SSW-540: Fundamentals of Software Engineering

Software Construction and Modeling

Dr. Richard Ens
School of Systems and Enterprises

A Python *Pearl*

Many of you created great comparing functions in your last script, but you didn’t really need to. Python has a built-in function call **max** that finds the maximum value of the entries passed to it.

The **max** function works with any type of input so that:

6 is the maximum of **max(**5, 6, 3, 2**)**

5.7 is the maximum of **max(**5.2, 5.7, 5.4**)**

abigail is the maximum of **max(**"abigail", ”Josephine", "George"**)**

josephine is the maximum of **max(**"abigail", ”josephine", "George"**)**

What does that tell you about evaluating upper and lower case letters?

Python Pointers

How you interpret input mathematically can make a difference in your calculations. When asking for a temperature, one expects and may even check for an integer as input. But the calculations for temperature conversions are floating point calculations.

If 33 degrees Fahrenheit is converted to Celsius as an integer:

```
Fahr = int(input("Enter a Fahrenheit temperature: ")
Cels = (Fahr – 32) * 5/9
```

The answer one gets is 0 degrees.

But when it is converted as a floating point number:

```
Fahr = float(input("Enter a Fahrenheit temperature: ")
Cels = (Fahr – 32) * 5/9
```

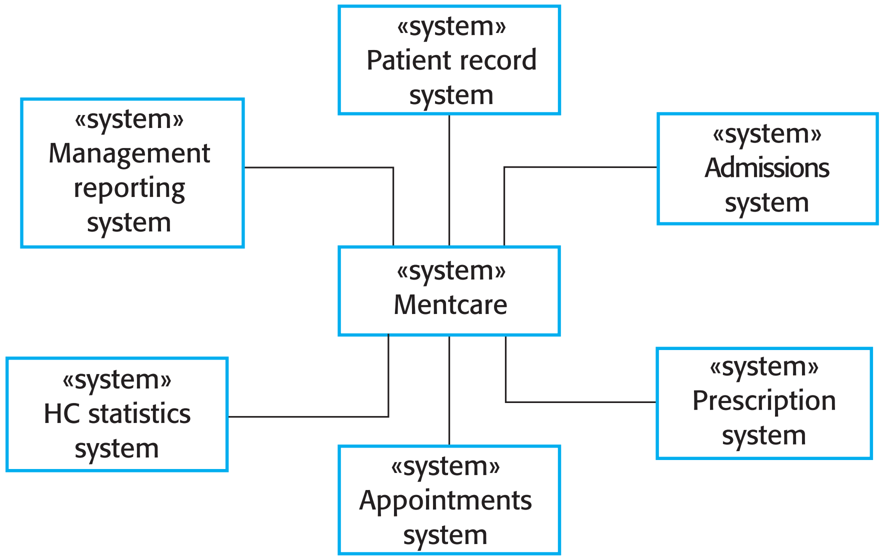
The answer is 0.556, which rounds up to 1 degree.

Software system modeling

- Software system modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- Modeling helps resolve ambiguities in and insure completeness of requirements; helps find problems early in the life-cycle
- We can use graphical modeling to
  - facilitate discussion about an existing or proposed software system (models need not be complete or completely accurate for this)
  - document an existing software system (models need not be complete but should be accurate)
  - provide a detailed software description that can be used to generate a system implementation (models must be complete and accurate!)
- For software, the graphical notation most often used is Unified Modeling Language (UML).

Context models

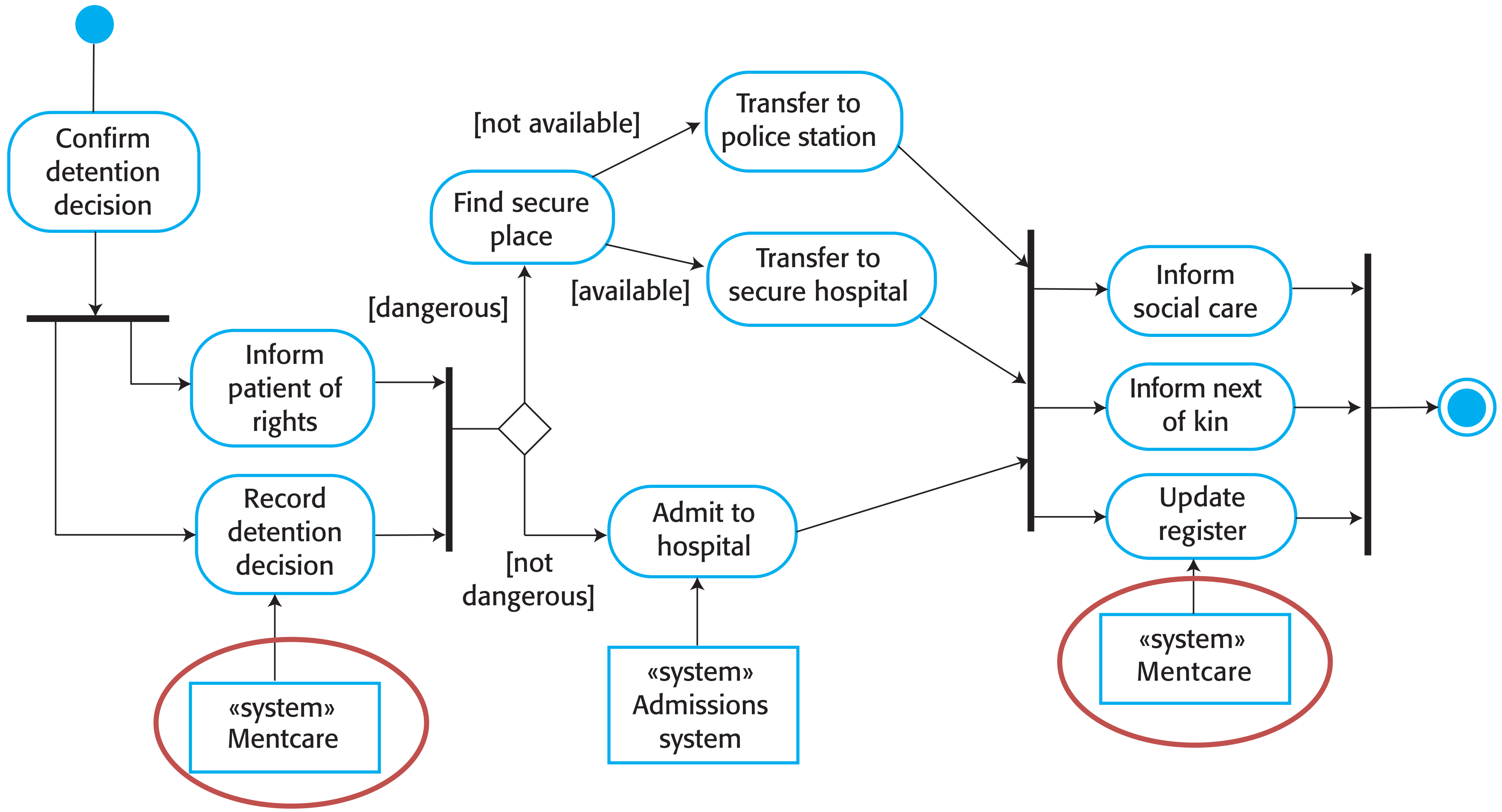
- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organizational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems that are used or depend on the system being developed


«system» Patient record system
«system» Management reporting system
«system» Admissions system
«system» Mentcare
«system» HC statistics system
«system» Prescription system
«system» Appointments system

Process perspective

- ✧ Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- ✧ Process models reveal how the system being developed is used in broader business processes.
- ✧ UML activity diagrams are often used to define business process models.
- ✧ Such models bridge the context and the interaction perspectives.

Process model of involuntary detention (an activity diagram)



