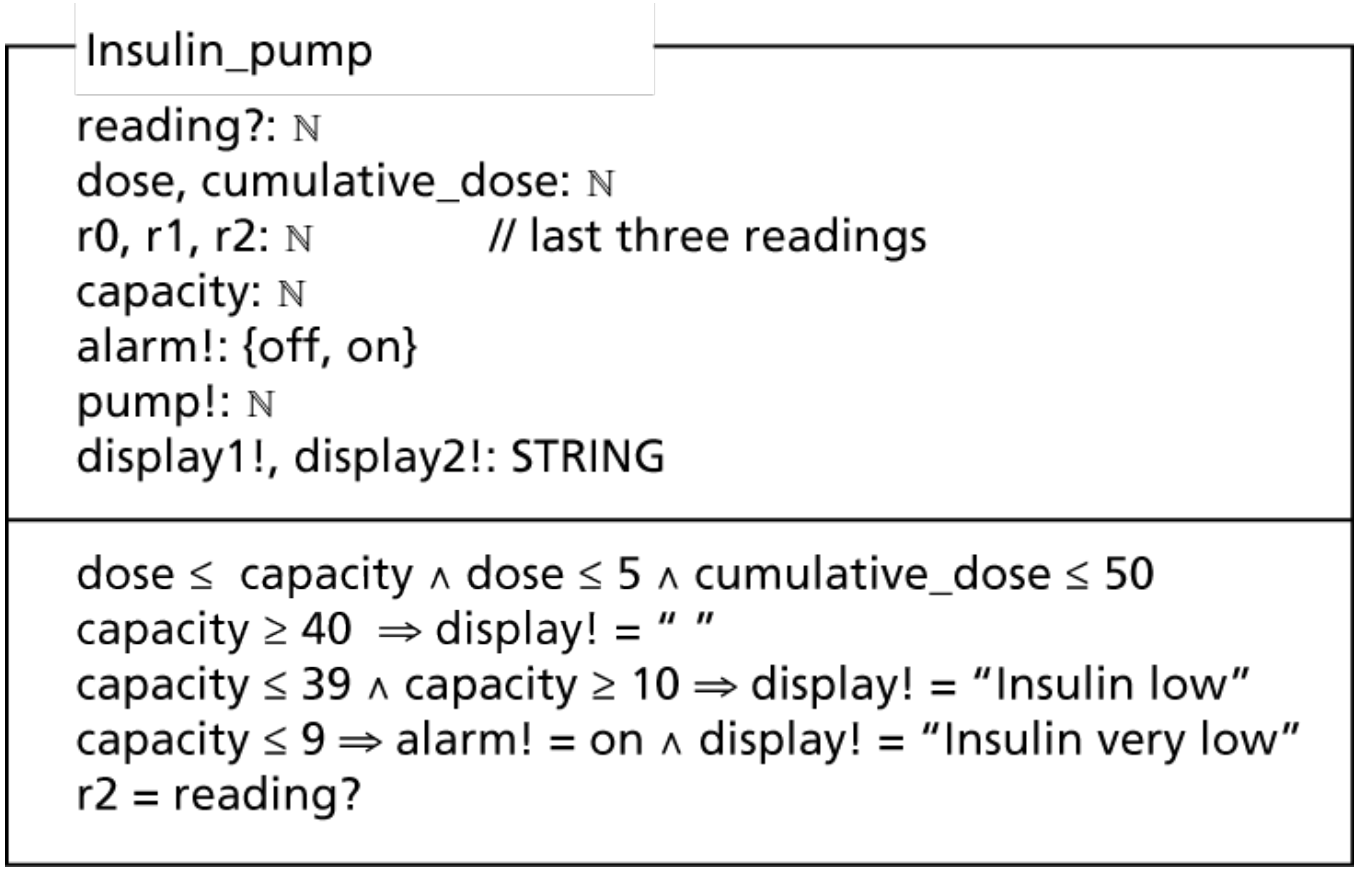
E. Transformational model



Example: Insulin pump



Example: Insulin pump schema



DOSAGE schema

DOSAGE

ΔInsulin_Pump

$$
\begin{aligned}
&(\\
&\quad dose = 0 \ \wedge \\
&\quad (\\
&\qquad ((r1 \geq r0) \wedge (r2 = r1)) \vee \\
&\qquad ((r1 > r0) \wedge (r2 \leq r1)) \vee \\
&\qquad ((r1 < r0) \wedge ((r1-r2) > (r0-r1))) \\
&\quad) \vee \\
&\\
&\quad dose = 4 \ \wedge \\
&\quad (\\
&\qquad ((r1 \leq r0) \wedge (r2 = r1)) \vee \\
&\qquad ((r1 < r0) \wedge ((r1-r2) \leq (r0-r1))) \\
&\quad) \vee \\
&\\
&\quad dose = (r2 - r1) * 4 \ \wedge \\
&\quad (\\
&\qquad ((r1 \leq r0) \wedge (r2 > r1)) \vee \\
&\qquad ((r1 > r0) \wedge ((r1-r2) \geq (r1-r0))) \\
&\quad) \\
&) \\
&\\
&capacity' = capacity - dose \\
&cumulative\_dose' = cumlative\_dose + dose \\
&r0' = r1 \wedge r1' = r2
\end{aligned}
$$

Output schemas

DISPLAY

ΔInsulin_Pump

---

display2!′ = Nat_to_string(dose) ∧
(reading? < 3 ⇒ display1!′ = “Sugar low” ∨
reading? > 30 ⇒ display1!′ = “Sugar high” ∨
reading? ≥ 3 ∧ reading? ≤ 30 ⇒ display1!′ = “OK”)