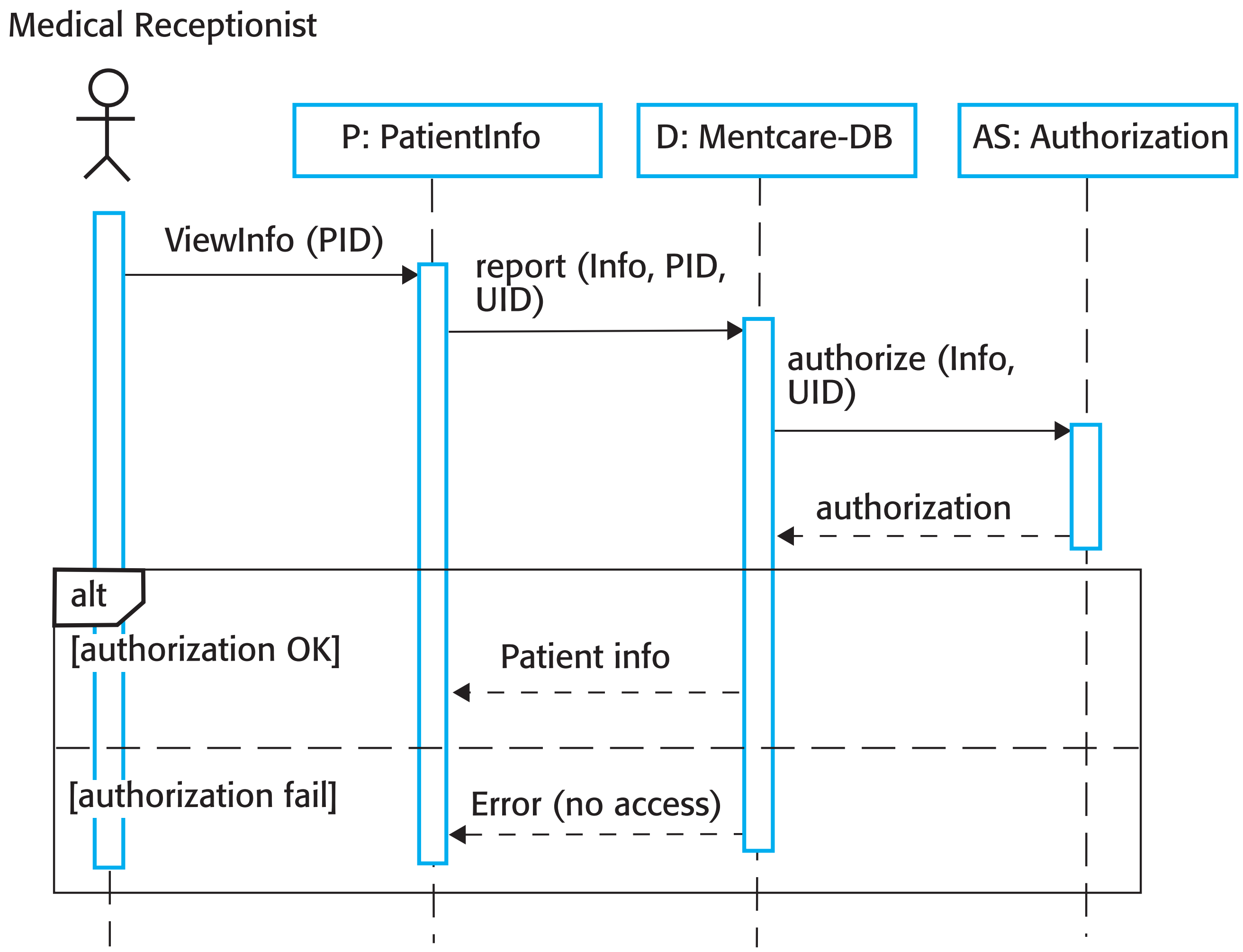
Interaction models

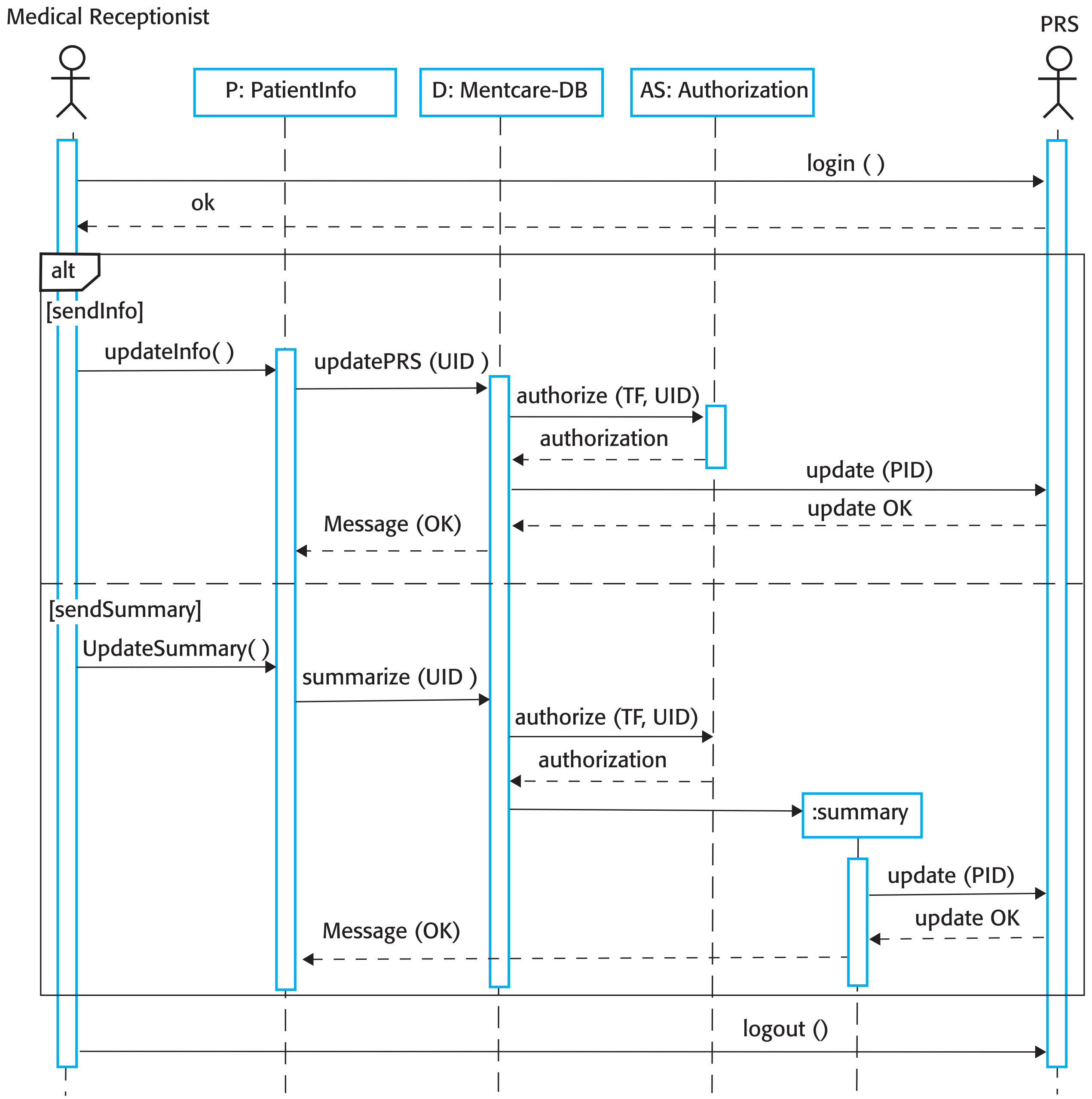
- ✧ Modeling user interaction is important as it helps to identify user requirements.
- ✧ Modeling system-to-system interaction highlights the communication problems that may arise.
- ✧ Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- ✧ Use case diagrams and sequence diagrams may be used for interaction modeling.
- ✧ We introduced use case diagrams last week; let's look at sequence diagrams.

Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions among the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. When interactions involve an object or actor, the dotted line becomes a rectangle, called a “lifeline” (*the actor/object comes to life*).
- Interactions between objects are indicated by annotated arrows.

Sequence diagram for the View Patient Information use case





Sequence diagram for the Transfer Data use case

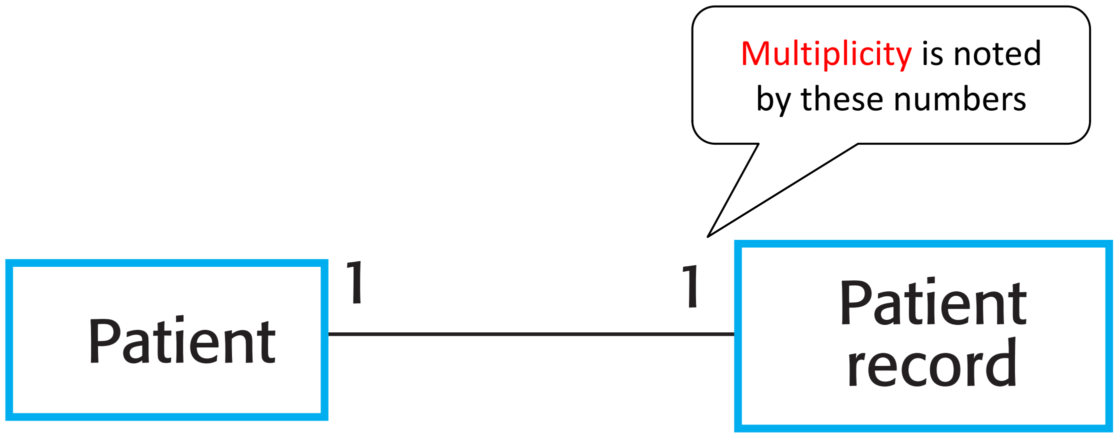
Structural models

- ✧ Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- ✧ Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- ✧ You create structural models of a system when you are discussing and designing the system architecture.