ALARM

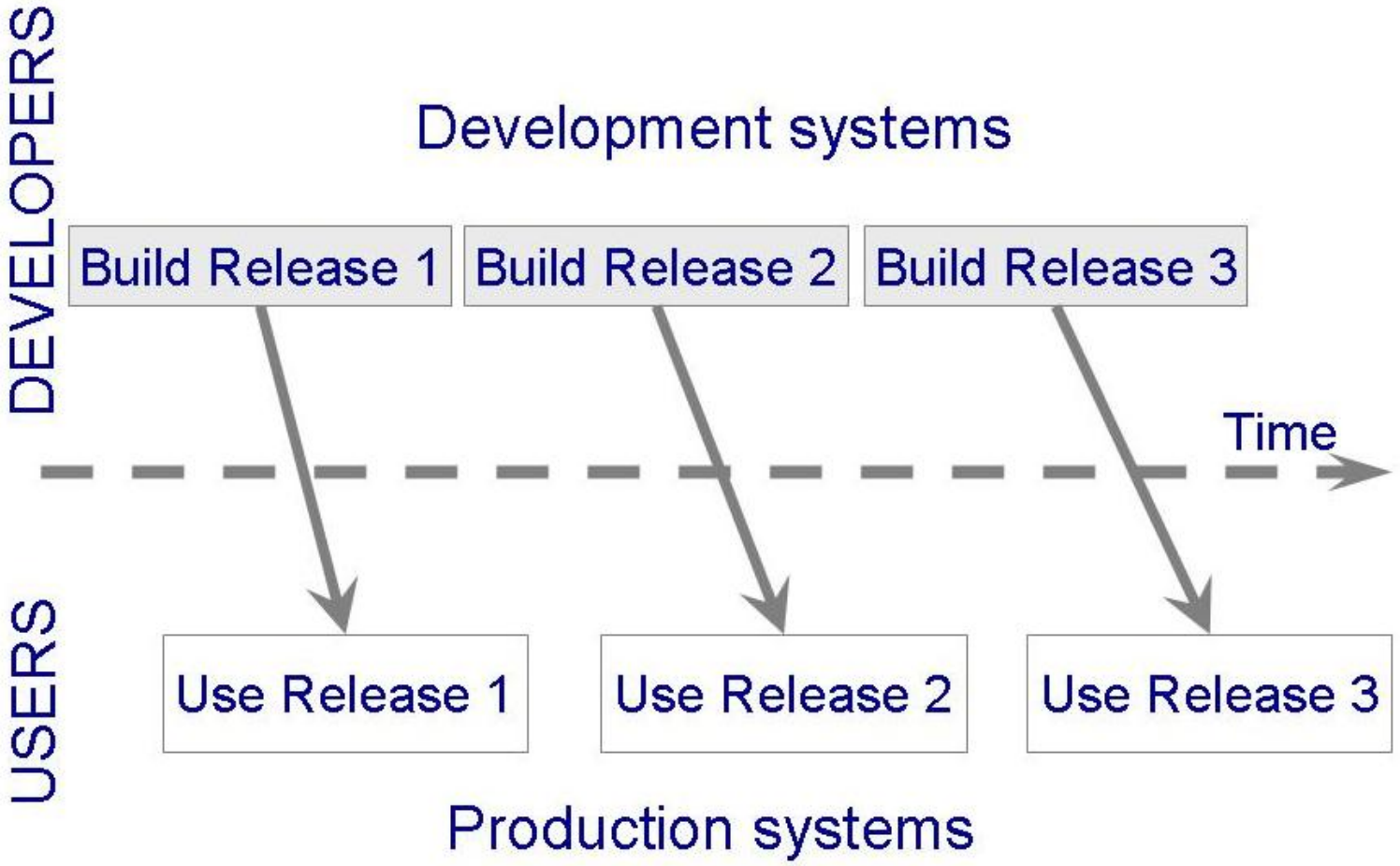
ΔInsulin_Pump

(reading? < 3 ∨ reading? > 30) ⇒ alarm!′ = on ∨
(reading? ≥ 3 ∧ reading? ≤ 30) ⇒ alarm!′ = off

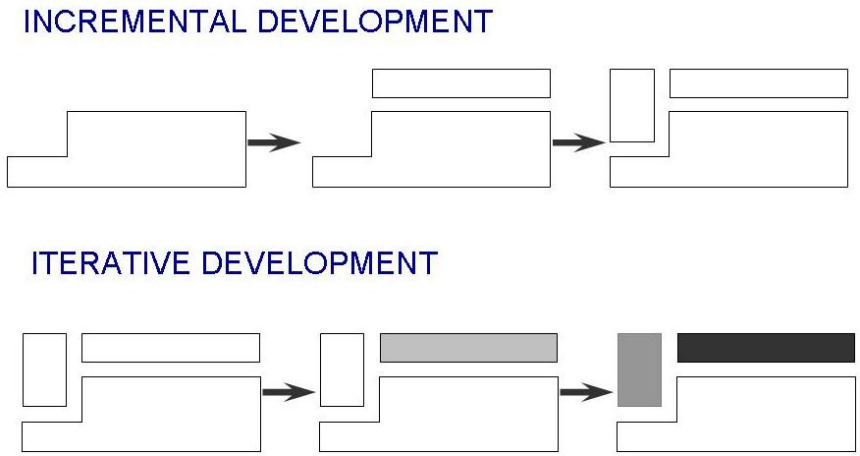
F. Incremental & Iterative

- Shorter cycle time
- System delivered in pieces
  - Enables customers to have some functionality while the rest is being developed
- Allows two systems to function in parallel
  - The production system (release n): currently being used
  - The development system (release n+1): the next version
- Has been used for over 50 years

F. Incremental & Iterative



F. Incremental vs. Iterative…



- **Incremental development**: Starts with **small functional subsystem** and **adds** functionality with each new release
- **Iterative development**: Starts with **full system albeit having minimal functionality**, then **changes** functionality of each subsystem with each new release