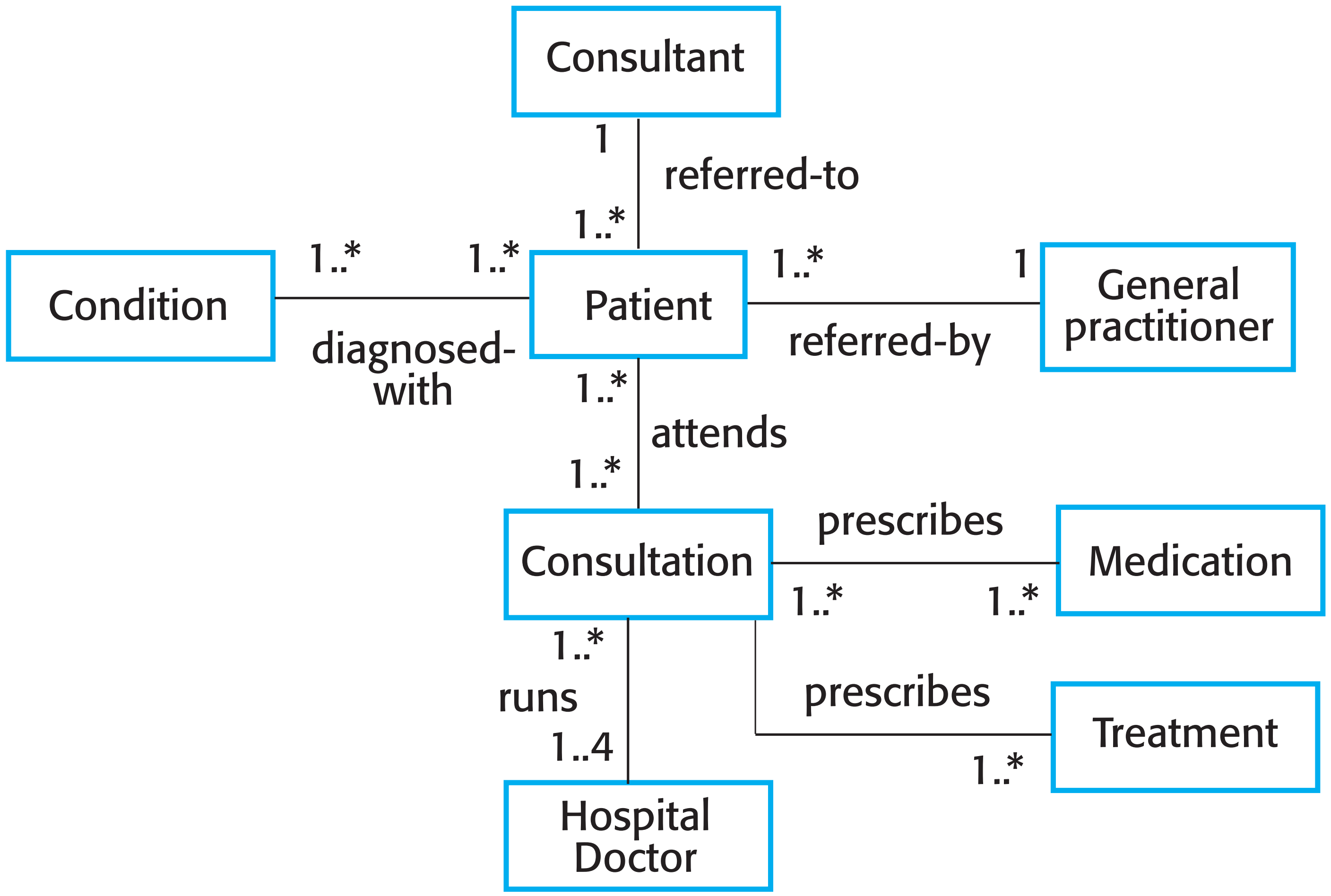
Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects often represent something in the real world, such as a patient, a prescription, doctor, etc.

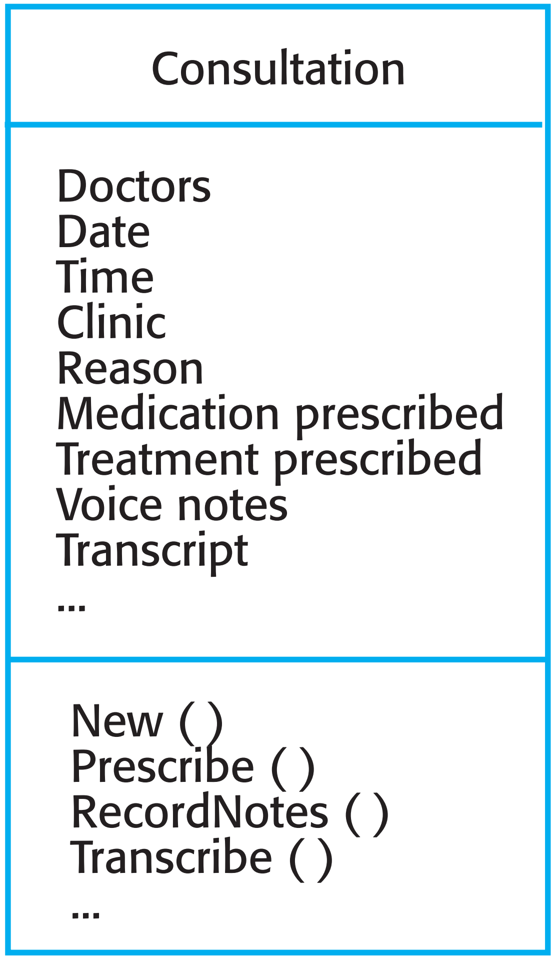
UML classes and association



Classes and associations in the MHC-PMS



The Consultation class



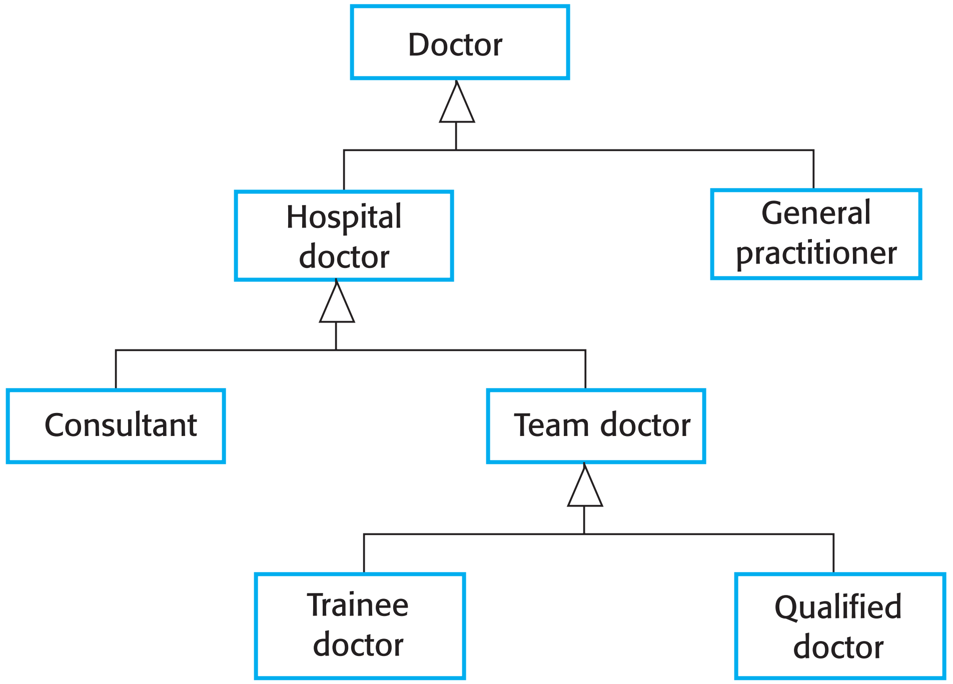
Generalization

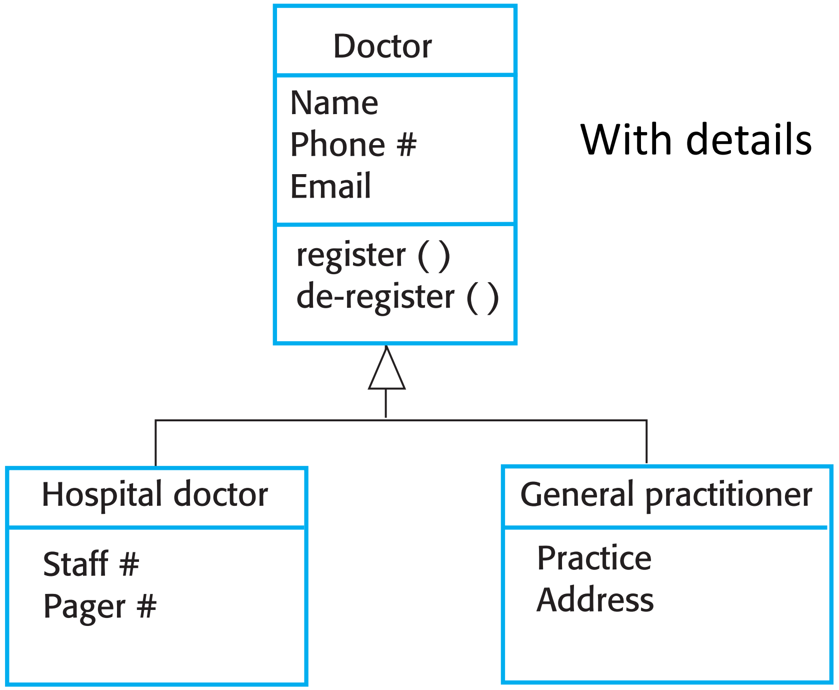
- Generalization is an everyday technique that we use to manage complexity.
- Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

Generalization

- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The lower-level classes are subclasses that inherit their attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy

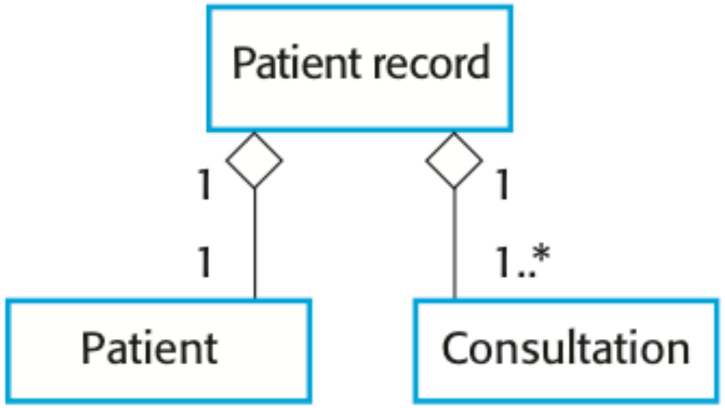




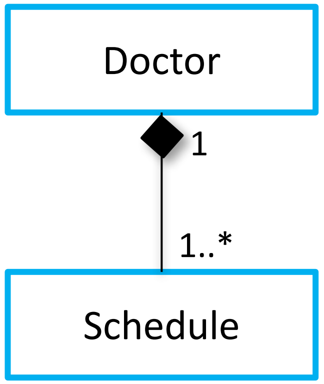
With details

Object class aggregation models

- An aggregation model shows how classes can be composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.