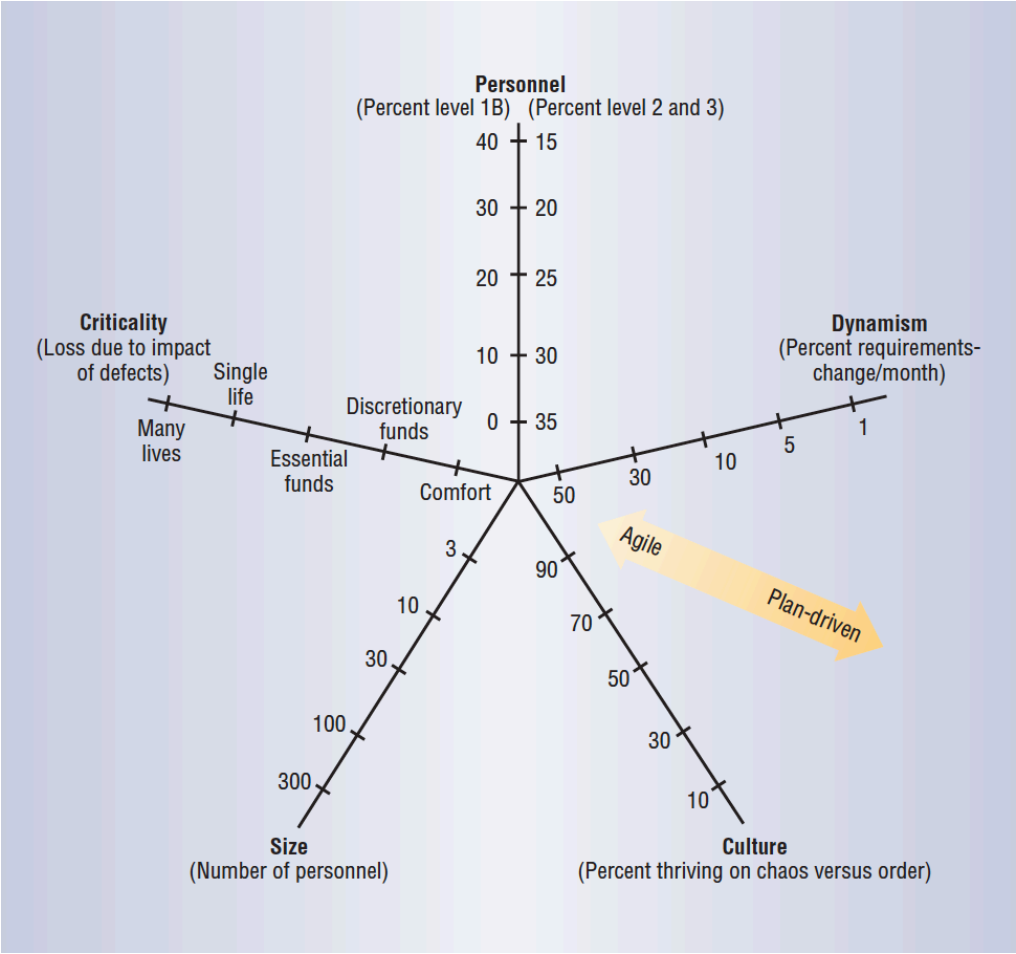
F. Incremental & Iterative

- This phased development is desirable for several reasons
  - Training can begin early, even though some functions are missing
  - Markets can be created early for functionality that has never before been offered
  - Frequent releases allow developers to fix unanticipated problems globally and quickly
  - The development team can focus on different areas of expertise with different releases

G. Spiral Model

- Suggested by Barry Boehm in the late 1980s
- Combines **development activities** with **risk management** to minimize and help control risks present in software-development projects
- The model is presented as a spiral in which each iteration is represented by a circuit around four major activities
  - Plan
  - Determine goals, alternatives, and constraints
  - Evaluate alternatives and risks
  - Develop and test
- What do we mean by **risk** in the context of software development?