- Composition is a form of aggregation.
- Compositions are whole-part relationships where the lifetime of the part is dependent on the lifetime of the whole.

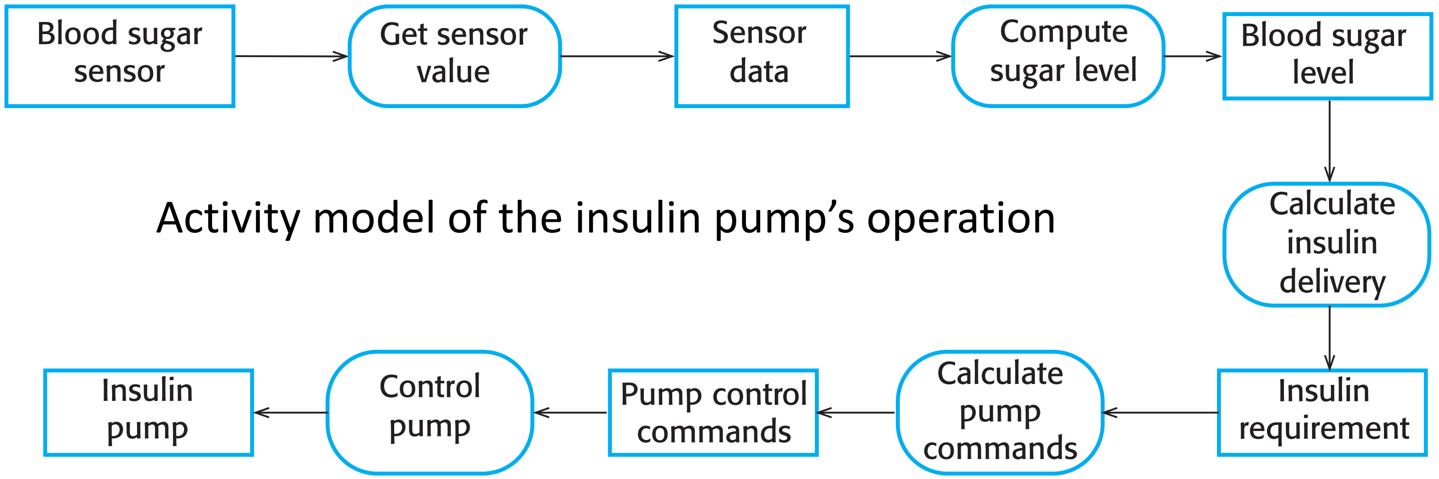


Behavioral models

- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- You can think of these stimuli as being of two types:
  - Data Some data arrives that has to be processed by the system.
  - Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

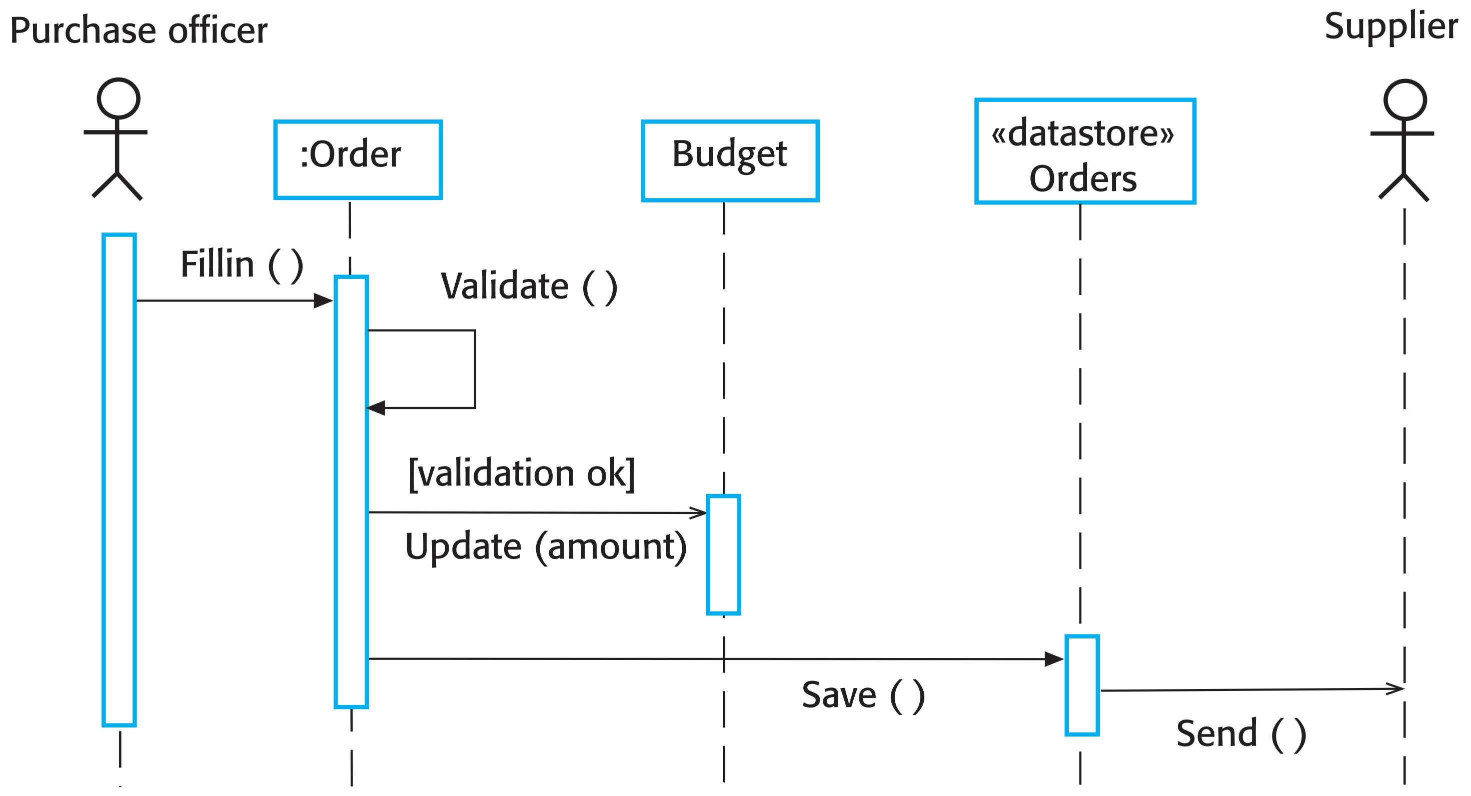
Data-driven modeling

- Many systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.