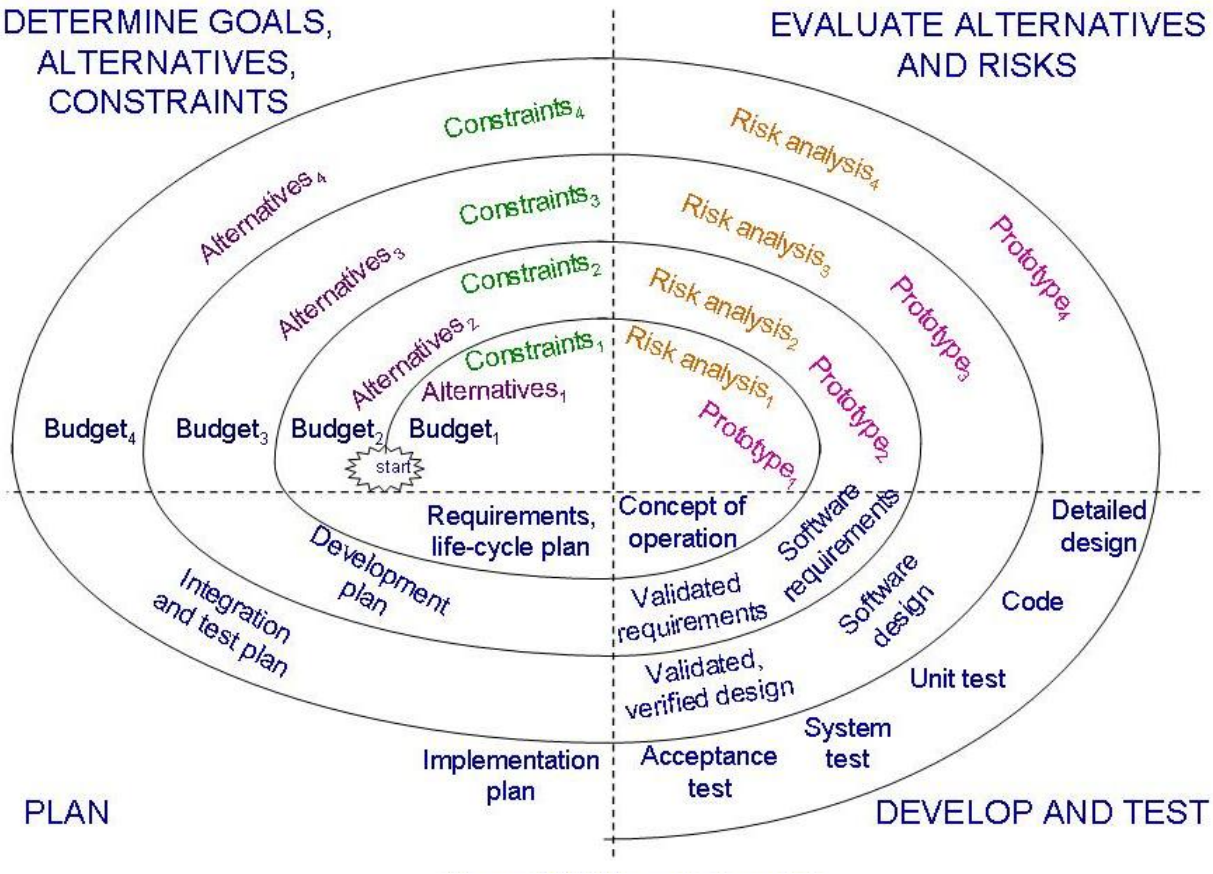
(An aside about development risks)



Cockburn's Three levels of Software Understanding (revised by Boehm)

- 3: Able to revise a method, breaking its rules to fit an unprecedented new situation
- 2: Able to tailor a new method to fit a previously-seen situation.
- 1A, 1B: With training, able to perform some discretionary or procedural steps (or both)
- -1: May have technical skills, but unable or unwilling to collaborate or follow shared methods.

G. Spiral model



H. Agile methods

- Emphasis on flexibility in producing software quickly and capably
- **Agile Manifesto**
  - Value individuals and interactions over process and tools
  - Prefer to invest time in producing working software rather than in producing comprehensive documentation
  - Focus on customer collaboration rather than contract negotiation
  - Concentrate on responding to change rather than on creating a plan and then following it regardless of what happens

H. Agile methods

- **Extreme programming** (XP) is one flavour of Agile methods
- **Crystal**: a collection of approaches based on the notion that every project needs a unique set of policies and conventions
- **Scrum**: Seven- to 30-day iterations; multiple self-organizing teams; daily “scrum” coordination
- **Adaptive software development** (ASD): repeating series of "speculate", "colloborate" and "learn" cycles.

H. Agile methods

- Emphasis on **four characteristics of agility**
  - **Communication**: Continual interchange between customers and developers
  - **Simplicity**: Select the simplest design or implementation
  - **Courage**: Commitment to delivering functionality early and often
  - **Feedback**: Loops built into the various activities during the development process

H. Agile methods

- The planning game: customer defines value
- Small releases
- Shared metaphors: common vision, common names
- Simple design
- Writing tests first
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration: small increments
- Sustainable pace: 40 hours/week
- On-site customer
- Coding standards

H. Agile methods: concerns

- Extreme programming's practices are interdependent
  - A vulnerability if one of them is modified
- Requirements expressed as a set of test cases must be passed by the software
  - System passes the tests but a "test-passing system" is not what the customer is paying for
- Refactoring is an issue
  - Difficult to rework a system without degrading its architecture

Summary

- Process development involves activities, resources, and product
- Process model includes organizational, functional, behavioral, and other perspectives
- A process model is useful for guiding team behavior, coordination, and collaboration

Colophon

- Some slides based on Pfleeger & Atlee, "Software Engineering: Theory and Practice" © 2006 Prentice Hall
- Everything else: © 2010 Michael Zastre, University of Victoria