Activity model of the insulin pump's operation

Order processing sequence diagram



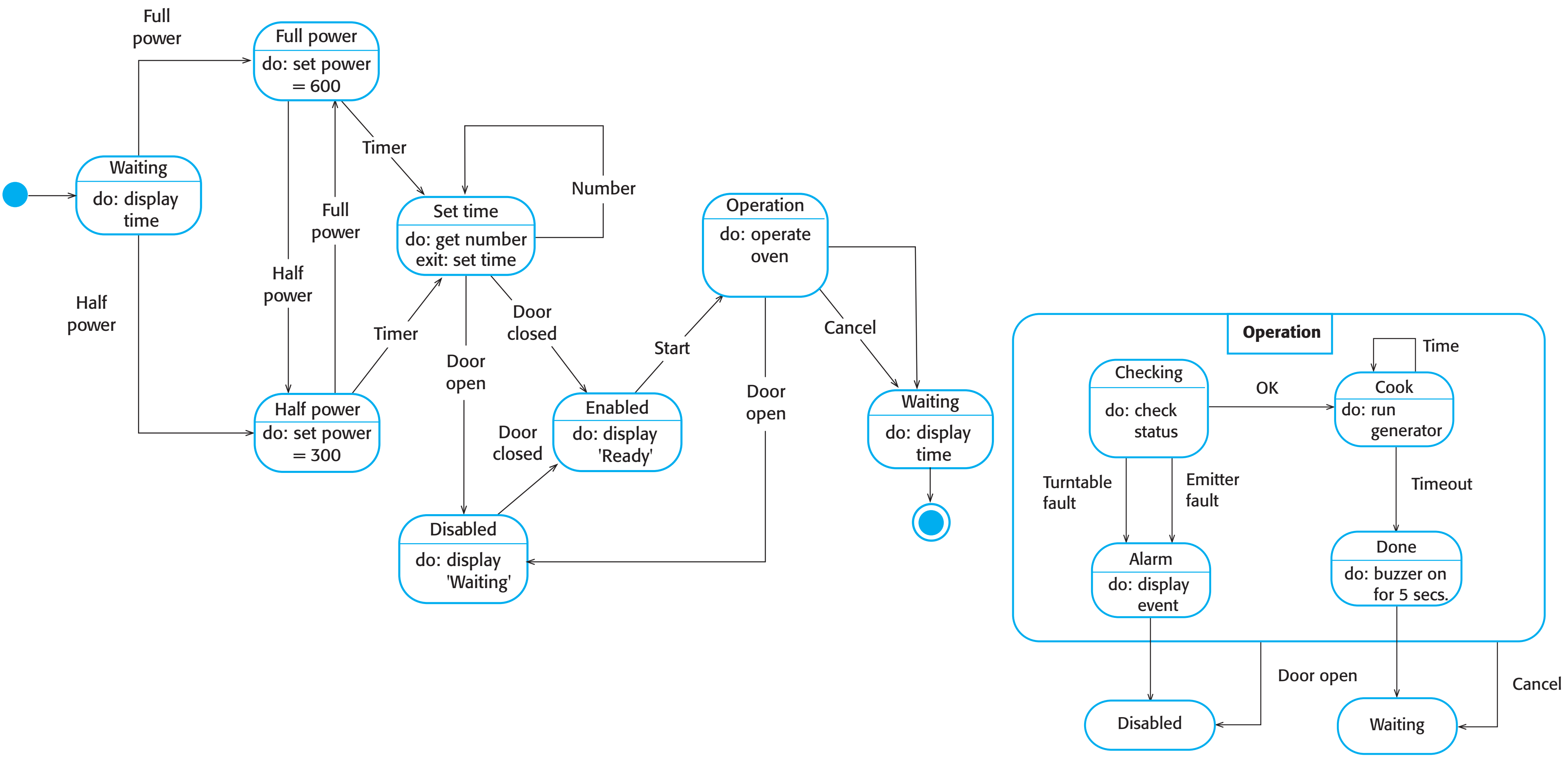
Event-driven modeling

- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone system responds to events such as ‘receiver off hook’ by generating a dial tone and a mobile phone system responds to events such as ‘message arrival’ by playing an alert tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

- ✧ These model the behaviour of the system in response to external and internal events.
- ✧ They show the system’s responses to stimuli so are often used for modelling real-time systems.
- ✧ State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- ✧ Statecharts are an integral part of the UML and are used to represent state machine models.

State diagrams – microwave oven



States and stimuli for the microwave oven

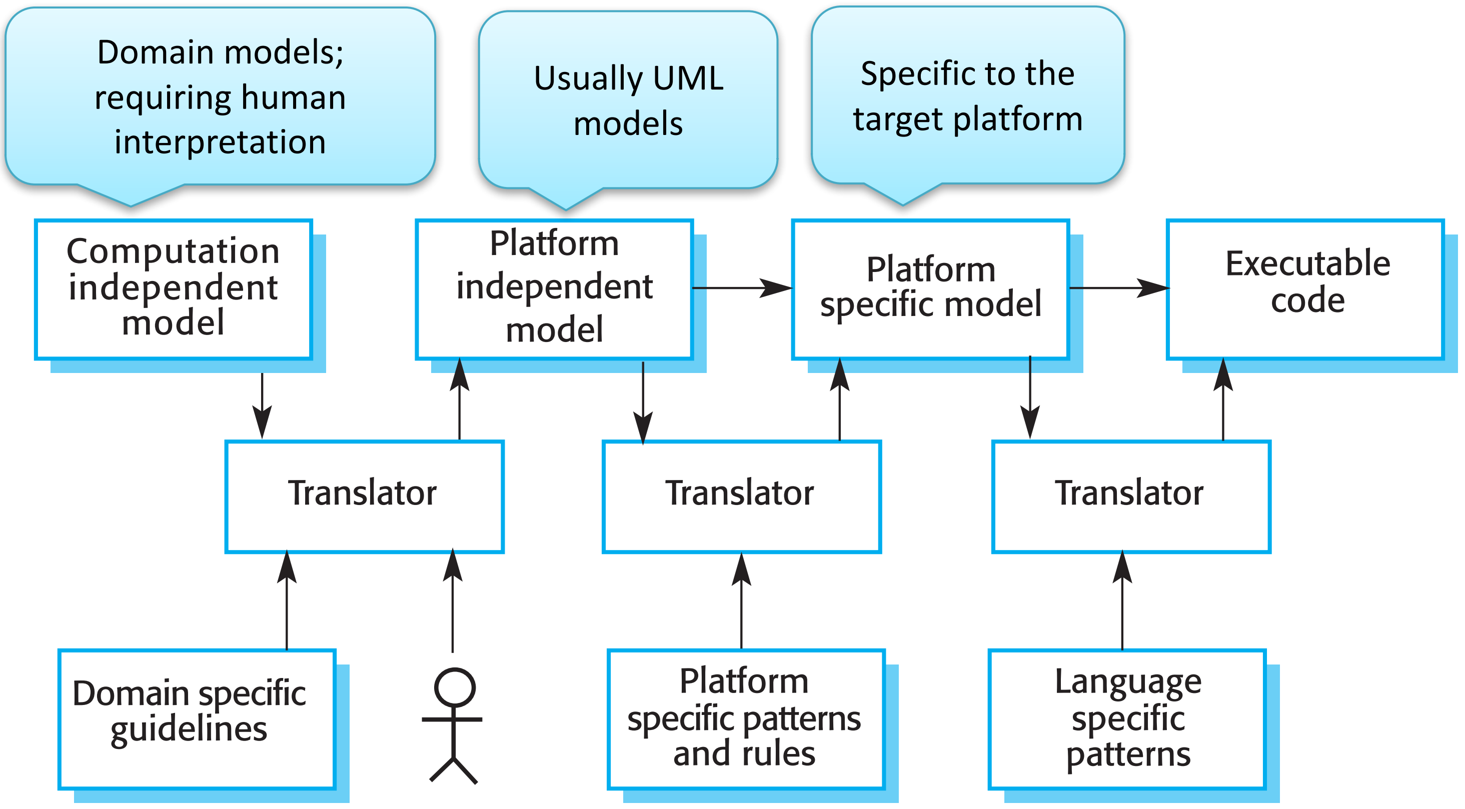
State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Model-driven software engineering

- Model-driven software engineering (MDSE) is an approach to software development in which the system is represented as a set of models that can be automatically transformed to executable code.
- In MDSE, models rather than programs are the principal outputs of the development process.
- Proponents of MDSE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.
- Model-driven architecture (MDA) is the approach taken by most adopters of MDSE

MDA transformations


Domain models; requiring human interpretation
Usually UML models
Specific to the target platform
Computation independent model
Platform independent model
Platform specific model
Executable code
Translator
Translator
Translator
Domain specific guidelines
Platform specific patterns and rules
Language specific patterns

Agile methods and MDA

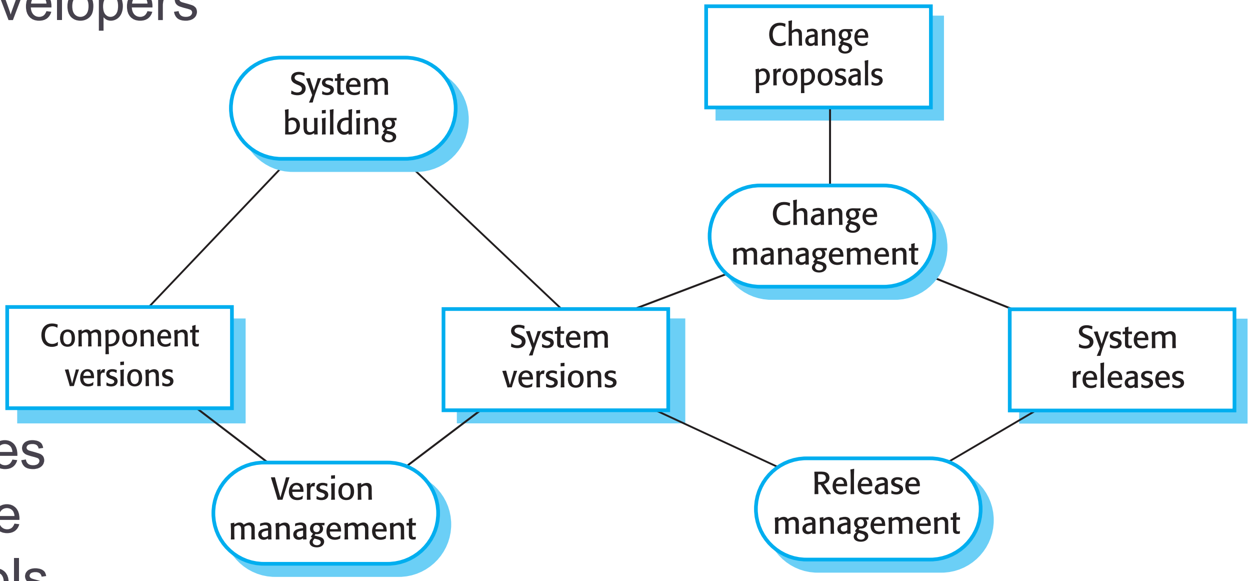
- The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- But, the notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto; few agile developers feel comfortable with model-driven engineering.
- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process; no separate coding would be required.

Adoption of MDA

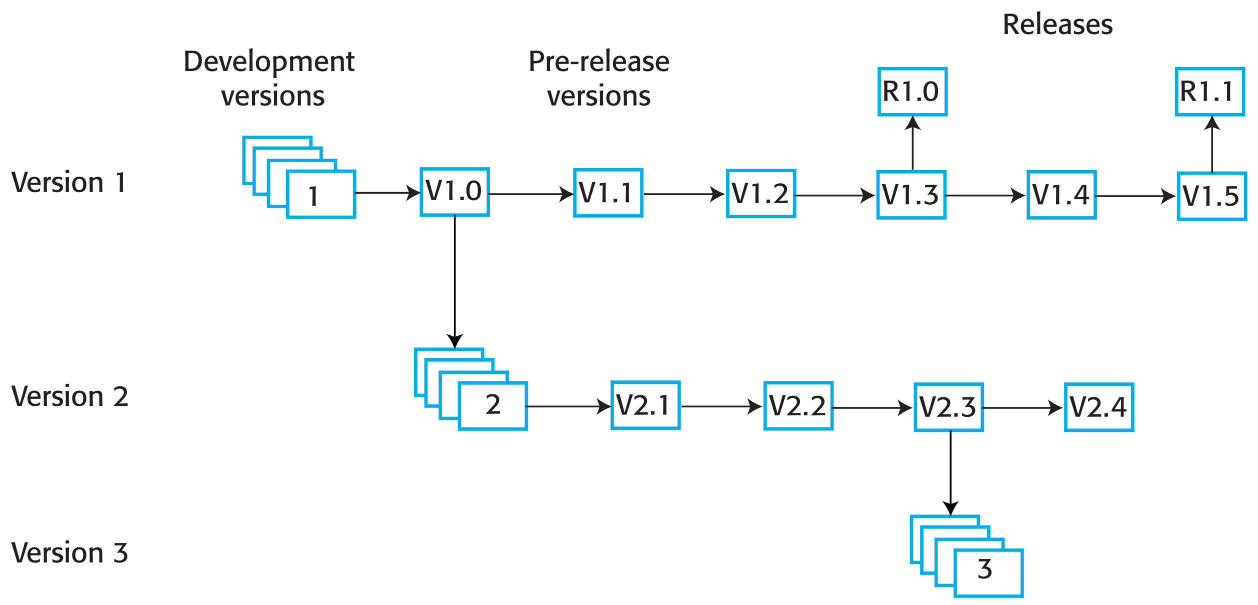
- For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.
- The arguments for platform-independence are only valid for large, long-lifetime systems. For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
- Specialized tool support is required to convert models from one level to another
  - There is limited tool availability and organizations may require tool adaptation and customization to their environment
  - For long-lifetime systems that could be developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business

Configuration management

- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems
- It is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers
- We use tools to support CM
- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.



Multi-version systems



- For large systems, there is never just one 'working' version of a system.
- There are always several versions of the system at different stages of development (in development, in system test, in release)
- There may be several teams involved in the development of different system versions.

CM terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. It is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Mainline	A sequence of baselines representing different versions of a system.

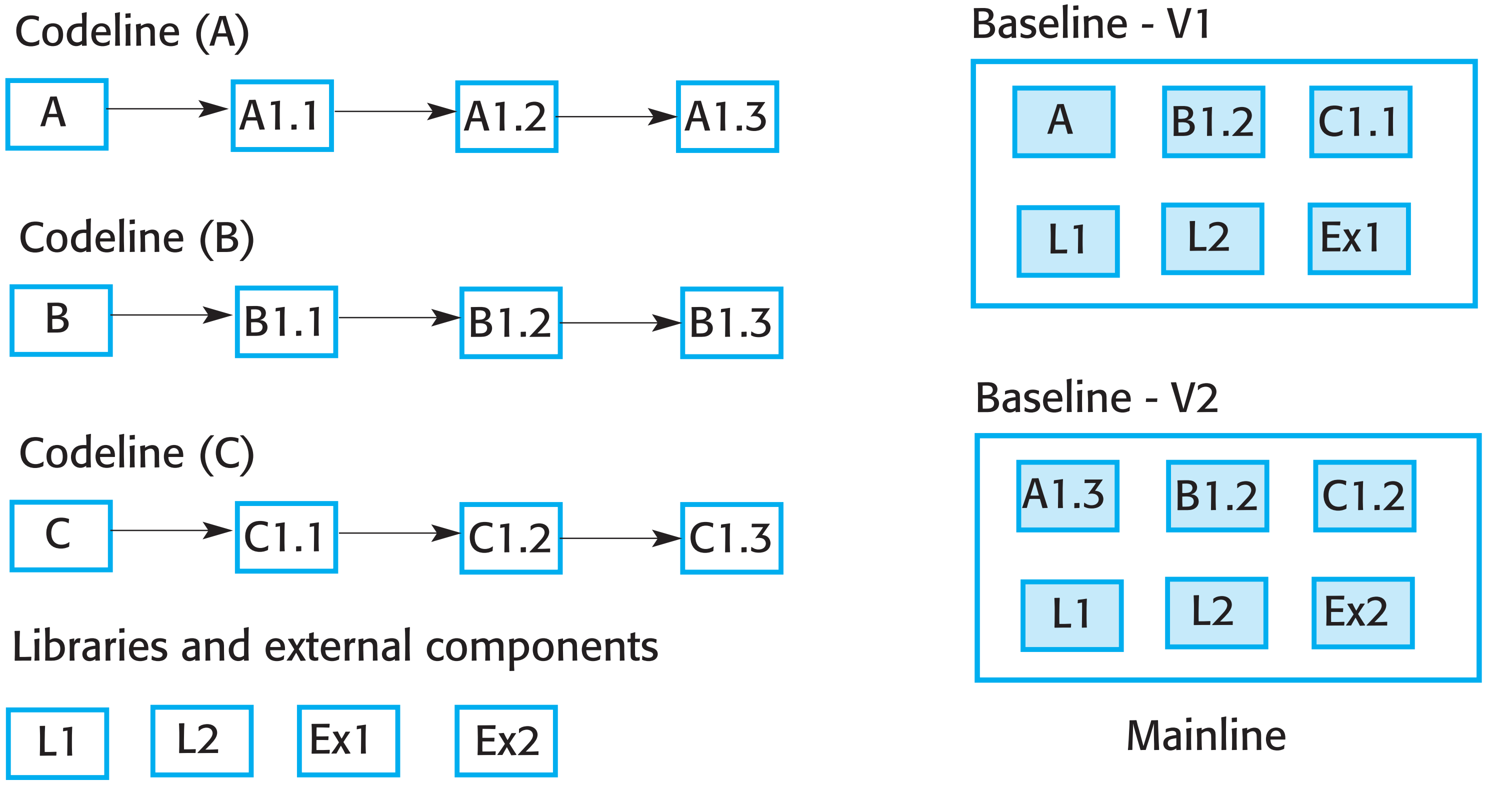
CM terminology

Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Version management

- Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing codelines and baselines.

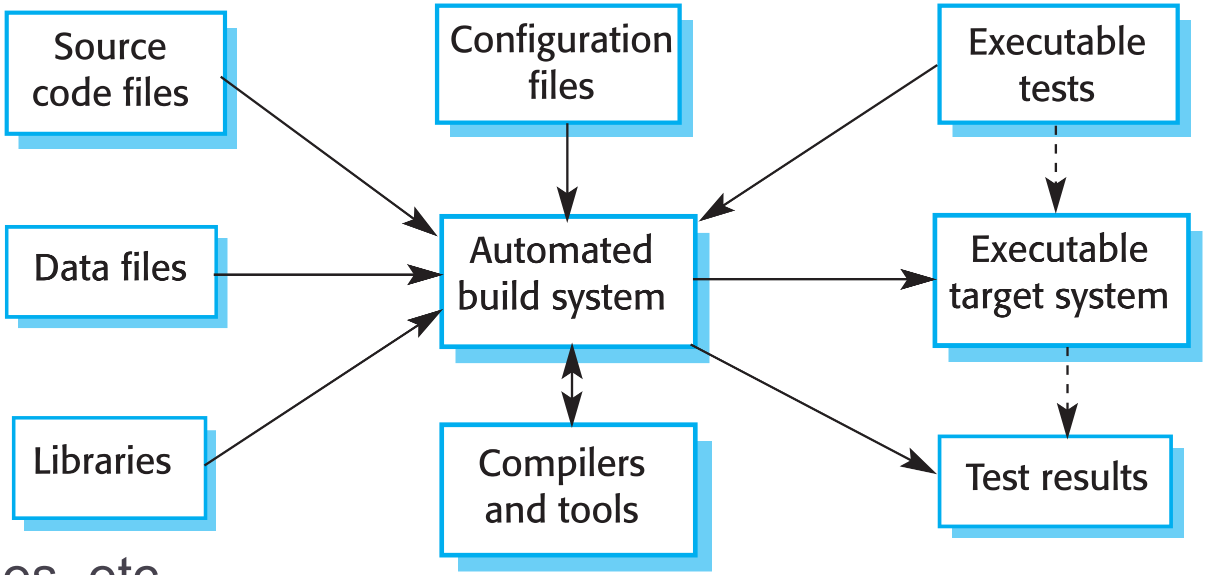
Codelines and baselines



Version control systems

- Version control (VC) systems identify, store and control access to the different versions of components.
  - Centralized systems have a single master repository that maintains all versions of the software components that are being developed. E.g., Subversion.
  - Distributed systems, where multiple versions of the component repository exist at the same time. E.g., Git and GitHub
- Key features of version control systems include
  - Version and release identification
  - Change history recording
  - Support for independent development (including branching and merging)
  - Project support
  - Storage management

System building

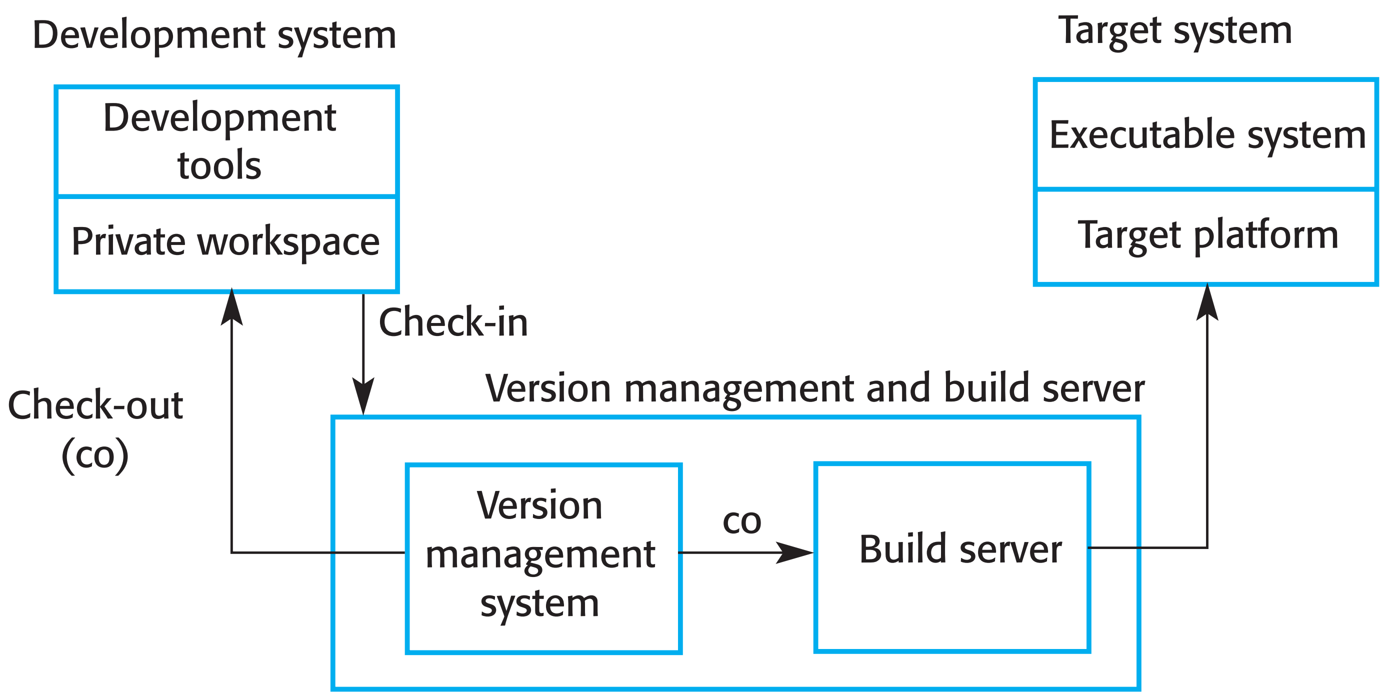


- ✧ System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- ✧ System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- ✧ The configuration description used to identify a baseline is also used by the system building tool.

Build system functionality

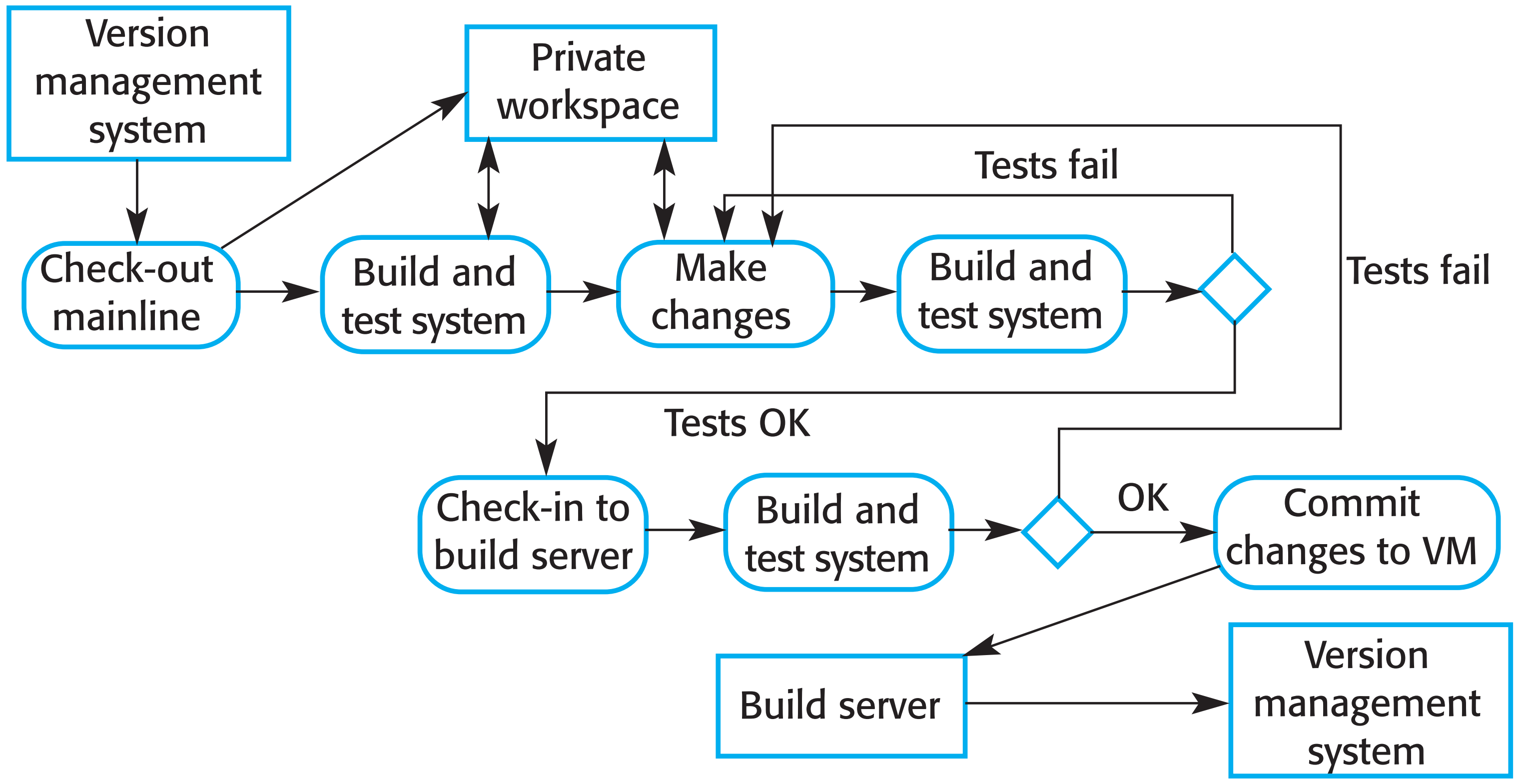
- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

Development, build, and target platforms



For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

Continuous integration (Agile building)



Release management

- A release is a version of a software system distributed to customers.
- For generic software, there are usually major releases that deliver significant new functionality, and minor releases that repair bugs and fix customer problems that have been reported.
- For custom software or software product lines, releases of the system may have to be produced for each customer and customers may be running several different releases at the same time.
- Factors influencing release planning include
  - Competition
  - Marketing requirements
  - Platform changes
  - Technical quality of the system

Release components

- As well as the the executable code of the system, a release may also include:
  - configuration files defining how the release should be configured for particular installations;
  - data files, such as files of error messages, that are needed for successful system operation;
  - an installation program that is used to help install the system on target hardware;
  - electronic and paper documentation describing the system;
  - packaging and associated publicity that have been designed for that release.
- All of these go out with the release!

Release creation

- The executable code of the programs and all associated data files must be identified in the version control system.
- Configuration descriptions may have to be written for different hardware and operating systems.
- Update instructions may have to be written for customers who need to configure their own systems.
- Scripts for the installation program may have to be written.
- Web pages have to be created describing the release, with links to system documentation.
- When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

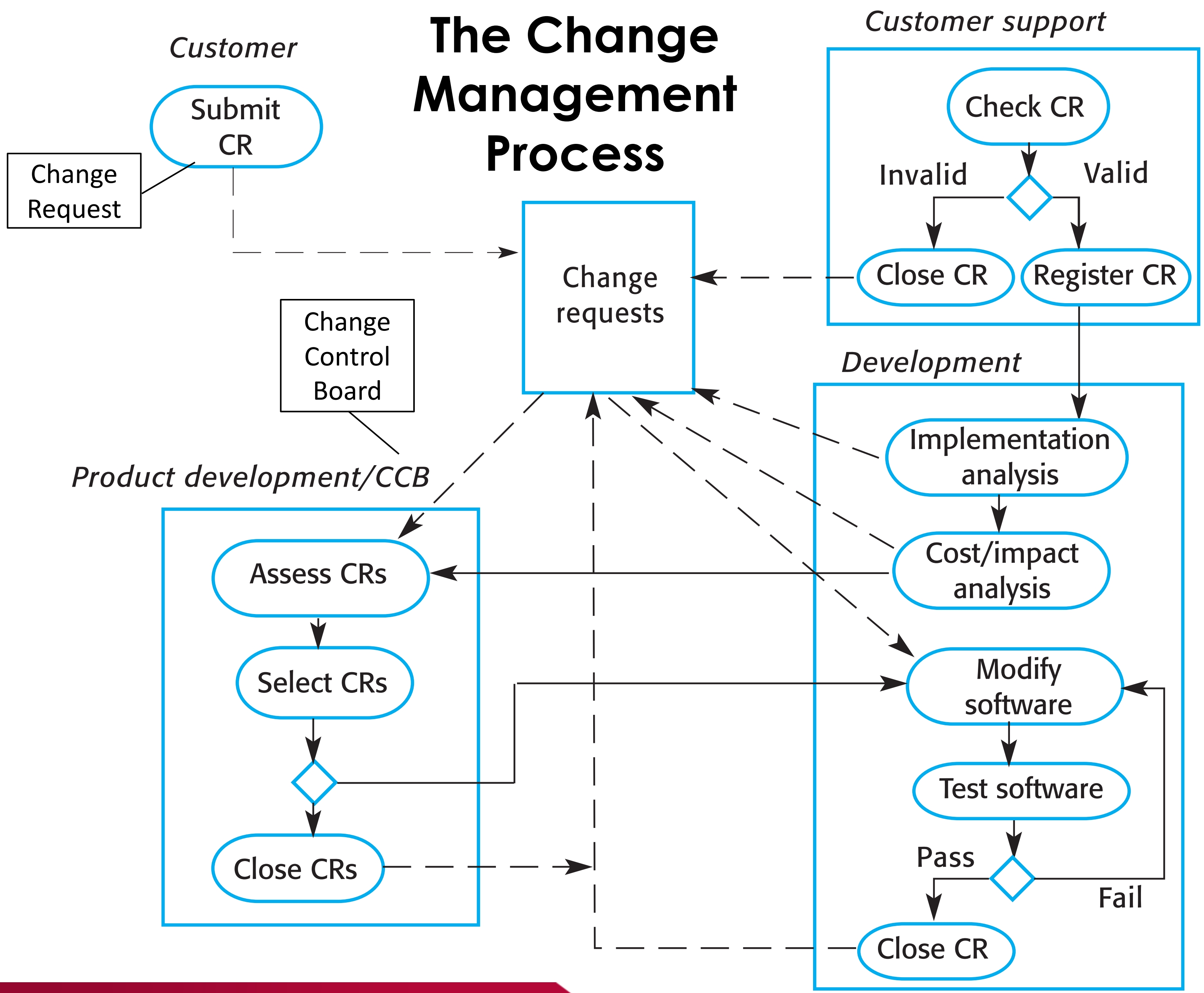
Release tracking

- In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
  - You have to record the specific versions of the source code components that were used to create the executable code, keep copies of the source code files, corresponding executables and all data and configuration files, and also record the versions of the operating system, libraries, compilers and other tools used to build the software
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

Change management

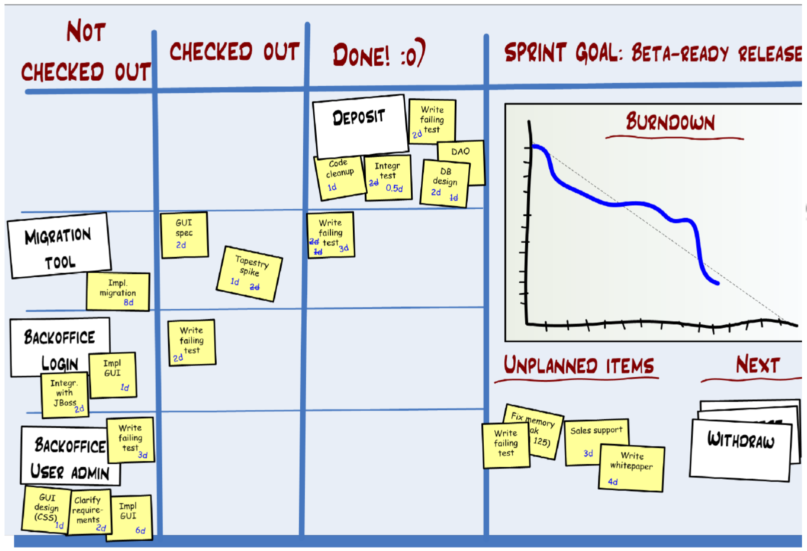
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.
- Factors in change analysis
  - The consequences (risks) of not making the change
  - The benefits and risks of the change
  - The number of users affected by the change
  - The costs of making the change
  - The product release cycle

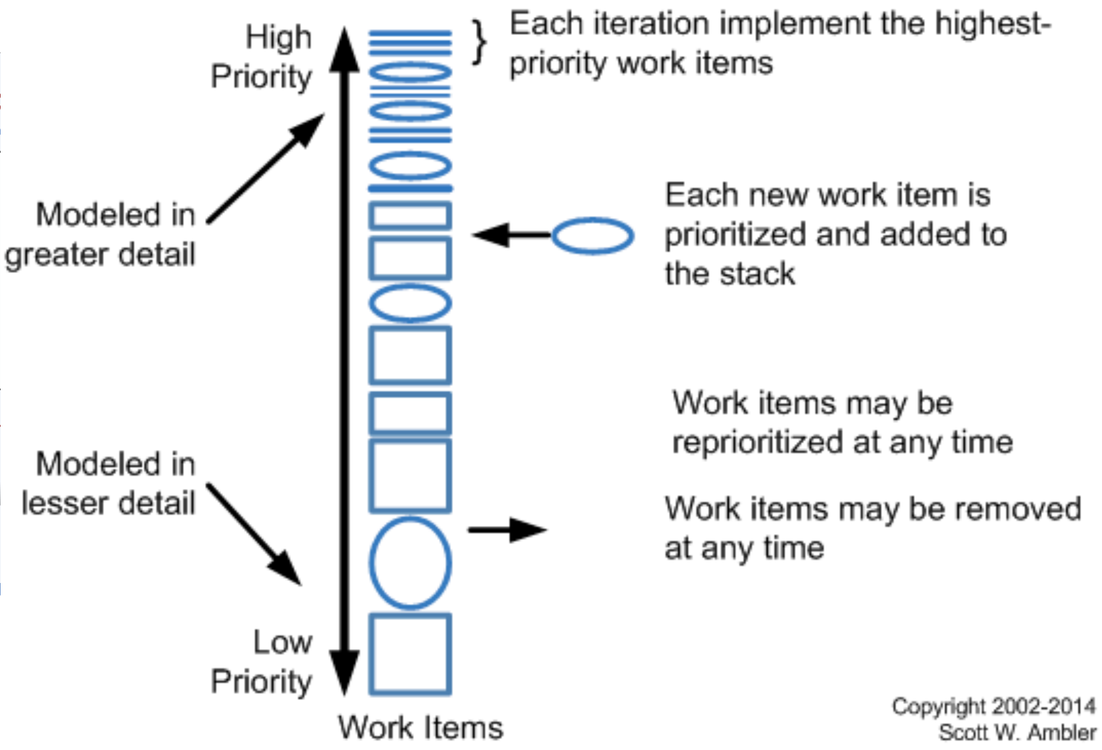
The Change Management Process



Compare with Agile change management

- ✧ Scrum product backlog strategy
- ✧ Work item stacks/queue





High Priority
Low Priority
Work Items
Modeled in greater detail
Modeled in lesser detail
Each iteration implement the highest-priority work items
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Work items may be removed at any time
Copyright 2002-2014 Scott W. Ambler

Configuration management begins with requirements

- We don’t wait for code, builds and releases to set up configuration management processes and tools
- Since we must trace our code back into requirements, our change management process begins as we generate requirements and map those requirements to software elements.
- Our software elements are first envisioned as abstract models of what the software must do and how the software will do it.
- To create those models, software modeling is used.

Classroom Activity – within your EMSS project teams

Activity:

Develop UML Sequence, Activity or State Machine diagrams for each of your use cases. Be sure that the use case steps and data you are using is evident in whichever diagram you use.

Readout to the class should be your criteria for modeling particular use cases with a Sequence diagram, an Activity diagram or a State Machine diagram. Give this some thought since while sequence diagrams capture give-and-take among software entities really well, activity diagrams do a better job of capturing process-like flows, and State Machine diagrams capture embedded system activity really well.

Submission to complete the Canvas Assignment EMSS Week 4:

Document each of your use cases using either a UML State Chart, UML Sequence Diagram or a UML Activity Diagram and submit those diagrams with your Use Case diagram. (The Use Case diagram acts as the table of contents).

Quiz 1 results

Average Score was 87%

High score was 95%, low score was 74%

Two troublesome questions:

1. In requirements discovery, scenarios and use cases **document an interaction that occurs with the software system. (**They don't distinguish user from system interactions.)
2. All but the smallest of software products will contain multiple individual software modules.